ADAPTIVE FUNCTION MODAL LEARNING NEURAL NETWORKS

BY

MIAO KANG

A THESIS SUBMITTED IN PARTIAL FULFILMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTING

FEBRUARY 2011

ABSTRACT

Modal learning method is a neural network learning term that refers to a single neural network which combines with more than one mode of learning. It aims to achieve more powerful learning results than a neural network combines with only one single mode of learning. This thesis introduces a novel modal learning Adaptive Function Neural Network (ADFUNN) with the aim to overcome the linear inseparability limitation in a single weight layer supervised network. Adaptation in the function mode of learning within individual neurons is carried out in parallel with the traditional weights adaptation mode of learning between neurons; thus producing a more powerful, flexible form of learning.

ADFUNN employs modifiable linear piecewise neuron activation functions and meanwhile adapts the weights using a modified delta learning rule. Experimental results show the single layer ADFUNN is highly effective at assimilating and generalising on many linearly inseparable problems, such as the Iris dataset, and a natural language phrase recognition task. A multi-layer approach, a Multi-layer ADFUNN (MADFUNN) is introduced to solve highly complex datasets. It aims to find a suitably restricted subset of neuron activation functions which has a good representational capacity and enables efficient learning for complex models with large datasets. Experiments on analytical function recognition and letter image recognition are solved by MADFUNN with high levels of recognition. In order to further explore modal learning, ADFUNN is combined with an unsupervised modal learning neural network called Snap-Drift (Palmer-Brown and Lee) to create a Snap-drift ADFUNN (SADFUNN). It is used to solve an optical and pen-based handwritten digit recognition task from the UCI machine learning repository and exhibits more powerful generalisation ability than the MLPs.

An additional benefit of ADFUNN, as well as a MADFUNN and SADFUNN, is that the learned functions can support intelligent data analysis. These learned activation function curves reveal many useful information about the data.

ACKNOWLEDGEMENT

First of all, I would like to thank Faculty of Computing, London Metropolitan University for providing resources and support for my research.

And the most important acknowledgement is to my project supervisor Professor Dominic Palmer-Brown. I would like to extend my heartfelt gratitude to him for his guidance, patience, encouragement and kindness from the beginning to the final level of my PhD. His enthusiasm and insights in the subject, incredible talent, energy and personality have not only inspired me throughout the project, but also will motivate me every day for the rest of my career.

Special thanks to Dr Vincent Hargy's great feedbacks and proof-reading to this thesis.

I would also like to thank my parents for their support and encouragement, also for creating an environment where I can follow my dreams.

Last but not least, Songtao, the very special person you are. Thank you for your support and love over the past few years which have made the completion of this thesis possible.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

This research looks at neural network supervised learning methods which have advantages that lead to efficient learning in terms of power, flexibility and the ability to solve complex nonlinear problems.

The general idea of this project is to enhance a single neural network or module by equipping it with several modes of learning to achieve powerful results. It focuses on investigating and combining novel activation function modifiable learning methods with the conventional adaptive pattern of connections between neurons, to overcome the linear inseparability limitation, in a single weight layer supervised network.

In this chapter, some general fundamentals of neural networks are introduced, in order to provide a context for the literature review in the following chapter.

1.1 NEURAL NETWORKS

1.1.1 The Definition of Neural Network

A neural network is a distributed network which simulates the structure of human brain with fully interconnected artificial neurons working together to produce an output. It is like a multiprocessor computer which has very simple processing neurons and synapses but a very high degree of interconnection.

1.1.2 The Appeal of Neural Networks

Whenever it is not possible to formulate an algorithmic solution to the data, or the data or the algorithm is too complex to be processed by either humans or other computer techniques, neural networks are worth considering. The ability to learn complex relationships between

input and output patterns that would be difficult to model with conventional algorithms is the appeal of neural networks. This ability can significantly simplify the modelling work involved in data analysis. With their excellent ability to find meaningful patterns in complicated or imprecise data, neural networks have very broad applicability to be used to extract patterns and detect trends in many real world problems. For instance, they can be widely applied in industry process control [1]; environmental forecasting [2]; financial and accounting predictions[3] like sales forecasting, target marketing prediction, risk management and credit evaluation; medical diagnostics [4] like disease recognition; and speech and face recognition [5, 6].

1.1.3 Computation in the Human Brain

In the nervous system, a neuron is the basic computational unit. It receives thousands of inputs from other neurons, computes the input's sum and discharges an electrical pulse down the axon to the next neurons when this input exceeds a critical level. The transmission site of the electrical signal passing from one neuron to another is called a ***synapse***. Brains can learn either by modifying the weights of connections between neurons or by inserting or deleting connections between neurons. The learning is 'on-line' and based on experience. However, there are still many things unknown or unsure about how the brain trains itself to process information [7] .

1.1.4 Neural Networks in the Brain

The bundles of neural connections between the difference parts of the brain (cortex, midbrain, brainstem, and cerebellum) are extremely complex, and they are only partially discovered or known by now. So far only feed-forward projections that go from earlier processing stages to later ones can be distinguished from feed-back connections that go in the opposite direction.

In addition to these long-range connections, neurons also link up with many thousands of their neighbours. In this way they form very dense, complex local networks.

1.1.5 A Brief History of Neural Networks

Many neural network applications seem to have been introduced and developed quite recently. However, this field was established even before the advent of computers. It experienced at least one major setback and several eras.

In 1943 [8], the neurophysiologist Warren McCulloch and the logician Walter Pits produced the first artificial neuron. They demonstrated that: "networks of simple interconnected binary units which they called 'formal neurons', when supplemented by indefinitely large memory stores, were computationally equivalent to a Universal Turning Machine". During that period, even funding and professional support were limited, some researchers still delivered convincing technologies [9].

However in 1969, Minsky and Papert [10] published a book, called 'Perceptrons', in which they summed up a general feeling of frustration against neural networks among researchers. Without further analysis, their theory was accepted. They even demonstrated that: "the single layer perceptron which was introduced by Frank Rosenblatt [11, 12 and 13] was restricted to learn classes of patterns that were linearly separable only". The publication of 'Perceptrons' was the only factor in the decline of network research in the late sixties and early seventies. A number of apparently significant research successes from the non-network approach, also proved to be influential.

After a ten year gap, neural networks come back to the stage. One important reason for this was because a number of technical developments were made which seemed to indicate that Minsky and Papert were wrong and their publications turned up to be very premature and too early to be concluded.

To overcome the linear separablity limitation, many researches were carried out in response to Minsky and Papert. The classic multilayer supervised back-propagation method was then introduced .

A brief history of neural networks can be summarised as the following milestones:

- Research started in the 1950's and 1960's by researchers like Rosenblatt (Perceptron), Widrow and Hoff (ADALINE) [14].
- In 1969, publications from Minsky and Papert effectively ended the interest in neural network research.
- In the late 1980's interest in neural network increased with the success of algorithms like Back Propagation, Hopfield, Cognitions and Kohonen. (Many of them developed quietly during the 1970s) [15, 16, 17 and 18] and also the parallel distributed processing model [19, 20].
- Progress continued during the 1990's with more emphasis of Bayesian statistics.
- Currently neural networks are widely used in commercial applications like character recognition, image recognition [5, 6], credit evaluation [3], fraud detection [3], insurance [3], and stock forecasting [3] etc [1, 2, 4].

1.1.6 Model of a Simple Active Neuron

In neural networks, a neuron represents a model of a neural cell in the brain. It is the basic processing element which has one or more inputs and produces one output in a neural network. The inputs simulate the signals that a neuron gets and the output simulates the signal which the neuron generates. The output is normally calculated by the sum of weighted inputs, being passed through an activation function (sigmoid function in this example), and being scaled to a number between 0 and 1.

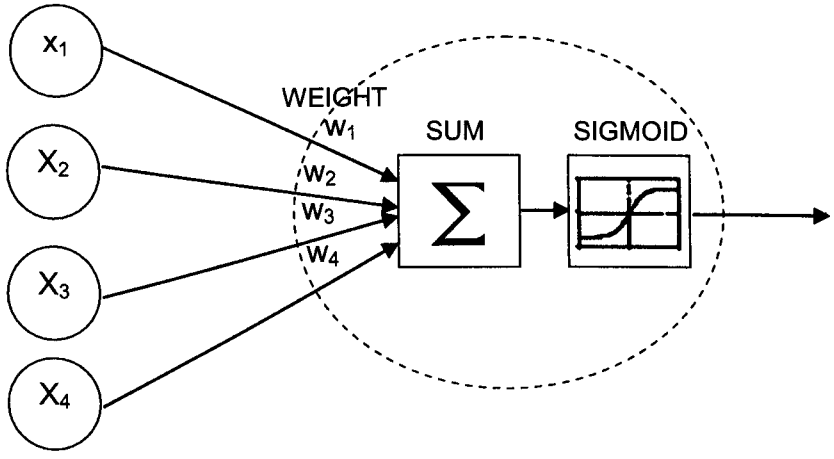


Figure 1: A simple processing active neuron

An active neuron is used in hidden and output layers of the neural network. However, the neuron used in the input layer does not normally have an activation function; the input will only be weighted and passed to next layer neurons. An example of simple processing active neuron is shown in figure 1.

1.1.7 Modelling a Simple Neural Network

A neural network is a group of neurons connected together. The most commonly used structure of a neural network is formed in three layers, the **input layer, hidden layer, and output layer**. There is at least one neuron in each layer. In this type of neural network, the information flows only from the input to the output. However, other types of neural networks may have more complex connections, like feedback paths. A simple three layer neural network is shown in figure 2.

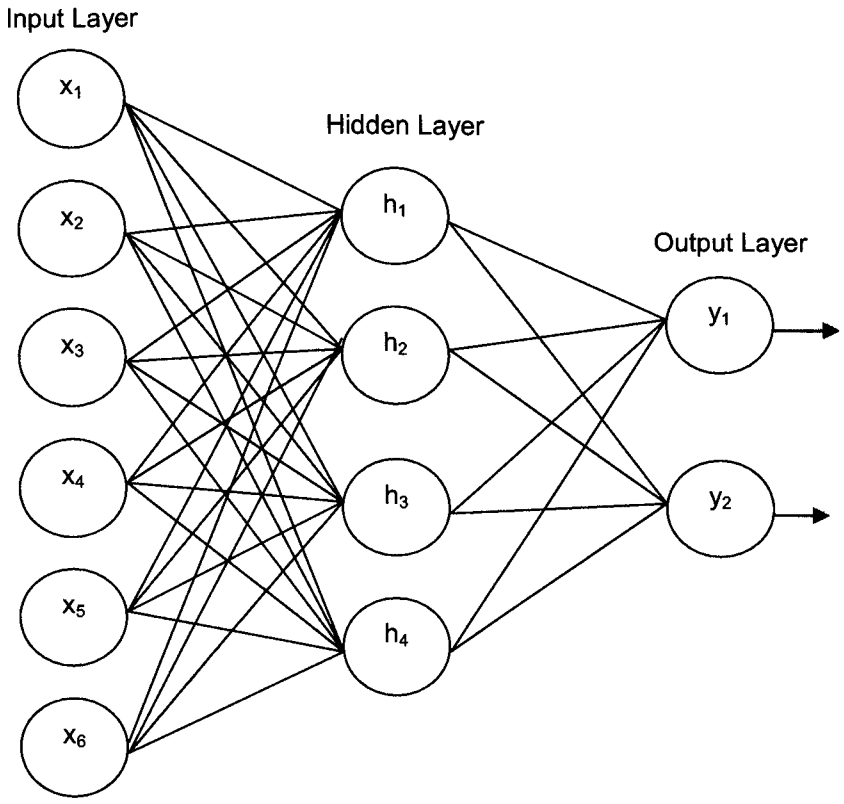


Figure 2: A simple three layer neural network

Each value from input layer is duplicated and sent to all of the hidden neurons, while the values entering a hidden neuron are multiplied by weights and then summed to produce a

single number. This number is then passed through an activation function to generate an output. Each of these outputs from the hidden layer is duplicated and applied to all of the output neurons. The active neurons in the output layer will combine and modify the data to produce outputs values of this network.

1.1.8 The Learning of Neural Networks

In adaptive neural networks, learning is the process of adjusting the weights to proper values. There are two major learning methods categories: supervised learning and unsupervised learning. In a supervised learning neural network, the learning starts by initialising the weights for all neurons to the random numbers normally between 0 and 1, then

1. Present input to the network
2. Calculate the output
3. Calculate the error by compare the actual output with the expected output for the given input
4. Adjust the weights according to the error
5. Repeat 1-4 until error gets to an acceptable value.

While in unsupervised learning, the network self-classifies data applied to the network and automatically detects their emergent collective features.

There are other learning methods like associative mapping, in which the network learns to produce a particular pattern on the set of input neurons whenever another particular pattern is presented on the set of input neurons, for instance, the nearest–neighbour learning. Also learning like regularity detection, neurons learn to respond to particular features of the input patterns. In associative mapping the network saves the associations among patterns whereas in regularity detection the response of each unit has a particular meaning. In some methods, the weights are fixed before hand according to the problem to solve instead of being adapted.

1.2 PATTERN CLASSIFICATION

An important application of neural network is pattern recognition [21]. It is widely used, also known by the names of 'classification', 'diagnosis' and 'learning from examples'.

Pattern Classification [22, 23, 24 and 25] aims to classify patterns based on either a priori knowledge or on statistical information extracted from the patterns. The patterns used for classification are usually groups of measurements or observations, defining points in an appropriate multidimensional space.

1.2.1 *Single Layer Network for Pattern Classification*

A single-layer neural network as shown in figure 3 consists of a set of neurons organised in an input layer. They are fully connected to a neuron (or multiple neurons) in the next output layer. Each neuron in output layer takes a weighted sum of all its inputs x_j with weight w_{ji} and provides one network output o_i.

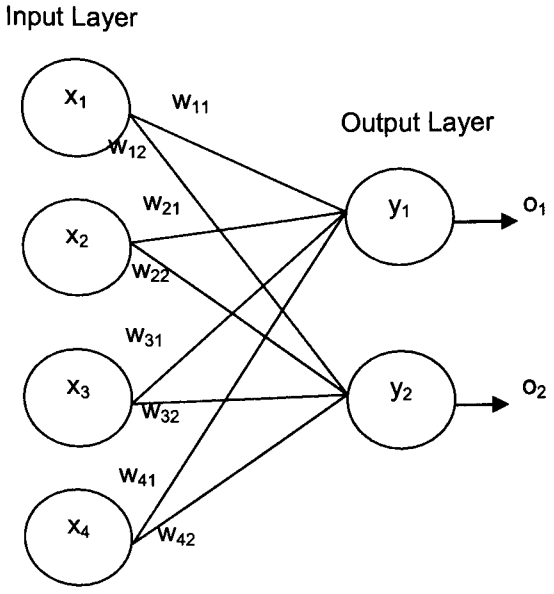


Figure 3: A single layer neural network

1.2.2 *Multi Layer Network for Pattern Classification*

Multi-layer networks are multiple layers links of input, hidden and output neurons. Hidden neurons are so called because they are not visible from either input or output.

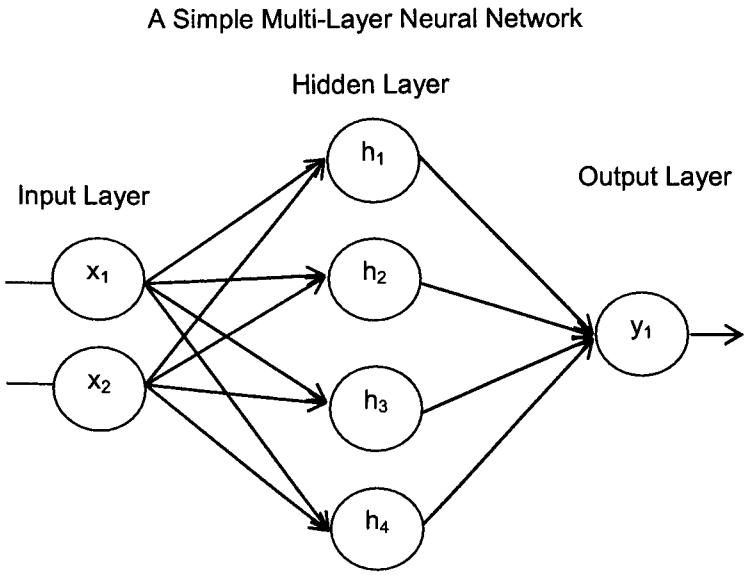


Figure 4: A simple multi-layer neural network

The network diagram shown in figure 4 is a simple fully-connected multi-layer neural network with only one hidden layer. All multi-layer neural networks have an input layer and an output layer, but the number of hidden layers may vary. Input can be a vector and so is output. There may be any number of hidden nodes. When there is more than one hidden layer, the output from one hidden layer is just simply fed into the next hidden layer.

The term 'fully connected' means that the output from each input and hidden neuron is distributed to all of the neurons in the following layer. But if a weight is adapted to zero, then that connection may as well not exist.

1.2.3 Linear Separability

Classification problems for which there is a line that exactly separates the classes are called linearly separable as shown in figure 5. Most single layer networks are only able to solve linearly separable problems and most real world problems are not linearly separable. However, the work of this thesis questions this by solving linearly inseparable problems using a single layer network. The detail of the approach will be introduced and explained in Chapter 3.

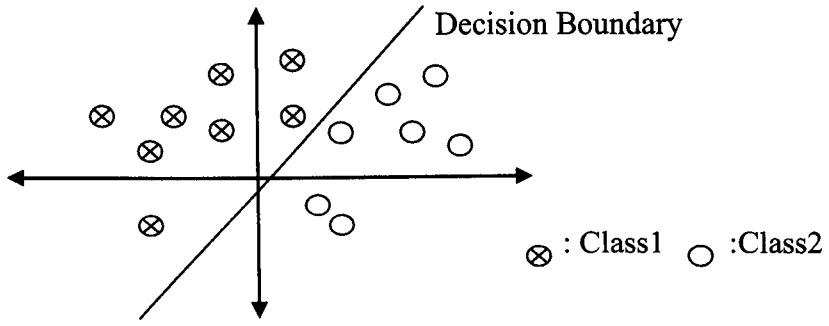


Figure 5: A linear separable problem

The simplest well known linear inseparable problem is the XOR problem (in figure 6). Here two straight line boundaries are needed to separate the two classes of 0 and 1.

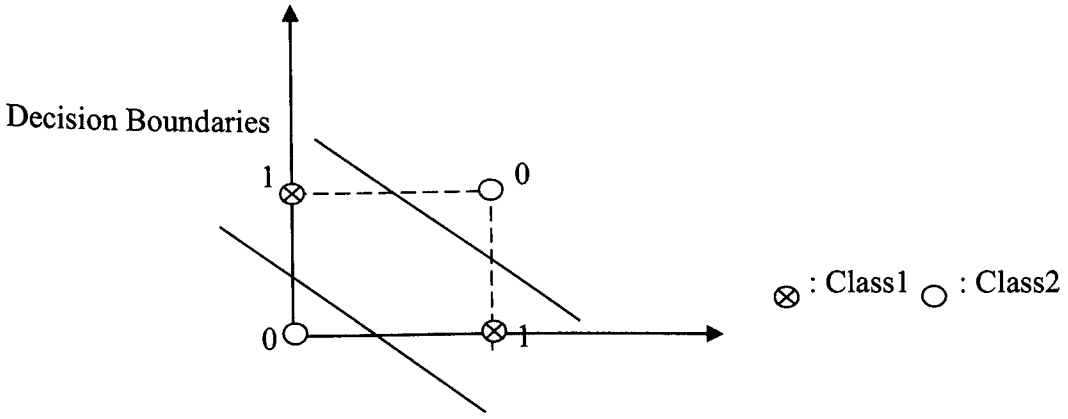


Figure 6: A linear inseparable XOR problem

1.2.4 Perceptrons

The perceptron is a type of artificial neural network invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt [11, 12 and 13]. It can be considered as the simplest kind of feed forward neural network, the linear classifier.

The perceptron is more like a binary classifier that maps its input vector x to an output f(x), calculated as

$f(x) = \langle w, x \rangle + b$

where w is a vector of weights and $\langle w, x \rangle$ computes a weighted sum. b is the "*bias*", a constant which does not depend on any input value.

The perceptron is considered to be equivalent to an artificial neuron with weighted inputs and some additional, fixed, pre-processing. Although Perceptrons have been mainly used in pattern recognition, their capabilities can be extended a lot further in other areas, like Natural Language Parsing [26] and feature extraction [27].

1.2.5 Perceptron Learning Algorithm

The perceptron learning rule was originally developed by Frank Rosenblatt [11, 12 and 13] in the late 1950s. Training patterns are presented to the network's inputs and the output is computed. Then the connection weights are modified by an amount that is proportional to the product of the difference between the actual output, the desired output and the input pattern. This is to say, the perceptron learning uses iterative updating of weights to learn a correct set of weights which minimise the errors to achieve $de/dw = 0$ (de/dw is the error derivative of the weights('e' for error and 'w' for weight).

The algorithm can be summarised as:

1. Initialise the weights and threshold to small random numbers
2. Present an input pattern as the neuron inputs and calculate the outputs
3. Update the weights
   $w_j(n+1) = w_j(n) + \eta \bullet Error \bullet A$
   where error is the difference between expected output and real output, η is the learning rate and A is the input vector
4. Repeat step 2 and 3 till the error is less than a user-specified error threshold.

1.3 SUPERVISED AND UNSUPERVISED LEARNING

The learning algorithm of a neural network can either be supervised or unsupervised. For a set of inputs, if the correct set of outputs used to train the network is given in the database, it is a supervised neural network. Inputs are applied to the input layer in the network and outputs are generated by the output layer. Outputs are then compared to the target outputs. An error value is computed based on the difference between output and target. The greater the computed

error value is, the more the weight values will be changed.

However, in unsupervised learning, there are no target outputs. It can't be determined what the result of the learning process will look like. During the learning process, the weight values of such a neural net are arranged inside a certain range, depending on given input values. Many unsupervised methods are similar to clustering [28].

1.4 MODAL LEARNING

Since neural network became popular, many artificial neural network algorithms and architectures have been developed. Along the way, many remarkable forms of learning are presented, including Back-propagation (BP), Bayesian [29] and Kernel methods [30]. However, none of the methods have dominated with each method showing strength in different application.

The term "Modal Learning" was introduced into neural computing by Palmer-Brown [31] with the aim of achieving powerful learning results by equipping a single neural network or module with the combined power of several complementary modes of learning. In modal learning, a mode refers to an adaptation learning method that could be applied in more than one type of architecture or network. Widely known modes include Delta Rule [13], BP [33], Learning Vector Quantisation (LVQ) [31, 32], and Hebbian learning [34]; but not Adaptive Resonance Theory (ART) [35] or Bayesian Neural Networks [29], since they define architectures and approaches to learning.

Modal Learning (ML) differs from hybrid and modular approaches. In hybrid and modular approaches, every module or network utilises one mode, and modules are bolted together with each designed to solve a sub-problem. Whereas, in Modal Learning, more than one mode of learning is working on the same problem in one network. Modes in a hybrid and modular approach are less able to work cooperatively

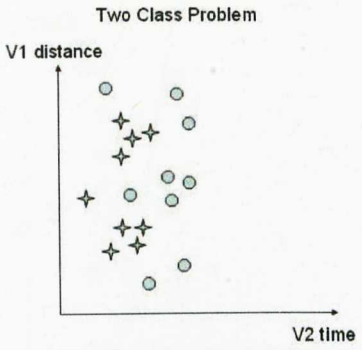


Figure 7: Two class problem in a 2D environment

Figure 7 is a classic 2D two class problem which can be used to illustrate some of the advantages of the modal learning approach. The following example illustrates the potential benefits of a modal learning approach in the sequence of class boundaries. It can be seen how necessary it is to increase complexity to an extent that is not justified by the data in order to find a single mode solution from step 1 to step 4 (in Figure 8). Also as can be seen from the sequence solutions, the geometrical margins are very small along the separating curve which suggests poor generalisation and over fitting.

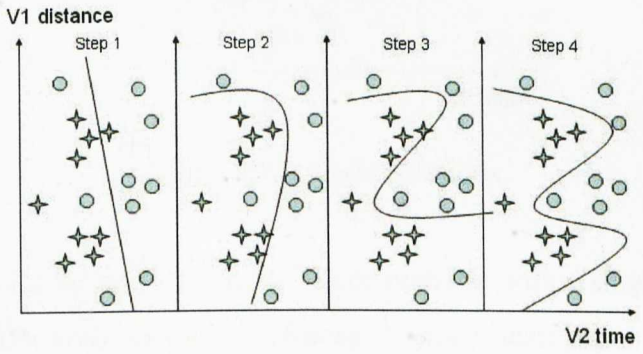


Figure 8 Increasingly complex solution to a 2-class problem

Alternatively, a relatively simple and good margin solution can be achieved by combining a straight line (perceptron), a simple curve (multilayer perceptron) and a cluster. This requires 3 modes of learning, as in figure 9:

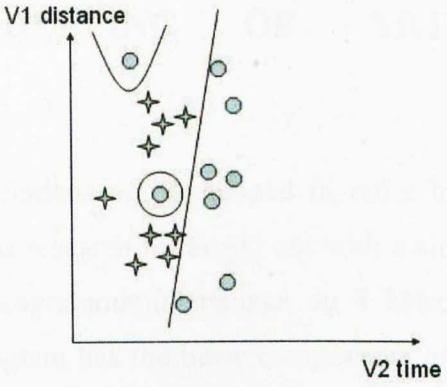


Figure 9: 3 mode solution

Or by combining a simple curve (multilayer perceptron) and a cluster. This requires 2 modes of learning (as shown in figure 10):

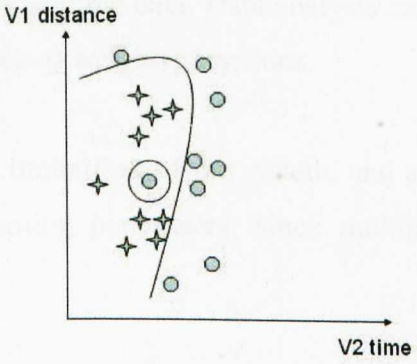


Figure 10: 2 mode solutions

It is clear that rather than trying to solve the whole problem with a single mode of learning, a simpler learnt solution is achievable by combining modes of learning.

When looking at human and machine learning in a wider context, there are many reasons and motivations to consider modal learning, as it allows for the spectrum of learning to be taken into account, from memorisation to generalisation.

1.5 THE MODELLING OF ARTIFICIAL NEURAL NETWORK

In this thesis, software simulations are developed in order to model the behaviour of an artificial neural network. This research is carried out with a simulator program implemented by the author using Java programming language on a Microsoft Windows XP platform computer. The simulator program has the basic components of the proposed neural network architecture: the input patterns as the input data, the output as the output data of the program and the activity of the network are the processes in the program.

The powerful GUI design tool in Java enables the simulations with helpful visual representations of the neural network, for example input nodes, hidden nodes, output nodes, the performance of the network and the data. Data analysis can be investigated with the help of visual simulation results, such as weights, functions.

Moreover, it helps to find the limitations of the system, and any suitable parameters such as weights initialisation and learning parameters, since multiple simulation runs are easily achieved.

1.6 OBJECTIVES OF THIS THESIS

This thesis aims to create an innovative new Modal Learning method by adapting the activation function inside each neuron in parallel with the traditional weights adaptation between neurons in a single layer neural network structure.

The new approach is expected to overcome linear inseparability limitations of single layer neural networks, as a single layer neural network is incapable of solving linear inseparable problems due to the lack of enough neurons. By combining the power of both function adaptation and weights adaptation, this new approach aims to produce powerful performance improvements, significantly speed up the learning process, simplify the hardware structure and reduce the hardware requirements, compare with traditional and related approaches.

Additional benefits are also expected from the learned results of this new approach. Traditionally, intelligent data analysis using neural networks only looks at the learned weights because the activation function shape is fixed. However, this new approach can produce both learned weights and functions. It should offer a more intelligent way to interpret data and this interpretation can be modelled to a generic method which offers speedy, accurate and efficient data analysis.

1.7 THE STRUCTURE OF THE THESIS

In the next chapter, a review of the work from other researchers will be discussed to justify the work produced in this thesis. Both the strong and weak points of related works are introduced and compared specifically with the intended work of this thesis. It is then followed by a chapter on the proposed learning algorithm, ADFUNN, and its system architecture. It includes the explanation of how and why the system is chosen and the learning algorithms of ADFUNN. It also summarises a general learning rule for ADFUNN. The simulations and experimental results of ADFUNN on different datasets are then described and compared with other networks. Datasets applied to ADFUNN include: classic XOR problem, iris dataset, and a natural language processing phrase recognition task.
In Chapter 4, an extended multi-layer ADFUNN (MADFUNN) is proposed. It explains the reason to extend ADFUNN and how is it constructed and learns. A general learning rule for MADFUNN is also presented. MADFUNN is applied to two complex datasets: analytical function recognition and letter image recognition tasks. Generalisation abilities are discussed and compared to other methods on the same data.

In Chapter 5, a multi modal combination of ADFUNN with another modal learning method, Snap-Drift is introduced to solve the optical and pen-based handwritten digit recognition tasks from the UCI repository. Generalisation is compared with other methods on the same data.

Chapter 6 introduces a complementary benefit of the approach, intelligent data analysis. It describes how to find intelligent solutions and new ways of looking at problems from the learned system.

In the final chapter, the research from the earlier chapters is reviewed. A final discussion and conclusion is presented together with a section on some proposed research that provides a logical progression of the research work in the thesis.

CHAPTER 2 A SURVEY OF RELATED WORK

2.1 INTRODUCTION

In this chapter, the advantages of related learning methods, network architectures, data analysis methods and their limitations are discussed. It provides a platform for the following chapter where a new approach is introduced to overcome limitations and enhance advantages.

2.2 LEARNING ALGORITHMS

In general, there are two approaches for machine learning. One is the symbolic approach and the other is the non-symbolic approach. A Decision tree is a typical model for symbolic learning, and an artificial neural network (ANN) is the most popular approach for non-symbolic learning.

2.2.1 Symbolic Learning

Symbolic learning algorithms are determined by a set of rules that form a relationship between the attributes and classes. There are a variety of symbolic learning algorithms and they vary in the way that they construct these rules.

2.2.1.1 Decision Tree

A decision tree [36] is one of the symbolic learning algorithm examples. A decision tree is "a decision support tool that uses a model of decisions and their possible consequences" [37].

In the data mining and machine learning area, a decision tree [38] maps from one item's observations to conclusions about its target value. In the tree structure, leaves represent classifications and branches represent conjunctions of features that lead to those classifications.

However, the basis of decision tree learning [39] is that the tree is induced from a set of labelled training instances represented by a group of pairs of attribute values and a class label. Because of the huge searching space, the decision tree learning is typically a greedy, top down and recursive process starting with the entire training data and an empty tree. Therefore, if the tree is too thick (with plenty branches and leaves), the searching can be very inefficient.

Sometimes the model is too complex to be visualised as a single tree. It could be created as a decision tree forest model [40] which contains a set of several decision trees. However, this makes the model rather complicated in the sense of various methods, predictors and parameters being used in the network.

Neural networks can be used to learn decision tree type problems by casting attributes and classes as binary or analogue patterns [41, 42 and 43]. There have been many research efforts in converting decision trees into equivalent neural networks [44 and 45] that classify accurately as the input decision trees.

2.2.1.2 Rule Induction

In machine learning, rule induction [46] introduces formal rules which are extracted from a set of observations. As rule induction is sensitive to the order of data, it is not always suitable for data with noise [47]. One good example of rule induction is the CN2 induction algorithm.

2.2.1.2.1 CN2 Induction Algorithm

The CN2 algorithm [48] was developed by Peter Clark. The CN2 algorithm induces an ordered list of classification rules from examples using entropy as its searching guide in solving a problem [49]. It uses the classic if statement: "if *condition,* then *class*". The algorithm consists of two main procedures: the searching algorithm searches for a good rule to apply and a good control algorithm for repeatedly executing the search.

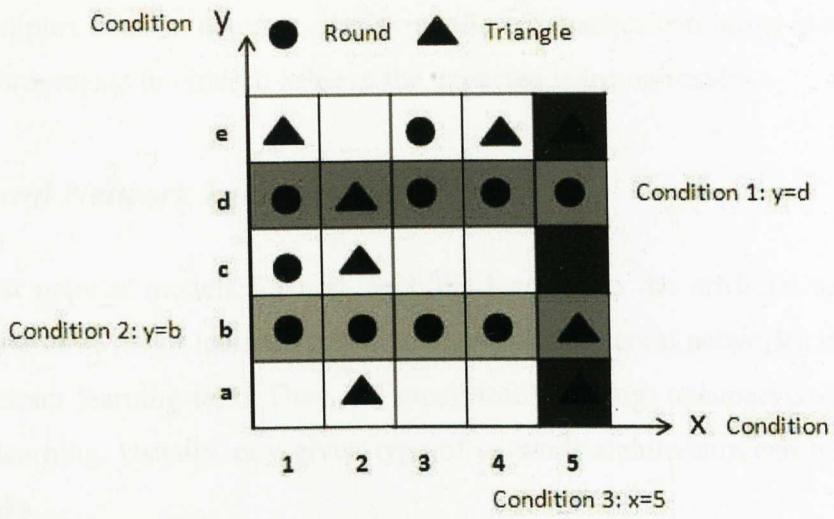


Figure 11 Searching for conditions to classify class "*Round*" and class "*Triangle*"

For example, as illustrated in figure 11, by searching for a condition (y = d in the example) that covers a large amount of examples of an arbitrary class ("*Round*" in this case) and a couple of other classes (one from "*Triangle*" class), a rule (condition 1) is conducted. This condition is then added as a rule "*if condition then predict A*" to the end of the list and the algorithm removes those examples it covers from the training set. The procedure is repeated for the remaining set, as new rules are constructed until no further conditions of sufficient quality can be found.

As the size of data increases, the time consumption [50] can be a big issue when the CN2 algorithm (as well as other rule induction methods) is the essential symbolic learning method. The system efficiency is significantly reduced because of the huge consumption of learning time and system memory. Besides, the performance of using a set of rules will be limited and depended on whether the problem is solvable by the rules. The rule induction methods will not be effective if the required solution is a fizzy, probabilistic or weighted combination of attributes.

2.2.2 Non-Symbolic Learning

As mentioned before, symbolic and non-symbolic approaches are two main categories in machine learning. In general, symbolic approaches can provide comprehensible rules, but cannot adapt to changing environments efficiently and can be very difficult and inefficient in

dealing with complex data. In contrast, non-symbolic approaches can adapt more effectively to changing environments in order to achieve the expected learning results.

2.2.2.1 Neural Network Learning Paradigms

One of the most popular models for non-symbolic learning is the artificial neural network (ANN). There are three major learning paradigms in artificial neural networks; each addresses a particular abstract learning task. They are supervised learning, unsupervised learning and reinforcement learning. Usually, any given type of network architecture can be employed in any of these tasks.

2.2.2.1.1 Supervised Learning

In supervised learning, a set of paired *(x, y)* examples are given and the aim is to find a cost function which maps from x to y in the allowed class of functions that matches the examples. In other words, the expected output of the network is already known for each pattern and it is expected to discover how the mappings are implied by the data [51].

2.2.2.1.2 Unsupervised Learning

Unsupervised neural networks learn on their own as a form of self-organisation. In unsupervised learning the input data and an activation function are given, but the network's output is not given. The network learns to recognise patterns in the data set to categorise into groups [51].

However, the network does not label these groups. It is up to human users or another method to interpret or label the groups in some meaningful way.

2.2.2.1.3 Reinforcement Learning

"Reinforcement learning is learning what to do and how to map situations to actions [52]". It differs from the supervised learning problem in the sense that the correct input and output pairs are not present, nor any alternative actions that might be used. This is because, in many situations, there is less detailed information available. Even in some extreme situations, there

is little information indicating whether the output is right or wrong after a long sequence of inputs. Reinforcement learning is one method developed to deal with such situations. It focuses on the online performance by finding the balance between the unknown knowledge and existing knowledge.

Artificial neural networks are frequently used in reinforcement learning as part of the overall algorithm. Reinforcement learning is mainly applied in control problems [53], games [54] and other sequential decision making tasks [55].

2.2.2.2 Neural Network Learning Algorithms

There are many algorithms for training neural networks; some of them are applications of optimisation theory [56] and statistical estimation [57]. Other commonly used methods for training neural networks include: evolutionary computation methods, simulated annealing [58], expectation maximisation [59]; and, non-parametric methods [60].

2.2.2.2.1 Linear Classifiers

Linear classifiers predate and are related to simple neural network methods.

2.2.2.2.1.1 Fisher's Linear Discriminant

The Fisher linear discriminant (FLD) gives a projection matrix that reshapes the scatter of a data set to maximise class separability. The projection defines features that are optimally discriminating. In other words, the high dimensional data is projected onto a line and so the classification can be simply performed in this one dimensional space. The projection maximises the distance between the means of the two classes while minimising the variance within each class.

It projects data from d dimensions onto a line. For example, in figure 12, two figures project from mixed two class samples onto two different lines.

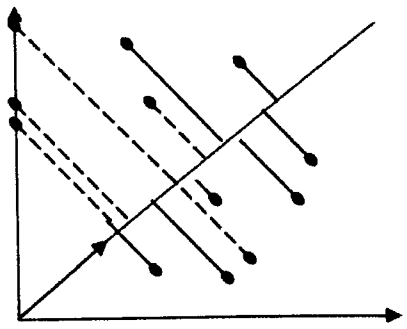


Figure 12: Mixed two classes project onto one line

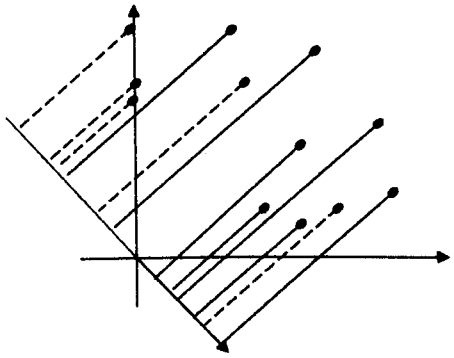


Figure 13: Better separated two classes

Figure 13 shows a better separated two classes. However, the constraint on the number of features available from the Fisher linear discriminant has significantly limited its application to a large class of problems [61]. In other words, FLD works out the projection that best separates the data corresponding to different classes, so the number of features derived is dependent on the number of the classes to be recognised. This disadvantage prevents FLD from classifying problems with large number of classes.

2.2.2.2.1.2 Linear Regression

When people want to look at the relationship between two different things (e.g. between a person's blood type and height), a good way is to use the scatter diagram. Linear regression is the process of finding the straight line that is satisfied by the points on the scatter diagram.

It is to find the line that best predicts y from x. Linear regression does this by finding the line that minimises the sum of the squares of the vertical distances of the points from the line. This

method assumes that the data is linear, and finds the slope and intercept that make a straight line best fit the data as shown in figure 14.

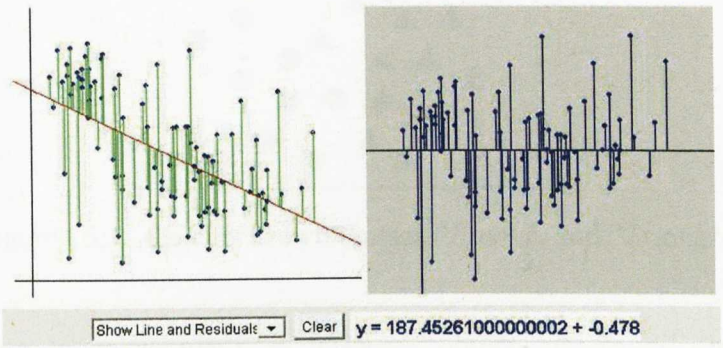


Figure 14: The regression line which separates data (generated using linear regression Java applet [62])

However, linear regression is appropriate only if the data can be modelled by a straight line function, which is often not the case. Also, linear regression cannot easily handle categorical variables nor is it easy to look for interactions between variables.

2.2.2.2.1.3 Naive Bayes Classifier

The Naive Bayes Classifier technique is based on the Bayesian theorem and is suited when the dimensionality of the inputs is high [63]. Its classifier assumes that the presence of a class feature is unrelated to the presence of any other feature.

If the objects in figure 15 can only be classified as either *Round* or *Triangle*, then to classify a new class based on the existing ones using Naïve Bayes Classifier is very easy to understand.

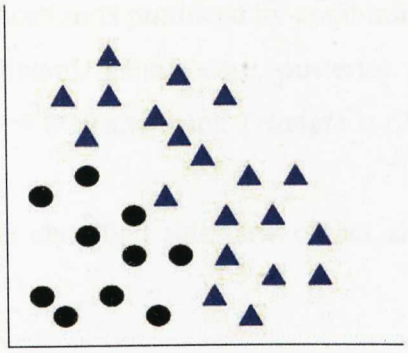


Figure 15: Objects of two classes "*Round*" and "*Triangle*"

Obviously there are twice as many as *Triangle* objects as *Round* ones, therefore it is reasonable to think that a new object is twice likely to be a *Triangle* class than a *Round* class, which is known as prior probability in Bayesian method. Since there are 10 *Round* objects and 20 *Triangle* objects, the prior probability for *Round* is 10/(10+20) = 10/30 and for *Triangle* is 20/(10+20) = 20/30.

Now when a new object arrives as the question mark in figure 16, it is obvious that the objects are well clustered and therefore if more of one class of object are close to the new object, this object is more likely to be from this class (e.g. more *Round* objects around the question mark than *Triangle* objects). The big circle covers all objects around the new object. Hence the likelihood that this new object is *Round* is 3/(10+20) = 3/30 of that it and is *Triangle* is 1/(10+20) = 1/30.

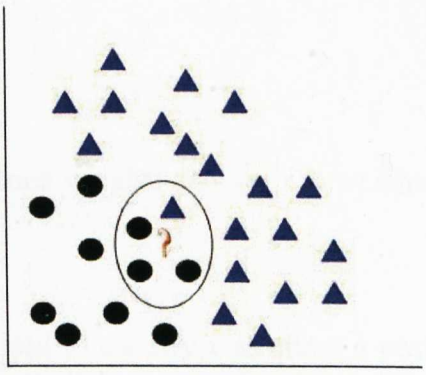


Figure 16: Objects surround the new object

In Bayesian method, the classification is produced by combining both sources of information, e.g. prior probability and likelihood. In this case, posterior probability of this new object being *Round* is *(10/30) x (3/30) = 1/30* and being *Triangle* is *(20/30) x (1/30) = 1/45*.

Finally Naïve Bayes Classifier classified this new object as *Round* since it achieved the largest posterior probability.

Naive Bayes Classifiers are easy to understand as above and if data is discrete the induction can be very fast because only a single pass through is needed for data. However, Naïve Bayes Classifiers require the attributes to be statistically independent, when this assumption is broken; the achievable accuracy will fail to improve as the database size increases. Hence this method does not scale well in larger and complex databases [64].

2.2.2.2.1.4 Perceptron

As introduced in chapter 1, the perceptron is a type of artificial neural network invented in 1957[11, 12 and 13] at the Cornell Aeronautical Laboratory by Frank Rosenblatt. It is one of the simplest forms of feed-forward neural networks and acts as a linear classifier.

The perceptron is a binary classifier [65] that maps its input x (a binary vector) to an output value $f(x)$ (a single binary value) calculated as

$$f(x) = \begin{cases} 1 \text{ if } (w \cdot x + b) > 0 \\ 0 \text{ else} \end{cases}$$

where w is a vector of real valued weights and $w \cdot x$ is weighted sum of inputs , and b is the bias.

The result *(0 or 1) of* $f(x)$ is used to classify x as either a positive or a negative instance, to solve a binary classification problem. The bias is like the offset of the activation function, or gives the output neuron a basic level of activity. If b is negative, then the weighted inputs

must produce a positive value greater than b in order to push the classifier neuron over the 0 threshold. The bias alters the position of the decision boundary.

Because the inputs are presented directly to the output using the weighted connections, the perceptron can also be considered as the simplest feed forward neural network.

Data from a training set are applied to the network one after another. If the actual output is the same as the expected output, it means the result is correct and no change is needed. Otherwise, the weights and biases are updated using the perceptron learning rule. The procedure of passing through all of the input training vectors is called an epoch. The procedure is repeated until errors are minimised. The perceptron network training is then completed. If a pattern not in the training set is presented to the network, the network will generalise to a certain extent by responding with an output similar to target vectors for input vectors close to the previously unseen input pattern.

There are several limitations of perceptron networks. Firstly, the output values of a perceptron can take on only one of two values (True 1 or False 0). Secondly, perceptrons can only classify linearly separable problems. As described before, if a straight line can be drawn to separate the input vectors into their corresponding categories, the input vectors are linearly separable and the perceptron will find the solution. If the input vectors are not linearly separable learning will never reach a point where all input vectors are classified properly. The most famous example of the perceptron's inability to solve problems with linearly nonseparable vectors is the boolean XOR problem.

2.2.2.2.2 *Support Vector Machines*

Support Vector Machines (SVMs) [66 and 67] are an alternative approach to solving the classification problems by constructing an N-dimensional hyper-plane that optimally separates data into two categories. SVMs models are closely related to neural networks [68]. They can perform pattern recognition, regression estimation tasks and non-linearly map their n-dimensional input space into a high dimensional feature space [69]. In this high dimensional feature space a linear classifier is constructed.

In SVMs, *'an attribute'* is used as the predictor variable, and *'a feature'* is a transformed attribute that is used to define the hyper plane. 'Feature selection' is to pick the most suitable representation and 'a vector' is a set of features that describes one case.

SVMs typically generalise well from a small number of examples and do not assume prior knowledge of the probability distribution of the underlying data. They use a particular type of function class: classifiers with large 'margins' in a feature space induced by a kernel. SVMs are computationally intractable when dealing with large problems, and they require a pre-determined system parameter for the kernels. The selection of the kernel function parameters is also addressed as a major practical problem by many researchers [70]. A batch algorithm, SVMs require many repeated presentations of input patterns [71].

2.2.2.2.3 K-Nearest Neighbour

In pattern recognition, the k-nearest neighbour algorithm (*k-NN*) is a method for classifying patterns based on closest training examples in the feature space. It is a supervised learning algorithm where the result of new class is created based on the majority k-nearest neighbour category. The classifier is therefore memory-based. For a given object to be classified, the method finds k objects closest to the query object. The classification votes the majority among the classification of the k objects.

K is normally a positive integer, typically very small. For example if k is 1, then the pattern is simply assigned to the class of its nearest neighbour.

The k-nearest neighbour classifier labels an unknown pattern with the label of the majority of the k nearest neighbours. A neighbour is believed nearest if it has the smallest distance, in feature space. In neural networks, this is analogous to the winning neuron, with the highest activation. For example, the data points in figure 17 can either be blue (diamond shape) or pink (square shape) objects:

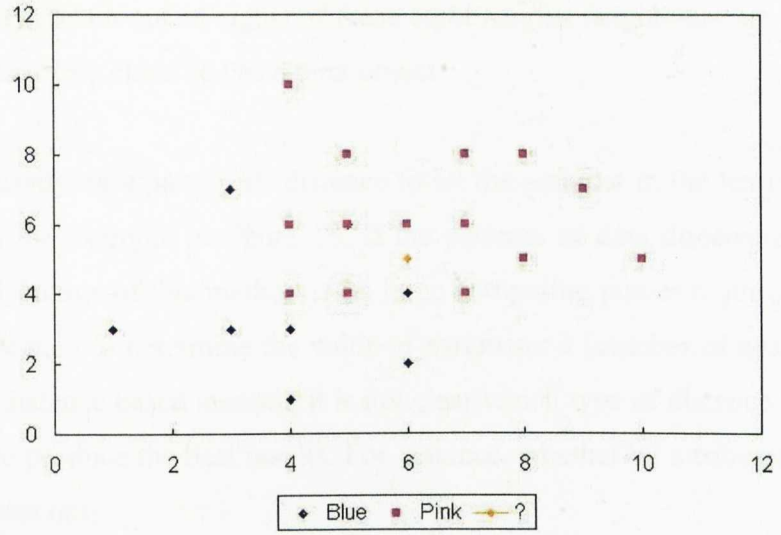


Figure 17: Random blue or pink data

To classify whether the unknown yellow object is blue or pink, a parameter of k must be determined. Suppose 8 nearest neighbors are used, then the distance between all training objects and the unknown object are calculated using Euclidean Distance [72]. However, if the data contains more than two variables, a weighted distance will be calculated instead. The eight nearest neighbours are found in figure 18 to the query object:

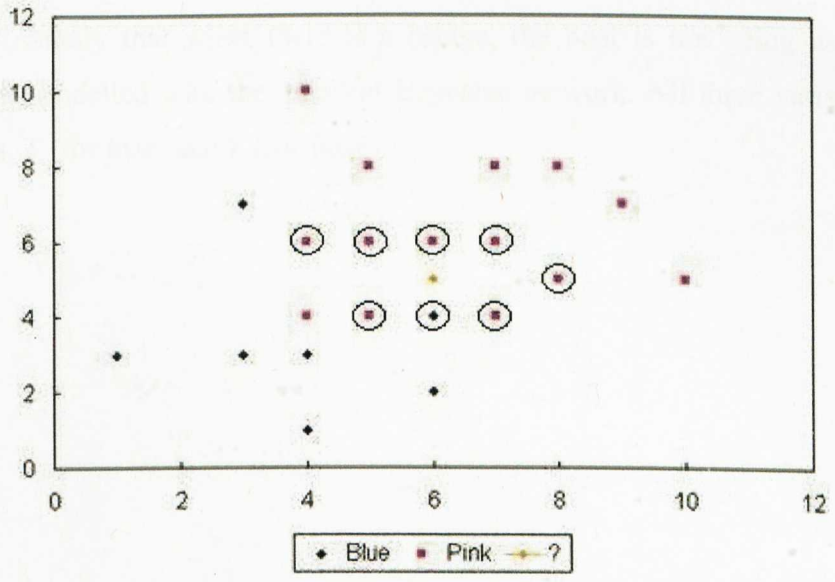


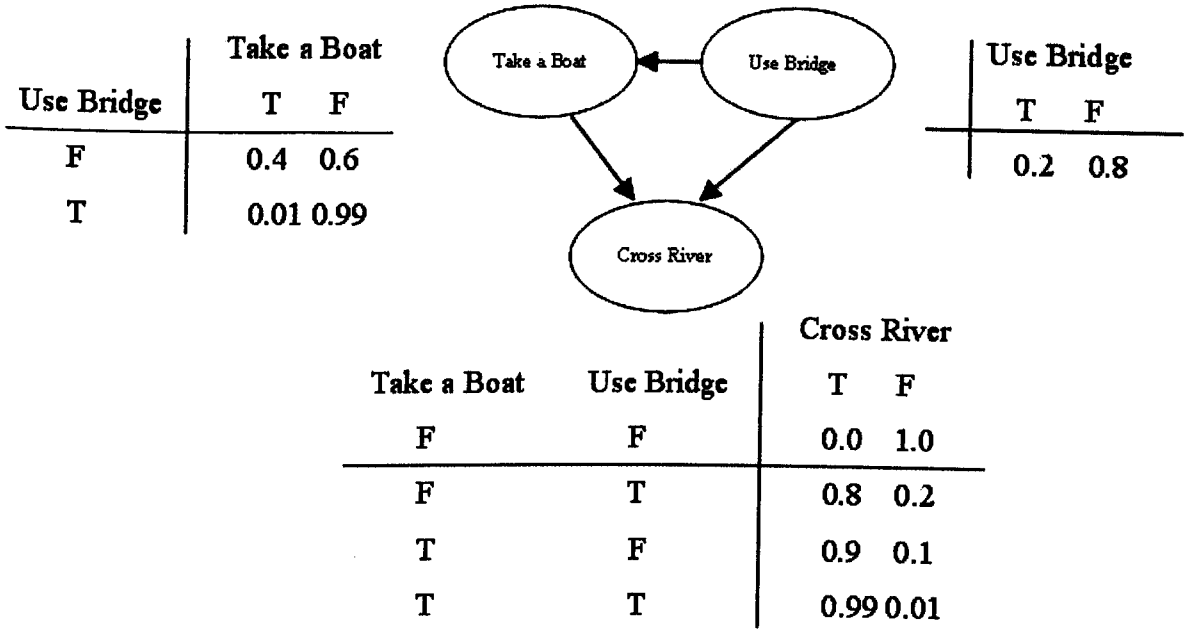
Figure 18: 8 nearest neighbours calculated by Euclidean Distance [72]

Since the majority (seven out of eight) of these eight nearest neighbours are pink objects, the query object is therefore classified as a pink object.

However, for classifying a pattern its distance to all the patterns in the learning set has to be calculated, as in the example in figure 18. If the patterns or data dimensions are huge, one significant disadvantage of this method is its large computing power requirement. Also extra effort needs to be paid to determine the value of parameter k (number of nearest neighbours). Furthermore, in distance based learning it is not clear which type of distance to use and which attribute to use to produce the best results. For instance, whether all attributes need to be used or certain attributes only.

2.2.2.2.4 Bayesian Networks

Bayesian networks use a simple, graphical notation for conditional independence assertions and therefore for compact specification of full joint distributions. Normally, a Bayesian network has a set of nodes, one variable per node, a directed and acyclic graph of direct influences and a conditional distribution for each node given its parents. For example, in figure 19, assume that there are two ways to cross the river: either taking a boat or using a bridge if there is one. Also, assume that there is a bridge has a direct effect on the use of the taking a boat (namely that when there is a bridge, the boat is not being used). Then the situation can be modelled with the adjacent Bayesian network. All three variables have two possible values, T (for true) and F (for false).



Use Bridge	Take a Boat T	Take a Boat F
F	0.4	0.6
T	0.01	0.99

Use Bridge T	Use Bridge F
0.2	0.8

Take a Boat	Use Bridge	Cross River T	Cross River F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

Figure 19: A simple Bayesian network

This Bayesian model can answer questions like "What is the probability that there is a bridge, if the person has crossed the river?" by using the conditional probability formula and variables [73].

One advantage of Bayesian networks is that it is easier for a human to understand direct dependencies and local distributions than complete joint distribution. However, deciding conditional independence is hard in noncausal directions and exact inference in large networks takes a very long time [79]. In neural networks, it is not necessary to determine which factors or variables are dependant.

2.2.2.2.5 Boosting

Boosting is a machine learning meta-algorithm to perform supervised learning. It is based on the question posed by Kearns [74]: "*Can a set of weak learners create a single strong learner?*" A weak learner is a classifier which is slightly related with the true classification. In contrast, a strong learner is a classifier that is well related with the true classification.

Normally boosting occurs in iterations, it continually adds weak learners to a final strong learner. In each iteration, a weak learner learns the training data with respect to a distribution. Then the weak learner is added to the final strong learner. This is done by weighting the weak learner related to its accuracy. After the weak learner is added to the final strong learner, the data is weighted once again. Misclassified examples gain weight and on the other hand, the correctly classified examples lose weight. Thus, future weak learners will focus more on the examples that previous weak learners misclassified.

However, a disadvantage of boosting is the overweighting of errors, which is sometimes a problem in real world classification problems that contain noise [75].

2.2.2.2.6 Minimum Message Length

Minimum Message Length (MML) is a method of Bayesian model comparison and it gives every model a score. MML [76] is a technique based on information theory for discovering and confirming patterns in data. Essentially, it considers a pattern to have occurred in data if the assumption of the pattern enables the data to be encoded more concisely.

It is a formal information theory restatement of Occam's Razor [77]: "Even when models are not equal in accuracy, the one generating the shortest overall message is more likely to be correct". It is to say that "When deciding between two models which make equivalent predictions, choose the simpler one [78]". There is a reasonable question, "Why not just pick the hypothesis with the highest Bayesian posterior probability?" instead of using MML. This is because in Bayes's theorem, the probability of a hypothesis *(H)* given evidence *(E)* is proportional to $P(E \mid H)\, P(H)$, which is just $P(H \wedge E)$. But MML produces the model with the highest such probability and also generates the shortest description of the data.

However, like Bayesian networks, it also takes very long time to get accurate inference in huge networks [79].

2.2.2.2.7 Quadratic Classifier

A quadratic classifier finds a quadratic discrimination function, like a circle or a parabola in feature space. An example is the normal densities based quadratic classifier. It is almost identical to the linear classifier based on normal densities, but it calculates covariance matrices for each individual class.

The quadratic classifier is used in machine learning to separate measurements of two or more classes of objects or events by a quadric surface. It is almost identical to the linear classifier based on normal densities, but a more general version because it calculates covariance matrices for each individual class.

However, quadratic classifiers may have a severe disadvantage in that they tend to have significantly larger biases than linear classifiers particularly when the number of design samples are relatively small [80].

2.3 NEURAL NETWORKS ARCHITECTURE

2.3.1 Fuzzy Neural Networks

Fuzzy logic is a type of multi-valued logic in which there are many possible values for a statement rather than truth or false. In other words, fuzzy logic can express attributes in different levels (0.9, 0.7, 0.5, 0.2...), rather than just 0 or 1 or different shades of greys, rather than just black and white [81].

A fuzzy neural network embeds a fuzzy system in the body of an artificial neural network. It uses a learning algorithm derived from neural network theory to determine the fuzzy sets or rules by processing data samples [82]. To integrate an artificial neural network with a fuzzy system, the selection of learning algorithm and fuzzy rules totally depends on the data or the application itself.

According to [83], three categories can be used to classify different types of fuzzy neural networks: Cooperative Neuro-Fuzzy System, Concurrent Neuro-Fuzzy System and Hybrid Neuro-Fuzzy System [83].

In a cooperative model, the neural network algorithm will learn from the training data and then determine the parameters or rules for the fuzzy system, whereas in a concurrent model, the neural network will assist the fuzzy system continuously to determine the required parameters. In other words, the neural network pre-processes the inputs of a fuzzy system in a concurrent model.

In a hybrid neuro-fuzzy system, a neural network is used to learn some parameters of the fuzzy system through the patterns processing. The majority of neuro-fuzzy systems used in recent research refer to the hybrid neuro-fuzzy system. There are many popular hybrid fuzzy neural network models, for example, ANFIS [84], FuNe [85], GARIC [86], FALCON [87].

The combination of fuzzy logic system with artificial neural network can help solve the inherited limitations of each isolated paradigm and therefore produce a more powerful and efficient model. However, this thesis will not further investigate fuzzy-neuro systems as it will mainly focus on developing efficient learning algorithms, rather than control systems.

2.3.2 Feed-forward Neural Network

Feed-forward artificial neural networks (as in figure 20) allow signals to travel in one way only; from input to output. No feedback is allowed. The output of any layer does not affect that same layer. Feed-forward neural networks are straight forward networks that connect inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down network.

A Feed-forward Neural Network

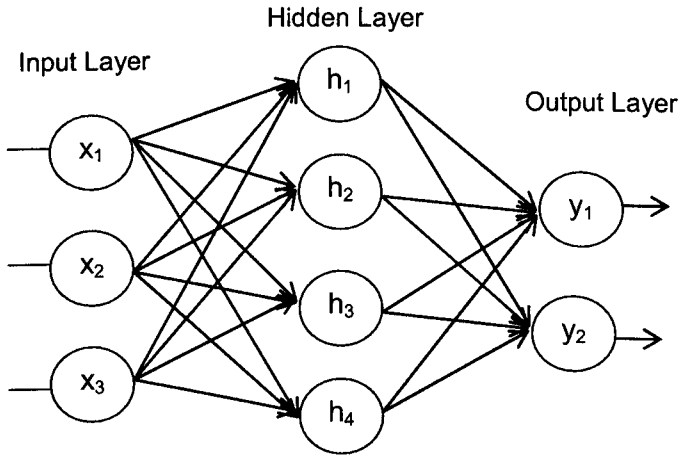


Figure 20: A feed-forward network structure

2.3.2.1 *Single-Layer Perceptron*

The simplest perceptron is the single layer perceptron, developed by Rosenblatt in 1958, which is capable of classifying linearly separable patterns only [10]. A single-layer perceptron network consists of one or more artificial neurons in parallel. It is a single layer network with continuous perceptrons and each neuron possesses a continuous activation function. This network learns by adopting the delta learning rule separately to each neuron. However, a single layer perceptron can not deal with linear inseparable problems [10], that is why a multi-layer perceptron is introduced.

2.3.2.2 *Multi-Layer Perceptron*

The Multi-Layer Perceptron (MLP) is the most widely used type of neural network. It is both simple and based on solid mathematical grounds. There is an input layer, with a number of neurons equal to the number of variables of the problem; and, an output layer, where the perceptron response is made available, with a number of neurons equal to the desired number of quantities computed from the inputs. The layers between input and output layers are "hidden" layers. A network can contain more than one hidden layer. Theoretically the perceptron can not perform non-linear tasks without a hidden layer.

2.3.2.3 *Back Propagation*

In order to enable a neural network to learn tasks, the weights of each neuron are adjusted so that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (*de/dw* the error derivative of the weights ('*e*' for error and '*w*' for weight)). In other words, it must calculate how the error changes as each weight is increased or decreased slightly.

Supposing all the neurons in the network are linear, the back-propagation algorithm will be easier to understand. In back-propagation, it computes each *EW* by firstly computing the *EA*, which is the rate at which the error changes as the activity level of a neuron is changed. For

each output, *EA* is simply the difference between the actual and the desired output. In order to compute the *EA* for a hidden unit, firstly all the weights between that hidden neuron and the output neurons to which it is connected need to be identified. Then the weights are multiplied by the *EA*s of those output neurons and add the products. The sum is the *EA* for the chosen hidden neuron. After calculating all the *EA*s in the hidden layer just before the output layer, it is able to compute the *EA*s for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. That is where the term back propagation comes from. Once the *EA* has been computed for a neuron, it is straight forward to compute the *EW* for each incoming connection of the neuron. The *EW* is the product of the *EA* and the activity through the incoming connection.

The above description can be summarised as the following back propagation rule:

1. Randomly initialise the weights in the network
2. Repeat

   For each pattern in the training set do

   1) Calculate the actual output (A) from each output neuron; forward pass
   2) Get the desired output (D) for each pattern
   3) Calculate error (D - A) at each output neuron
   4) Compute Δw for all weights from hidden layer to output layer ; backward pass
   5) Compute Δw for all weights from input layer to hidden layer ; backward pass
   6) Update the weights in the network

3. until all examples classified correctly or stopping criterion satisfied

The difference between feed-forward and back propagation: when considering a network with a single input x and network function F, the derivative of F(x) can be computed in feed-forward: the input x is presented to the network. The primitive functions and their derivatives are calculated at each node. Then the derivatives are stored.

However in *back propagation*: constant 1 is applied into the output neuron and the network processes backwards. Information came to a node is added and the result is then multiplied by the value stored in the left part of the neuron. The result is transmitted to the left of the neuron.

The result collected at the input neuron is the derivative of the network function with respect to x.

Over the traditional methods of error minimisation, the back propagation algorithm reduces the cost of computing derivatives by a factor of the number of derivatives to be calculated. Also it allows higher degrees of nonlinearity and precision to be applied to problems. It is an effective learning rule but it also has disadvantages, i.e. poor scaling properties [88], slow rate of convergence commonly for most of the multi-layer networks, requiring very long training time.

2.3.2.4 Adaptive Linear Neuron (ADALINE)

ADALINE (Adaptive Linear Neuron or later Adaptive Linear Element) is a single layer neural network and an important generalisation of the perceptron training algorithm. It was presented by Widrow and Hoff [89] as the "least mean square" (LMS) learning procedure, also known as the delta rule which accepts multiple inputs and generates a single output. The delta learning rule adopted in ADALINE is a data-adaptive technique for deriving a least squares error solution. The main difference between the delta rule and the perceptron training rule is the way that the output of the system is used in the learning rule. In the perceptron learning rule, it uses the output of the threshold function (-1 or 1) for learning. Whereas the delta rule uses net output without further mapping into output values -1 or +1 is used.

2.3.2.5 Radial Basis Function (RBF) Network

A radial basis function network (figure 21) is an artificial neural network which uses radial basis functions as activation functions. Radial functions are simply a class of functions. They could be used in both linear and nonlinear models, single layer and multi-layer networks. However, since Broomhead and Lowe's 1988 seminal paper [90], radial basis function networks (RBF networks) have traditionally been associated with radial functions in a multi layer network such as shown in the figure below.

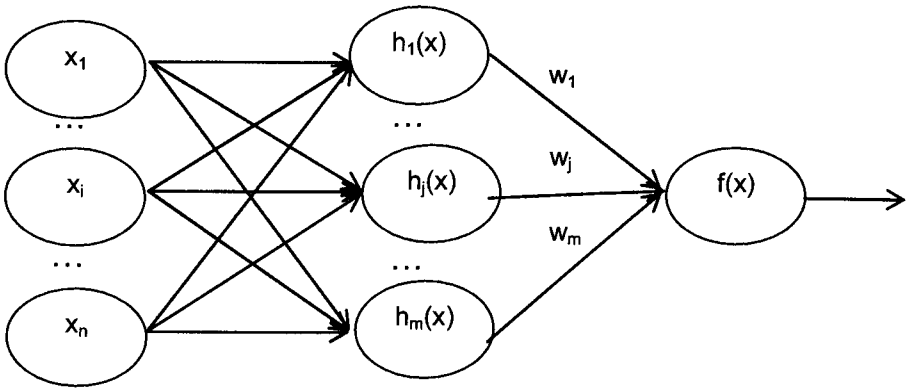


Figure 21: The traditional radial basis function network.

Each of n components of the input vector x feeds forward to m basis functions whose outputs are linearly combined with weights $\{w_j\}^m_j$ into the network output $f(x)$.

The most commonly used formula for any radial basis function (RBF) is

$$h(x) = \Phi((x-c)^T R^{-1}(x-c))$$

where Φ is the function used (Gaussian, multi-quadric), c is the centre and R is the metric. The term $((x-c)^T R^{-1}(x-c))$ is the distance between the input x and the centre c in the metric defined by R. There are several common types of functions used, for example, the Gaussian, $\Phi(z) = e^{-2}$, the multi-quadric, $\Phi(z) = (1+z)^{1/2}$, the inverse multi-quadric, $\Phi(z) = (1+z)^{-1/2}$ and the Cauchy $\Phi(z) = (1+z)^{-1}$.

A further simplification is a 1-dimensional input space in which case

$$h(x) = \Phi\left(\frac{(x-c)^2}{r^2}\right)$$

The training of RBF networks is relatively fast and they can deal with universal approximation with non-restrictive assumptions [91]. However, one of disadvantages of RBF is it is too expensive because of using a series of computationally expensive functions for a single model [92]. RBF networks also require good coverage of the input space by radial basis functions. The network includes all the input variables and as a result, representational

resources may be wasted on areas of the input space that are irrelevant to the learning task [93].

2.3.3 Recurrent Neural Network

A recurrent neural network [94, 95] is a neural network where the connections between the neurons form a directed cycle. Recurrent neural networks are approached differently from feed forward neural networks, both when analysing their behaviour and training them. Usually, dynamical systems theory is used to model and analyse these type of networks.

2.3.3.1 Simple Recurrent Network

Simple recurrent networks (SRNs) are similar to traditional feed-forward networks in the sense that information propagates all the way through the network during the forward pass on each tick. In simple recurrent networks, groups are updated in the order in which they appear in the network's group array. A group will be updated by computing its inputs and outputs. This differs from fully recurrent networks where all groups update their inputs and then the groups compute their outputs. Thus the continuous networks can be considered to have synchronous update and standard or simple recurrent networks have sequential update.

A SRN is just a feed-forward network with one or more ELMAN [96] type groups. As shown in figure 22, neurons of the input layer *I,* the recurrent layer *R* and the output layer *O* are fully connected as in the feed forward multilayer perceptron (MLP). Time delay connections send the current activities of recurrent neurons $R^{(t)}$ to the context layer so that $C^{(t)} = R^{(t-1)}$. Hence, every recurrent neuron is fed by the activities of all recurrent neurons from the previous time step through recurrent weights W^{RC}. Recurrent neurons' activities from the previous time step can be viewed as an extension of input to the recurrent layer. These activities represent the memory of the network, since they hold contextual information from previous time steps.

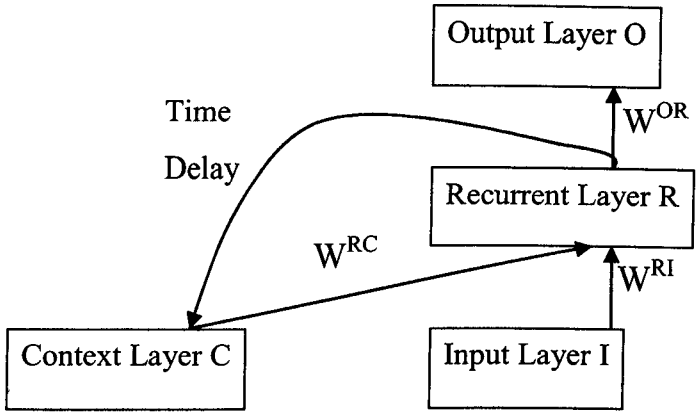


Figure 22: The simplified representation of Elman's [96] SRN

Williams and Zipser [97] have developed the real-time recurrent learning (RTRL) approach, which enjoys the generality of the BPTT [98, 99] approach which can be derived by unfolding the temporal operation of a network into a multilayer feed forward network that grows by one layer on each time step, while not suffering from its growing memory requirement in arbitrarily long training sequences. Although the RTRL algorithm has great power and generality, it has the disadvantage of being computationally very expensive. In spite of several modifications of RTRL [100 and 101] to reduce the computational expense, it is still complicated when dealing with complex problems.

2.3.3.2 Hopfield Network

In the beginning of the 1980s Hopfield [102] showed that models of physical systems could be used to solve computational problems. These type of systems could be implemented in hardware by combining standard components such as capacitors and resistors.

The importance of the different Hopfield networks in practical applications is limited due to the theoretical limitations of the network structure. However, in certain situations, they may form interesting models. Hopfield networks are typically used for classification problems with binary pattern vectors.

The Hopfield network is created by supplying input data vectors with corresponding different classes. When a distorted pattern is presented to the network, it is associated with another

pattern. If the network works properly, this associated pattern is one of the class patterns. In some cases (when the different class patterns are correlated), false minima can appear. This means that some patterns are associated with patterns that are not among the pattern vectors.

However, the main disadvantage of the Hopfield network is that operates satisfactorily it needs a large number of nodes [103] which can accordingly increase the physical system requirements.

2.3.3.3 Echo State Network

The echo state network (ESN) is a recurrent neural network with a sparsely connected hidden layer (typically with 1% connectivity). The connectivity and weights of hidden neurons are randomly initialised and are normalised. The weights of output neurons can be learned so that the network can produce a specific temporal pattern.

Although this network is non-linear; the only parameters are the weights of the output layer. The error function is thus quadratic with respect to the parameter vector and can be differentiated easily to a linear system.

Echo state networks (ESN) provide architecture and supervised learning principle for recurrent neural networks (RNNs). The main aim is to drive a random, large, fixed recurrent neural network with the input signal, thereby inducing in each neuron within this network a nonlinear response signal, and also to combine a desired output signal by a trainable linear combination of all of these response signals.

However, ESNs have a number of disadvantages, higher computational cost (quadratic in filter length) and space complexity: they are very difficult to implement; and, can fall prey to instability [104].

2.4 FUNCTION MODIFIABLE LEARNING METHODS

2.4.1 Introduction

The classical neuron computes the weighted sum of its inputs and feeds it into a nonlinear function called an activation function [105]. The performance of a neural network built with neurons depends on the chosen activation functions.

Recently, more and more computational neuroscience research has started to suggest that neuromodulators play a role in learning by modifying the neuron's activation function [106, 107]. This research introduced new methods which involve the adaptation of the gain and the slope of the sigmoidal activation function during the learning [108] and using adaptive polynomial functions [109]. In this section, several recently proposed activation function modifiable learning methods are introduced.

2.4.2 Adaptive Polynomial Activation Functions

Uncini et al [109] introduced an adaptive polynomial neural network (APNN) based on adaptive polynomial activation functions.

The authors employed a polynomial function neural network which allows increasing the neuron complexity and reducing the size of the network. Experimental results showed similar generalisation ability to the conventional Volterra's polynomial classifiers or filters. As some of the parameters are related to the polynomial activation functions, the APNN is therefore able to reduce the structural complexity of the network. However, although APNN is efficient for various applications, APNN could have problems because of local minima and in the case of high-degree polynomials could cause numerical instability to arise, due to the presence of high narrow peaks in the polynomial functions.

2.4.3 Adaptive Spline Activation Function Neural Networks

Pizza et al. [110] use an adaptive spline approach to function modification, and elsewhere both Fiori [111] and Piazza [112] use a Digital Look-Up Table (LUT) for the activation function.

Vecci et al [110] described a cubic spline activation function by n control points (x_i, y_i) on R, where i = 1...n. These n points define n-1 intervals on the x-axis. Therefore each of these intervals is defined by function $f_i(x)$:

$$f_i(x) = f_i(x_i) + a_i(x - x_i) + b_i(x - x_i)^2 + c_i(x - x_i)^3$$

where a_i, b_i *and* c_i are constants on R. Demanding equality of the function value, and the first and second derivative at the interval borders the constants can be determined for each interval yielding a continuous and differentiable function composed of a number of cubic splines.

The maximum number of control points n_c is set prior to the evolutionary run. And the x-, y-range of the cubic spline activation function is fixed to specific intervals $[x_{min}, x_{max}]$ (sensitivity interval) and $[a_{min}, a_{max}]$ (activation interval) respectively.

Uncini et al [113] extended the above adaptive activation functions to a complex-valued neural network by varying the control points of a pair of Catmull–Rom cubic splines, which are used as an adaptable activation function.

This method is a complex extension of the adaptive spline neural network as shown in figure 23. The control points are points that define a spline, they are equally spaced and fixed on the x-axis.

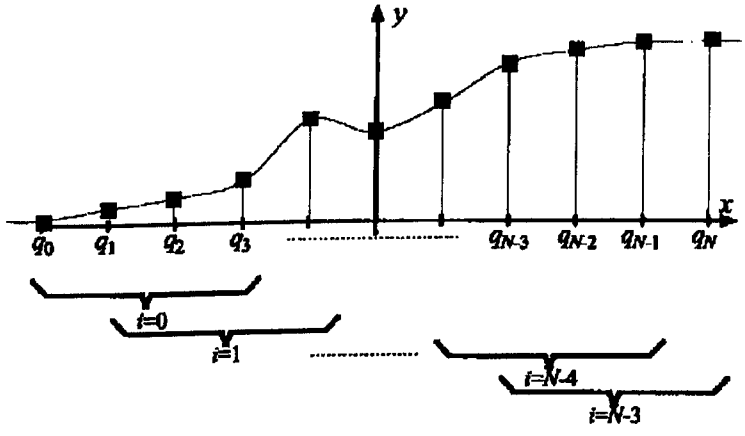


Figure 23: Spline activation function with its uniformly spaced control points [48]

The Catmull-Rom spline, a member of the family of cubic convolution filters, is the unofficial standard for high-quality rendering in visualization. However, Catmull-Rom spline filters are insufficient for high-quality rendering of refractive effects since here the error is multiplied by the length of the redirected ray before it hits an opaque surface.

By comparing with an adaptive linear combiner and sigmoidal MLP on QAM equalization problem, this method showed the ability to improve the generalisation capabilities using few training samples in terms of conquering the long adaptation time and high number of interconnections.

However, in the cubic analytic spline approach, a cubic level of complexity is assumed and a good suboptimal solution to the cubic spline curve fitting problem is applied. If the required function is linear, or a step, ramp or pulse function, splines are inappropriate.

In addition, LUT requires a bounded input address and a suitable linear transformation (scaling and offset adding) has to be performed on the output of the linear combiner in order to obtain the best LUT address.

Recently [174, 175 and 176], a class of activation function has been introduced based on bi-dimensional (2D) spline, which aims to obtain bounded and locally analytic functions. It also uses the generalised splitting activation function. Its architecture is based on a multi-dimensional adaptive cubic spline basis activation function that collects information from the previous network layer in an aggregate form. In other words, each activation function represents a spline function of a subset of previous layer outputs so the number of network connections in terms of structural complexity can be at a low level.

2.4.4 Adaptive Optical Radial Basis Function Neural Networks

In [114], an adaptive optical radial basis function neural network classifier is introduced. It is a spatially multiplexed system incorporating on-line adaptation of weights and basis function widths to provide robustness to optical system imperfections and system noise. The optical system computes the Euclidean distances between a 100-dimensional input vector and 198

stored reference patterns in parallel using dual vector-matrix multipliers and a contrast-reversing spatial light modulator. Software is used to emulate an analogue electronic chip that performs the on-line learning of the weights and basis function widths. An experimental recognition rate of 92.7% correct out of 300 testing samples is achieved with the adaptive training, as compared to 31.0% correct for non-adaptive training. They compare the experimental results with a detailed computer model of the system in order to analyze the influence of various noise sources on the system performance.

2.4.5 An Adaptive Activation Function for Classification of ECG Arrhythmias

Ozbay and Karlik carried out [115] a comparative study of the classification accuracy of ECG signals using a multi-layered perceptron (MLP) with back-propagation training algorithm, and a neural network with adaptive activation function (AAFNN) for classification of ECG arrhythmias. The ECG signals are taken from the MIT-BIH ECG database, which are used to classify ten different arrhythmias for training. These are normal sinus rhythm, sinus bradycardia, ventricular tachycardia, sinus arrhythmia, atria premature contraction, paced beat, right bundle branch block, left bundle branch block, atria fibrillation and atria flutter. For testing, the proposed structures were trained by back-propagation algorithm. Both of them tested using experimental ECG records of 10 patients (7 male and 3 female, average age is 33.8±16.4). The results show that the neural network with the adaptive activation function is more suitable for biomedical data like the ECG in the classification problems. Also, the training speed is much faster than a neural network with fixed sigmoid activation function.

2.5 INTELLIGENT DATA ANALYSIS

2.5.1 Introduction

Data analysis is the process of looking at and summarising data with the intent to extract useful information and develop conclusions. Data analysis is closely related to data mining, but data mining tends to focus on larger data sets, with less emphasis on making inference, and often uses data that was originally collected for a different purpose [116].

Applying neural networks to model complex data for data analysis has been developed for two decades; a successful example by Roadknight et al [117] is to use Artificial Neural Networks (ANN's) to model the interactions that occur between ozone pollution, climatic conditions, and the sensitivity of crops and other plants to ozone.

Intelligent data analysis can cover a variety of disciplines. Techniques applied in intelligent data analysis can cross many areas such as: data visualization, data mining, knowledge acquisition from data, knowledge discovery, machine learning, neural networks, and mathematical or statistical methods. Therefore, this section only looks at some aspects of intelligent data analysis related to this project, e.g. neural networks in data analysis and some mathematical methods which will assist neural network in analysing the data.

2.5.2 Important Information Revealed from Learned Weights

As described above, neural networks learn from experience and are useful in detecting unknown relationships between a set of input data and an outcome. It detects patterns in data, generalises relationships found in the data, and predicts outcomes. Training continues until a neural network produces outcome values that match the known outcome values within a specified accuracy level, or until it satisfies some other stopping criteria. The weights assigned to each of the inputs are obtained during a training process in which outputs generated by the nets are compared with target outputs. The answers people want the network to produce are compared with generated outputs, and the deviation between them is used as feedback to adjust weights. The process of readjusting weights is important to increasing a model's accuracy.

Therefore, when the specified accuracy level is achieved, training stops. The learned weights yield very important information about the dataset. For each output class, the stronger the weight is, the more this output will be dependant on this corresponding input.

2.5.3 Statistical Mathematical Method

2.5.3.1 Simple Moving Average

A simple moving average is formed by computing the average (mean) of a set of points over a specified window. The calculation is repeated for each point. The averages are then joined to form a smooth curve. The formula for a simple moving average is:

$$\overline{y_t} = (y_t + y_{t-1} + \ldots + y_{t-n-1}) / n$$

Where y is the variable, t is the current point, and n is the window width for the average. For instance, if there are 100 points, the window width, n, is 5, as the calculation continues, the newest point is added and the oldest point is subtracted.

2.5.3.2 Least Square Polynomial Smoothing

The least squares curve fitting technique is the simplest and most commonly applied form of linear regression. Its theory is to find the best fitting curve to a given set of points by minimizing the sum of the squares of the residuals (offsets) of the points from the curve. By implementing the least-squares linear regression analysis, it is easy to fit any polynomial of m degree $Y = a_0 + a_1x + \ldots + a_mx^m$ to experimental data $(x_1, y_1), (x_2, y_2)\ldots, (x_n, y_n)$, (provided that $n \geq m+1$) so that the sum of squared residuals S is minimized:

$$S = \sum_{i+1}^{n} [y_i - \hat{y_i}]^2 = \sum_{i+1}^{n} [y_i - (a_0 + a_1x_i + \ldots + a_mx_i^m)]^2$$

After getting the partial derivatives of S with respect to a_0, a_1, .., a_m and equating these derivatives to zero, the following system of m-equations and m-unknowns $(a_0, a_1, .., a_m)$ is defined: $s_0a_0 + s_1a_1 + \ldots + s_ma_m = t_0, \ldots\ldots\ldots, s_ma_0 + s_{m+1}a_1 + \ldots + s_{2m}a_m = t_m$

where:

$$s_k = \sum_{i+1}^{n} x_i^k, \quad t_k = \sum_{i+1}^{n} y_i x_i^k$$

Thus the set of coefficients $(a_0, a_1, \ldots, a_m)$ which gives the resulting smoothed curve can be calculated.

2.5.4 Other Data Analysis Methods

The plausible reasoning is rather open-ended and it is the central to data analysis [118]. Data analysis is a highly complex activity. It requires repetitive analysis of the data. Plus the size, format or the data may vary. Traditionally people use statistical and mathematical methods to look into their data.

Bayesian statistical analysis is one approach in the data analysis area. It involves a field that combines elements of decision modelling and statistical analysis. Applications of Bayesian data analysis appear in different fields, including business, computer science, economic, educational research, environmental science, epidemiology, genetics, geography, imaging, law, medicine, political science, psychometrics, public policy [119]. However, there are also some very significant disadvantages of Bayesian approach. First of all, the information theoretically infeasible in which means to specify a prior is extremely difficult. A real number of every setting of the world model parameters must be specified. Other disadvantages include computational infeasible and unautomatic.

2.6 NEURAL NETWORK GENERALISATION

A supervised neural network learns to approximate the actual outputs to the target outputs, from the given training set. This is helpful by itself, but a wider purpose of using a neural network is to generalise to predict the actual outputs to target values for inputs that are not in the training set.

Given an x value that has not been seen, the trained network can predict what the most likely y value will be. The ability to correctly predict the output for an input the network has not seen is called generalisation.

There are three factors that are typically considered important to achieve good generalisation in terms of acquiring good prior knowledge, proved by Wolpert [120]. They are just important factors, and not sufficient by themselves in order to get the best generalisation result.

The first factor is that the inputs have to contain sufficient information which somehow relates to the target output. This means there must exist a mathematical function which can link the inputs to the expected outputs with an expected level of accuracy. This is why a poor generalisation sometimes suggests insufficient data collection. The neural network is able to find and extract the answer only if enough information is present for an answer to be defined. A very simple example can explain this, e.g. if the experiment only has last year's global economic data, it is unlikely any neural network can predict the economic trend for the next 5 years. Similarly, if we only have data on blood attributes and eating habit attributes, it is still very hard to predict whether a person has diabetes or not, more attributes like medication history, sugar consumption need to be collected.

The second factor Wolpert proved is that the function the network is trying to learn is in some sense, smooth. In other words, a small change in the inputs should, most of the time, produce a small change in the outputs. However, the work in this thesis challenges this requirement as the shape of activation functions do not have to be smooth, they can be initialised to any random shape and they are adaptive, working with the adaptive weights together to reduce errors produced from the output neurons. More detail will be introduced in the next chapter.

The third necessary factor for good generalisation is that the training cases be sufficiently large and be a representative subset of the set of all cases that the network wants to generalise to.

In this thesis, for all dataset, generalisation is performed by multiple rounds of cross-validation using different partitions but same proportion for each class, and the validation results are averaged over the rounds.

2.7 CONCLUSION

This chapter firstly looks at two major categories of machine learning algorithms, namely symbolic and non-symbolic learning, and typical methodologies for each category. As one of the most popular models for non-symbolic learning, the Artificial Neural Networks (ANN)'

learning paradigms, learning algorithms, network architectures and their advantages and disadvantages are then discussed.

One of the most innovative ideas of this thesis is the adaptive activation function in a single layer network structure, therefore this chapter also looks at related works which involved function modifiable learning methods and their limitations.

To provide a context for intelligent data analysis which is expected to be an additional benefit from the new approach of this thesis, this chapter then looks at traditional data analysis methods using neural networks, as well as other mathematical or statistical methods and their limitations.

At the end of this chapter, neural network generalisation is discussed and introduced, in order to set up the standards in experimental practices in this thesis to evaluate the new method.

In summary, this chapter provides a platform and context for the new approach which is to be introduced to overcome limitations of and enhance advantages over the traditional methods in the next chapter.

CHAPTER 3 AN ADAPTIVE FUNCTION NEURAL NETWORK (ADFUNN)

3.1. INTRODUCTION

The motivation of the proposed modal learning method, ADFUNN (An Adaptive Function Neural Network) in the thesis is inspired from recent neuroscience, which suggests that neuromodulators play a role in learning by modifying the neuron's activation function [106, 107].

Conventionally, the artificial neural network system tries to simulate the structure and functional aspects of a biological neural system. In biological neural system, inputs to neurons are weighted by the strength of the synapse that the signal travels through. It is biologically plausible that the total input to the neuron is the sum of all such synaptic weighted inputs and adapting a weight is a reasonable model for synaptic modification. In contrast, the widely accepted assumption of a fixed shape output activation function is for computational rather than biological reasons. A fixed analytical function activation function can facilitate mathematical analysis to a higher degree than an empirical one. However, there are some computational benefits to trainable and modifiable activation functions, and they are quite possibly biologically plausible as well.

The norm in neural computing is to learn to classify different classes from input patterns by adapting the strength of connections between neurons [121]. However, the great success ANNs has brought to the world including the high speed neuron computing feasible on the desktop and more recently in the palm of the hand [122]. However, it has resulted in little attention being paid currently to the range of possibilities of adaptation within the individual neurons.

There might be good computational and biological reasons for examining and discovering the internal learning neural mechanisms. It is interesting that recent research in neurosciences also suggests the neuromodulators play a role in learning by modifying the neuron's activation function [106, 107]. In fact it is very surprising real neurons are essentially fixed entities with no adaptive aspect, apart from their synapses changes, since such a restriction leads to non-linear responses typically requiring many neurons.

Using MLPs is very effective with an appropriate number of hidden neurons but since the number of hidden neurons depends on the activation function it is, therefore, not efficient if the activation function itself is not optimal. Typically training is very slow on linearly inseparable data which always involves hidden nodes. In some cases adapting a slope-related parameter of the activation function may be helpful, but not if the analytic shape of the function is unsuited to the problem, in which case many hidden nodes may be required. In contrast with an adaptive function approach it should be possible to learn linear inseparable problems fast, even without hidden nodes.

In contrast with the conventional artificial neural network learning which is typically accomplished via adaptation between neurons, this chapter describes a modal learning method in which adaptation is simultaneously between and within neurons.

3.2. PIECEWISE LINEAR ACTIVATION FUNCTION

In mathematics, $f: \Omega \rightarrow V$ is a piecewise linear function (figure 24) if any function with the property Ω can be decomposed into finitely many convex polytopes, such that f is equal to a linear function on each of these polytopes" [123].

$$f(x) = a_i x + b_i$$

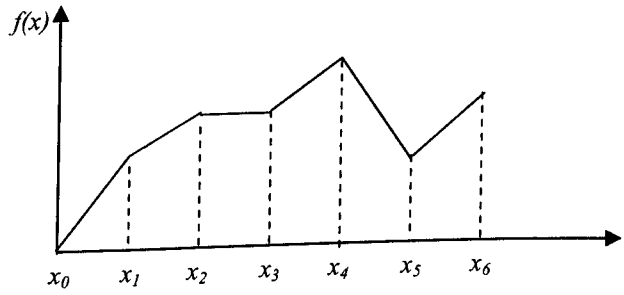


Figure 24: A Piecewise Linear Function

Functions like spline, polynomial and piecewise linear can all be chosen as the adaptive function in ADFUNN. The reason to choose piecewise linear function is simple, because it has the maximum plasticity and represent any shape the data may produce, whereas a spline function or a polynomial function cannot. As in the example shape in figure 25, if data produces a learned function shape like this, the polynomial function and spline function can only produce the shape on both sides, but they cannot represent any sharp steps as in the central part. However, the piecewise function can better represent all kinds of different shapes. The only constraint is that the maximum slope is A/interval (A is input data and interval is the space between two x points). Therefore, with sufficient points sharp edges (discontinuities) can be represented.

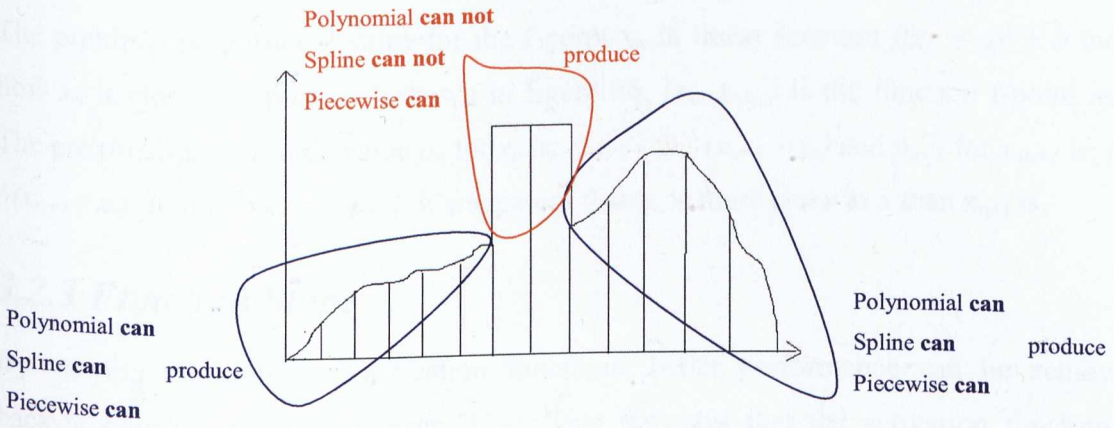


Figure 25: To represent learned function

3.2.1 The Number of F-Points

In this thesis, f-point refers to the x point interval in the piecewise linear function. The number of f-points depends on the input data range which applied to the neural network, the input data precision and the weights initialisation. For instance, if the input data has a known range of *[0, 10]*. Weights are initialised randomly between *[-1, 1]*, then the sum of weighted input $\sum aw_j$ will have a range of *[-10, 10]* which marks the function's X-axis range. The data precision is the minimum difference between data. If, for example, the dataset has a precision of *0.1*, then the whole range can be coded with a resolution of *0.1* as the f-point interval. Therefore, $(10-(-10))/0.1 + 1 = 201$ points are sufficient to encode the data precisely.

3.2.2 Proximal-Proportional Basis

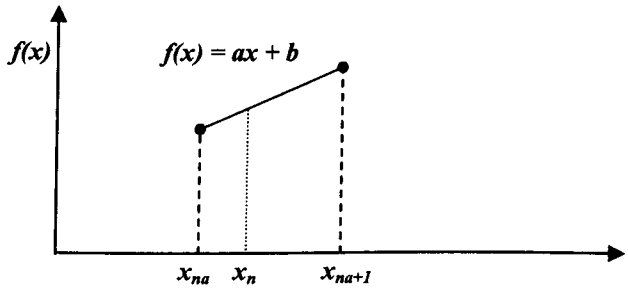


Figure 26: Proximal-Proportional Basis

The proximal-proportional value for the f-point x_n in linear function $f(x) = ax + b$ indicates how x_n is close to x point. As shown in figure 26, $[x_n, x_{n+1}]$ is the function f-point interval. The proximal-proportional value p_n for x_n is: $(x_{n+1} - x) / (x_{n+1} - x_n)$ and p_{n+1} for x_{na+1} is: $(x - x_n) / (x_{n+1} - x_n)$. In this case $p_n > p_{n+1}$, it's apparent that x_n is more close to x than x_{n+1} is.

3.2.3 Function Slope

By varying the slopes of activation functions, better performance can be achieved by back-propagation neural networks [124]. This indicates that the activation function slope plays a role in reducing errors, as well as accelerating the convergence speed, when used alongside weights adaptation.

In ADFUNN, since functions are adapted, it follows that functions slope is variable. Hence by calculus as follows the weight update rule includes the slope.
If:

z : ∑(aw)

f(z): real output

T: target output (constant)

a : input value

e : output error

S: activation function slope

$\because e = T - f(z)$

$\therefore de/dw = d(T - f(z))/dw = -d(f(z))/dw$

$\because dy/dx = dy/du \bullet du/dx$

According to chain rule [125]

$\therefore de/dw = (de/dz) \bullet (dz/dw)$

$\therefore de/dw = (de/dz) \bullet (dz/dw) = (-d(f(z))/dz) \bullet (d(\Sigma aw)/dw) = -S \bullet a$ [F3.1]

$\therefore \Delta w = k \bullet S \bullet a$ [F3.2]

In the standard non adaptive function delta rule, $d(f(z))/dz$ is a constant for the output neurons and hence from the calculus in F3.1 $de/dw = \Delta w = -a$. Since we also want Δw to be proportional to the error e, Δw becomes $de/dw = \Delta w = k \bullet e \bullet a$. In the adaptive function case, $de/dw = \Delta w = -S \bullet a$ and so it follows that Δw becomes

$\Delta w = k \bullet e \bullet a \bullet S$

3.2.4 Function Adaptation

In neural networks, the difference between the expected output and the actual output for each output is its error. Hence the piecewise linear activation function is adapted using a function modifiable learning rule as follows:

$\Delta f(\sum a_i w_{ij}) = l \bullet e_j$

where l is the learning rate for function and e_j is the difference between the expected output and the actual output at output neuron j. Thus the error at each output is directly reduced by modifying the function.

3.3. ADFUNN ARCHITECTURE

By providing a means of solving linearly inseparable problems using a modal learning adaptive function neural network, ADFUNN is investigated and introduced in this section; it is based on a single layer of linear piecewise function neurons, as shown in figure 27.

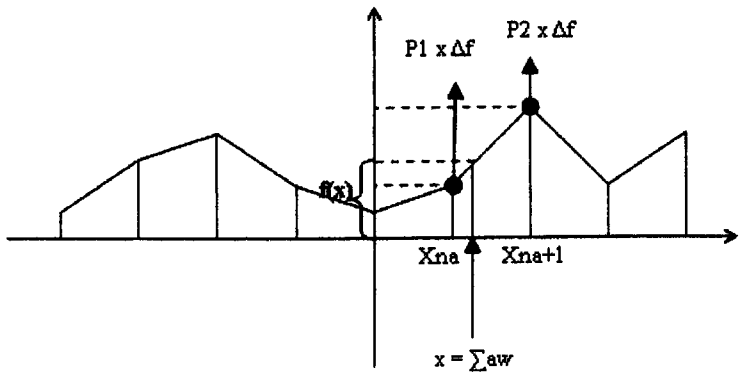


Figure 27: Adapting the linear piecewise neuronal activation function in ADFUNN

By calculating $\sum aw$, the two neighbouring f-points that bound $\sum aw$ can be found on the function. Two proximal f-points are adapted separately, on the proximal-proportional basis. The proximal-proportional value *P1* is $(x_{n+1} - x)/(x_{n+1} - x_n)$ and value *P2* is $(x - x_n)/(x_{n+1} - x_n)$. Thus, the change to each point will be in proportion to its proximity to x. The output error is then obtained and the two proximal f-points can be adapted separately, using a function modifying version of the delta rule to calculate Δf.

Figure 28 is a simple example of ADFUNN with 2 inputs and 1 output neurons.

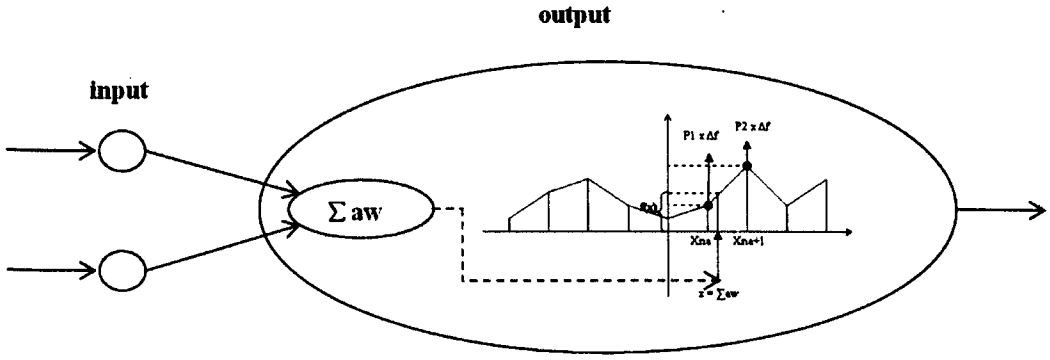


Figure 28: ADFUNN network architecture

ADFUNN is a single layer adaptive function neural network. It is composed by a layer of input neurons which fully forward connected to a layer of output neurons. Like the classic neural network structure, the nodes of the input layer are passive, meaning they do not modify the data. They receive a single value on their input, and duplicate the value to their multiple outputs. In comparison, the nodes of the hidden and output layer are active, they modify the data. The values entering an output node are multiplied by weights, a set of predetermined small numbers (or randomly initialised small numbers) stored in the program. The weighted inputs are then added to produce a single number. Before leaving the node, this number is passed through a piecewise linear activation function. This is continuous piecewise curve that limits the node's output. That is, the input to the piecewise function is a value between $-\infty$ and $+\infty$, while its output are between 0 and 1.

3.4. THE LEARNING METHOD

Artificial neural network learning is typically accomplished via adaptation between neurons. As a modal learning method, learning (adaptation) in ADFUNN is simultaneously taking place between (the synapses/weights between neurons) and within (the adaptation functions) neurons. It is introduced to overcome linear inseparability limitation in a single weight layer supervised network which normally cannot solve such problems with the traditional weights adaptation learning. .

3.4.1 The Delta Learning Rule

The delta learning rule is used by neural networks with supervised learning. It changes weights by multiplying a neuron's input with the difference of its output and the desired output and the network's learning rate. In ADFUNN, the delta learning rule is extended to add the function slope to adapt weights. The function slop is required, because it is changing.

3.4.2 Learning Rates for Weights and Functions

It is clear that every dataset will have its own range. In order to cope with this, two learning constants are introduced, where WL is for weights and FL is for functions. WL depends on input data range and the F point interval. Different WLs are used with different magnitudes of

input vectors to achieve better performance. Whereas FL just depends on the required output range.

3.4.3 Weight Normalisation

During learning, inputs are presented to the network and the sums of weighted input are calculated. As in delta learning rule, the weights are adapted by adding the neuron's input times its error and the network's learning rate. Some of the sums will become very large whereas the others are very small. As in ADFUNN, the sum of weighted input will be passed to the activation function as an input. Since the range of input to the piecewise activation function is fixed, a large weight will be out of the function range and therefore crash the system. Even if the training data is already between the limits 0 and 1 and the data dimension is limited, normalisation is still necessary in ADFUNN and desirable in other systems.

Weight normalisation is performed just like the vector normalisation in mathematics. For example, in ADFUNN, for a set of weights which are from all input neurons to one output neuron, the normalisation is done by dividing the given set of weights by its magnitude. The magnitude of this set of weights a is denoted by $||a||$, where $||a|| = (w_1^2 + w_2^2 + w_3^2)^{1/2}$ in a three-dimensional space. And w_1, w_2, w_3 are weights from the three input to one output. This will result in a weights vector with magnitude 1, but its direction remains the same as in the original vector.

3.4.4 Weight Adaptation

The aim of weight adaptation is to remove noise and find the weights which can produce the right outputs. In ADFUNN, the mode of weights adaptation is carried out using an improved delta learning rule. In the delta learning rule, the change in weight from input u_i to output u_j is normally given by:

$$\Delta w_{ij} = r \bullet a_i \bullet e_j,$$

where r is the learning rate, a_i is the activation of input u_i and e_j is the error from output u_j. As mentioned above calculus, multiplying the slope can achieve an accurate learning. Therefore, in ADFUNN, the delta learning rule is modified as:

$$\Delta w_{ij} = r \bullet a_i \bullet e_j \bullet s_j$$

where s_j is the slope of the line which runs through two neighbouring f points which bounds Σaw in output neuron u_j.

3.4.5 Function Adaptation

As mentioned before, ADFUNN has been invented to overcome the linear inseparability limitation in a single weight layer supervised network. In ADFUNN, piecewise liner function is used as the activation function. Apart from the mode of weights adaptation, the other mode in ADFUNN is function adaptation learning works by finding the two neighbouring f-points that bound Σaw and adapting them separately to reduce errors, on a proximal-proportional basis. The f-point which is nearer to the Σaw is adapted more than the other f-point based on its proportion. If the Σaw is accidentally the exact f-point, then only this f-point is being adapted with the proximal-proportional basis equals to 1.

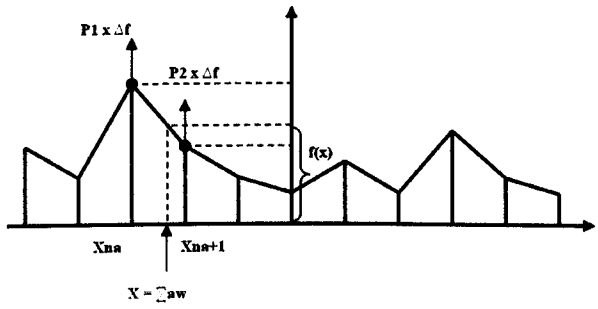


Figure 29: Activation function adaptation

As seen from figure 29, by passing the sum of weighted input as x to the piecewise function, the two neighbouring f points which encloses x are found. $f(x)$ is simply the actual output from the neuron which this function resides in. Δf is the value of function learning rate multiplied by the error. The proportional change of x_{na} $(P1)$ to the point of x_{na+1} is $(x_{na+1} - x)/(x_{na+1} - x_{na})$ and the proportional change of x_{na+1} $(P2)$ to the point of x_{na} is $(x - x_{na})/(x_{na+1} - x_{na})$. Thus x_{na} and x_{na+1} can be adjusted by their corresponding proximal proportional value multiply by Δf upwards or downwards depending on the error.

Therefore, it should come to a complete idea of how adaptation is simultaneously or sequentially between (weights) and within (functions) neurons. Figure 30 shows a whole system diagram of ADFUNN.

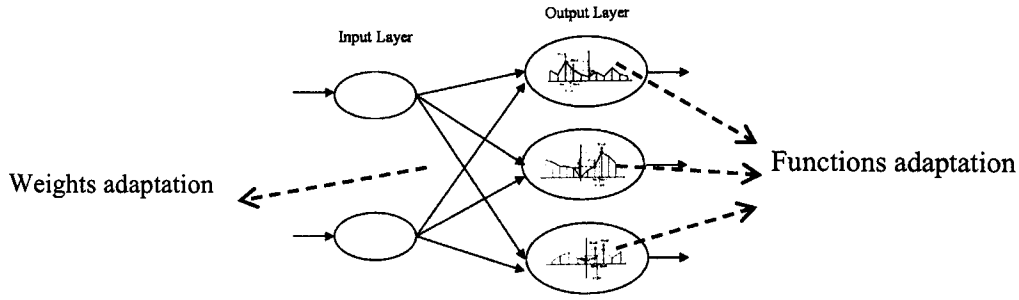


Figure 30: ADFUNN system diagram

3.4.6 ADFUNN General Learning Rule

The weights and activation functions are adapted in parallel, using the following algorithm:

A = input node activation, E = output node error.

WL, FL: learning rates for weights and functions.

Step1: calculate output error, E, for input, A.

Step2: adapt weights to each output neuron:

$$\Delta w = WL \bullet Fslope \bullet A \bullet E$$

$$w' = w + \Delta w$$

weights normalisation

Step3: adapt function for each output neuron:

$$\Delta f(\sum aw) = FL \bullet E$$

$$f'_1 = f_1 + \Delta f \bullet P1,$$

$$f'_2 = f_2 + \Delta f \bullet P2$$

Step4: $f(\sum aw) = f'(\sum aw)$;

$$w = w'.$$

Step5: randomly select a pattern to train

Step6: repeat step 1 to step 5 until the output error tends to a steady state.

3.4.7 Mathematical Principles Employed in the Learning

The delta learning rule introduced by Widrow and Hoff [14, 126] which is also called the Least Mean Square (LMS) method. It is mathematically proved by McClelland and Rumelhart [127] that the delta learning rule provides a very efficient way to modify the initial weights vector in order to minimise the errors. The extension of the delta rule in ADFUNN is to add the function slope in order to achieve a faster and better learning result as mathematically proved in 3.2.3. In ADFUNN the extended delta learning rule combine the above two mathematically proved theorems.

In linear algebra, the process of normalisation is to assign a strictly positive length or size to all vectors in a vector space. This length is calculated by the magnitude of the vector and then each single value of the vector is divided by this length in order to keep each single value (weight) within the range but without losing any information it contains. This is also called the Euclidean norm [128]. In ADFUNN, weight normalisation is strictly based on Euclidean normalisation in linear algebra.

Also functions are adapted to reduce the errors on a proximal-proportional basis. The proximal proportional value for each neighbouring point is based on mathematical calculation, in which:

The left hand neighbour's value = D_{RX}/D_{LR} (D_{RX} is the distance between right hand f-point to input x and D_{LR} is the distance between the left hand to right hand f-point/f-point interval), and the right hand neighbour's value = D_{LX}/D_{LR}.

Adapting the two neighbouring *f*-points proportionally which enclose the input x is theoretically reasonable in order to make sure the function is adapted accurately according to

the allocated position of Σaw. The f-point always move in the direction of reduced error, it is simply the explain of the movement that is moderated.

3.5. ADFUNN SIMULATIONS

There was a time that designing a modular simulator became difficult because of the need to consider how to dissect neural algorithms into the appropriate modules. These modules had to be flexible enough to support new algorithms. Moreover, the modules had to be computationally efficient [129].

Things were dramatically changed when Object Oriented Programming (OOP) was invented. OOP designs have an enormous advantage with respect to simulation speed [130]. Normally, interpreted environments are flexible but slow. On the other hand, compiled code is very fast, but it normally restricts the user interface. Object oriented designs combine the advantages of both methods. Object instantiation provides the user with a great deal of flexibility, while the objects themselves still run a compiled code [131].

Java is an object oriented programming language which has powerful tools to design graphical user interface [132]. It provides a highly flexible environment that allows the user to get direct access to the network parameters, provides more control over the learning details, and moreover, it provides a good visualisation for the learning results which leads to easier data analysis.

Therefore, the framework is written in Java and contains all the components of ADFUNN. It is used to create a powerful environment to train and test ADFUNN. Learning algorithms are written in code and network parameters can be adjusted in real time. Data resources to train or test ADFUNN are stored in external files and imported by the framework during learning. The real time learning processes can be visually captured through the graphical interface, including the weights and activation function changes.

In this chapter, a single layer ADFUNN is applied to three linear inseparable problems: the XOR, the Iris dataset, and the natural language processing of phrase recognition task. These tasks are simulated in the Java framework and visual results are shown as well.

3.5.1 XOR Problem

3.5.1.1 XOR Problem

The Exclusive-OR (XOR) is the simplest initial linearly inseparable test for any pattern recognition algorithm. It is a binary example, and therefore serves as a good basic test to establish that linearly inseparable problems can be solved by ADFUNN. It is important to set up this experiment before ADFUNN is further evaluated on more complex datasets.

The XOR problem can be easily described as: Consider a neural network implementation of the two input XOR function. In this function, if the two inputs match (i.e. input pattern = *{(0, 0), (1, 1)}*) the output is (0). If the inputs do not match (i.e. input pattern = *{(0, 1), (1, 0)}*) the output is (1).

3.5.1.2 ADFUNN on XOR Problem

In ADFUNN, two weights are needed for the two inputs problem and there is one output to show the result of 1 or 0. Weights are initialised randomly between -1 and 1, they are then normalised to make sure the weighted input does not go out of range of the fixed activation function x range. F point values are initialised to a constant value of 0.5 so that the final learned range of the function can be easily identified as it has been adapted towards either 0 or 1. Each F point is simply the value of the activation function for a given input sum. To proves this binary data, f points are equally spaced with an interval of 0.4, and the function value between points is on the straight line between them. This network is adapted using the the general learning rule introduced in section 3.4.6.

3.5.1.3 Simulation Result

The ADFUNN learns the problem very fast with a learning rate of 0.5. Too low a learning rate makes the network learn very slowly, but too high a learning rate makes the weights and objective function diverge. In this case, *WL* and *FL* are both *0.5*. As described in Section 3.4.2,

An example of the weights after learning is: $w_1 = 0.62$, $w_2 = 0.73$, and therefore, the sum of weighted inputs $w_1 \cdot 0 + w_2 \cdot 0 = 0$ for input pattern *(0, 0)*, $w_1 \cdot 0 + w_2 \cdot 1 = 0.73$ for input pattern *(0, 1)*, $w_1 \cdot 1 + w_2 \cdot 0 = 0.62$ for input pattern *(1, 0)* and $w_1 \cdot 1 + w_2 \cdot 1 = 1.35$ for input pattern *(1, 1)*.

As can be seen in figure 31, a characteristic XOR curve is learned. The raised curve (between 0 and 1.2) marks the learned region, within which adaptation has occurred. The data all projects onto this range, so beyond it none of the points are relevant in the final analysis.

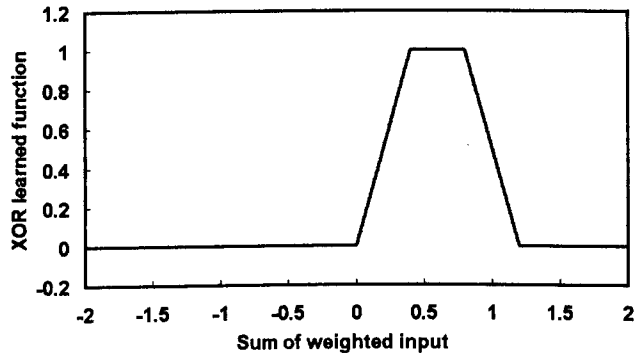


Figure 31: XOR problem solved using ADFUNN

3.5.1.4 Generalisation Ability

It can be seen from figure 31 that when input pattern is (0, 1), the weighted sum of input x = 0.73 and its corresponding $f(x) = 1.0$, which gives the accurate output as expected. Similarly, the other three inputs all give the expected correct answers. In the region projected onto between (0.3, 0.9), the slope of the activation is nearly 0 and f = 1, and beyond this region, the function slopes down towards 0. Thus, it can be summarised that ADFUNN has learned the XOR. The generalisation is 100% correct within no more than 50 epochs of training.

3.5.2 Iris Problem

3.5.2.1 Iris Dataset

Iris Dataset [133] is perhaps the best known dataset found in pattern recognition literature and therefore represents a clear benchmark to test and compare a simple linearly inseparable problem for any proposed learning algorithm. It consists of 150 four dimensional data. Four measurements: sepal length, sepal width, petal length, and petal width, were made by Fisher on 50 different plants from each of three species of Iris (Iris Setosa, Iris Versicolor and Iris Virginica) [133]. One class is linearly separable from the other two, but the other two are not linearly separable from each other.

The dataset contains three classes of Iris flowers collected by R.A. Fisher in Hawaii is a popular multivariate dataset that was introduced as an example for discriminant analysis [133]. The statistical analysis to this dataset was initially carried out by plotting the dataset onto scatter plots to determine patterns in the data in relation to the Iris classifications. And the relevant statistical information revealed from the dataset can be summarised as follows[134]: if the Iris flower has a long sepal *(6-8cm),* long petals *(5-7cm)* and wide petals *(1.5-2.5cm)* then the Iris is most likely an Iris Virginica. If the Iris flower has a short sepal *(4.5-5.5cm),* short petals *(1-2cm)* and very narrow petals *(0.1-0.5cm)* then the Iris is most likely an Iris Setosa. Any Iris flower that falls in between these two classifications is most likely an Iris Versicolor.

3.5.2.2 ADFUNN on Iris Dataset

A *4 • 3* network (in figure 32) is constructed to solve this problem using ADFUNN. The weights are initialised randomly between [-1, 1]. By looking into the dataset, the $\sum aw_j$ has a known range [-11, 11]. The Iris dataset has a precision of 0.1, so this range can be coded with a resolution of 0.1 as the F-point interval. Therefore, 221 points *((11 + 11) / 0.1 + 1 = 221)* are mathematically sufficient to encode the data precisely.

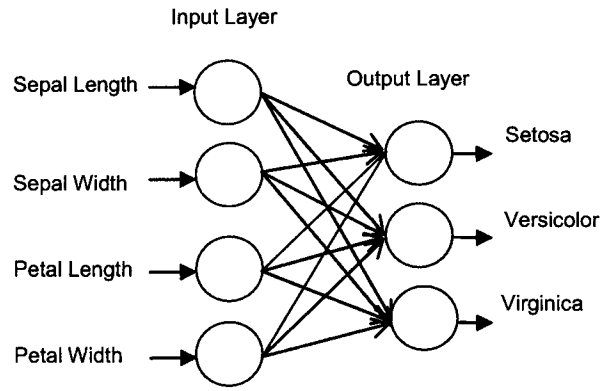


Figure 32: Single layer ADFUNN for Iris dataset

3.5.2.3 Simulation Result

Learning rates WL = 0.01 for weights, FL = 0.1 for functions are applied to ADFUNN to adapt weights and functions respectively on Iris dataset. The learning rates are chosen on the following basis. F point interval 0.1 and the activation function is adapted between [-1, 1], hence the range of function slope is [-10, 10]. To cope with weights adaptation involving a function slope of maximum 10, we set WL=0.01. In contrast, FL just depends on the required output range [0, 1] and hence FL = 0.1. Using the general learning rule of ADFUNN outlined in 3.4.6, learned function curves are achieved as in figure 33-35:

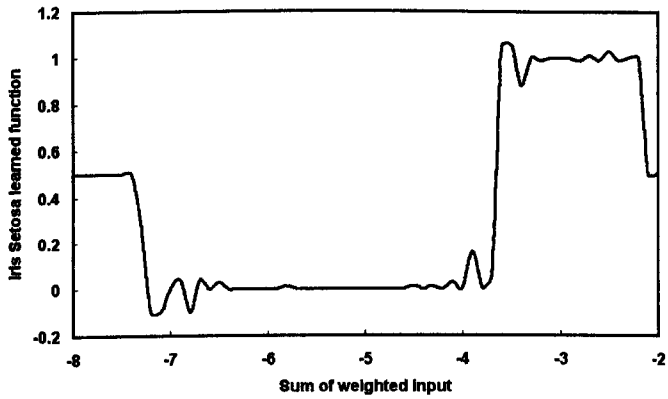


Figure 33: Iris Setosa learned function using ADFUNN

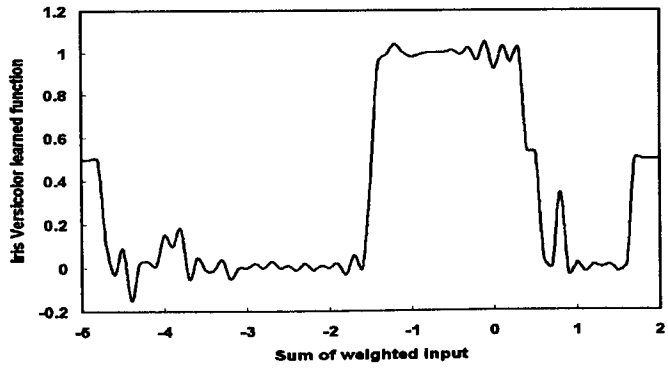


Figure 34: Iris Versicolor learned function using ADFUNN

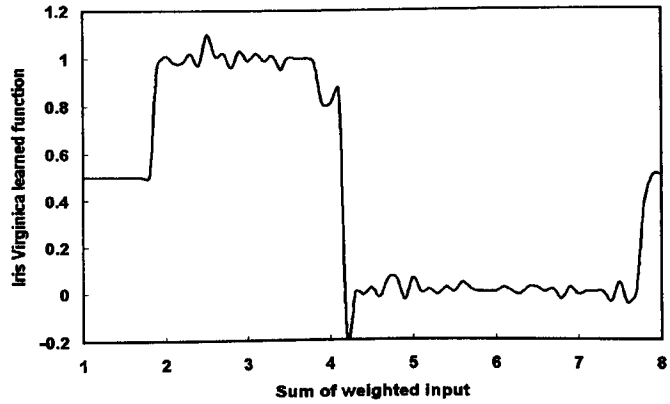


Figure 35: Iris Virginica learned function using ADFUNN

3.5.2.4 Generalisation Ability

After approximately 200 epochs, the average error for these three classes tends to a steady state, which is always below 0.05. Convergence takes an average of only about 50 epochs.

The problem is learned with 100% successful classification when all patterns are used in training, and also with only 120 (the other unseen 30 patterns are used for the independent test), whilst 90 patterns were insufficient to achieve complete generalisation in some simulations. Results are tabulated in Table 1. The functions are clearly non-linear and non-monotonic, illustrating the usefulness of the adaptive function method which is able to acquire these functions.

Table 1: Generalisation of ADFUNN for Iris dataset (150 total patterns, 100 runs)

Generalisation	Best (%)	Average (%)	Worst (%)
60 test patterns	100%	93.33%	86.67%
30 test patterns	100%	100%	100%

3.5.2.5 Comparison of Related Works Applied to Iris Dataset

There are many neural network methods for solving the popular linearly inseparable Iris plants dataset. Most of them use a network of multi-layer perceptrons (MLPs), so they require hidden neurons. In contrast, ADFUNN can solve the problem in a single weight layer supervised network.

A reinforcement learning method called SANE [135] (Symbiotic, Adaptive Neuro-Evolution) uses an evolutionary algorithm. It searches for effective connections and/or connection weights within a neural network. The weights or architecture of the neural networks are encoded in structures that form the genetic chromosomes. With the Iris dataset, SANE constructs a network of 5 input, 3 hidden and 3 output units, and the transfer function of the network is sigmoid function. The level (averaged over 50 runs) of the learning and generalisation abilities is about 90%. Cantu-Paz [136] also achieved about 90% accuracy with 5 hidden unites, and a spiking neural network with 4 hidden neurons has achieved more than 96% accuracy [137]. However, ADFUNN can achieve 100% accuracy without any hidden neuron.

There have been other attempts to solve the problem without a hidden layer, by transforming the input space, but this involves many extra input neurons. For example, Eldracher [138] use several units for each of the four measurements, in a form of coarse coding, and with 90 training and 60 test patterns they achieve an overall misclassification rate of 2.5%.

3.5.3 Natural Language Phrase Recognition Problem

According to George F. Luger [139] "One of the long-standing goals of artificial intelligence is the creation of programs that are capable of understanding and generating human language.

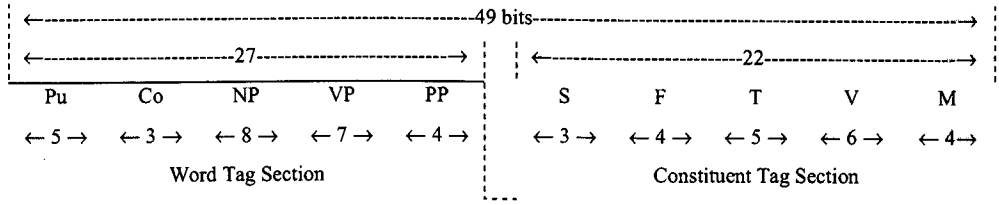
Not only does the ability to use and understand natural language seem to be a fundamental aspect of human intelligence, but also its successful automation would have an incredible impact on the usability and effectiveness of computers themselves".

Natural Language Phrase (NLP) datasets have been applied to many machine learning algorithms [140, 141], providing a challenging comparison task for ADFUNN. ADFUNN is applied to a natural language processing task of phrase recognition on a set of phrases from the Lancaster Parsed Corpus (LPC) [142].

3.5.3.1 Natural Language Phrase Recognition Data Source

The input patterns are generated using the pre-tagged corpus in [142]. It is achieved by separating the input space into several regions where each corresponds to a different symbol type. A total of 49 bits are used to encode all possible input symbols as shown in table 2. The terminal symbol groups are: punctuation (Pu), conjunctions (Co), nouns (NP), verbs (VP) and prepositions (PP). The non-terminal symbol groups are sentences (S), finite clauses (F) non-finite clauses (T), major phrase types (V) and minor phrase types (M). There are 4 look-back symbols, 10 phrasal symbols and 1 look-ahead symbol, which make a total of 15 inputs symbols. Thus, the total number of inputs is 49 bits x 15 symbols = 735. 254 input patterns are used from the pre-tagged sentences in [143, 144].

Table 2: Input Fields Representation



According to LPC [142], constituent tags can be sub-divided into five main groups: sentence tags, finite clause tags, non-finite and verbless clause tags, major phrase tags, and minor phrase tags. There are 41 constituent tags altogether [143], and so 41 outputs are needed. The anticipated output value is either 0 or 1 and only one output should ideally be 1 for each input pattern. The format of the output is, for instance: 1000...000. The first output in this case represents sentence and so it is 1, the other 40 outputs are 0. This indicates the input pattern is

a sentence. When the output is, for example: 0...010...0, and the 17th output is 1, the input pattern should be a verb phrase.

3.5.3.2 Natural Language Processing Background

Connectionist parsers are neural network based systems designed to process words or their tags to produce a correct syntactic interpretation, or parse, of complete sentences [145]. These neural network based systems are, for instance, Multi-Layer Perceptrons (MLP), Simple Recurrent Networks (SRN), Recursive Autoassociation Memory (RAAM), or localist networks [145, 146, and 147].

In the early 1990s, the first modular distributed parsers typically consisted of combinations of feed-forward multi-layer perceptrons (FF-MLP), SRN and RAAM architectures. The purpose of decomposing the parsing task into sub-modules is often to: "simplify the network's learning task, and to reduce training set size and complexity; or to evaluate the computational and cognitive plausibility of a given composition of modules" [145].

The hybrid connectionist parser is a hybrid of neural network and a symbolic module, which builds the neural network based on a given context free grammar. It has an advantage over traditional parsers because of the use of graded activation and activation passing in the network [148]. The symbolic modules are typically: "short-term storage of parse states; long-term storage of structured knowledge, such as grammar rules, semantic networks, and tree structures; and symbol manipulation and communication to control the parsing process and coordinate interactions between (connectionist) modules" [145]. The ability to learn to represent syntactic structures from examples automatically, without being presented with symbolic grammar rules is the key aspect for evaluating the connectionist parsers. In particular, learning phrase recognition has proved to be a good test of neural networks, as both learners and parsers [145].

3.5.3.3 ADFUNN on Natural Language Phrase Recognition Task

A single layer network with 735 input and 41 outputs is required to deal with the phrase recognition, using the 254 input patterns generated. Weights are initialised to very small random numbers between -0.1 to 0.1. F-points are initialized to 0.5. $\sum aw$ has a know range: {-5, 5} after analysing the input patterns and therefore 2001 points are considered sufficient to encode the data precisely, giving a resolution of 0.005 on the function.

The f-point interval is 0.005 and the activation function is adapted between [-1, 1], hence the range of function slope is [-200, 200]. To cope with weights adaptation involving function slope, WL is set to 0.00001. Whereas FL depends solely on the required output range [0, 1] and therefore just one decimal place is needed. So WL = 0.00001 is picked to reduce the effect of big slopes on weights adaptation and FL = 0.1 is picked to adapt functions. According to the general learning rule, $\Delta w = WL \bullet Fslope \bullet A \bullet E$ and $\Delta f(\sum aw) = FL \bullet E$ are used to adapt weights and functions respectively.

3.5.3.4 Simulation Result

Over many simulations, each consuming about 400 epochs, the average error tends to 0, and 100% generalisation can be achieved with 200 training patterns (out of 254). Only 3 phrase types are listed out of the 41 in figures 36, 37 and 38. They are for the major types phrases: sentence, verb phrase and noun phrase (in which f-points are also initialized to a constant value (0.5) making it easy to identify the active range over which adaptation has occurred).

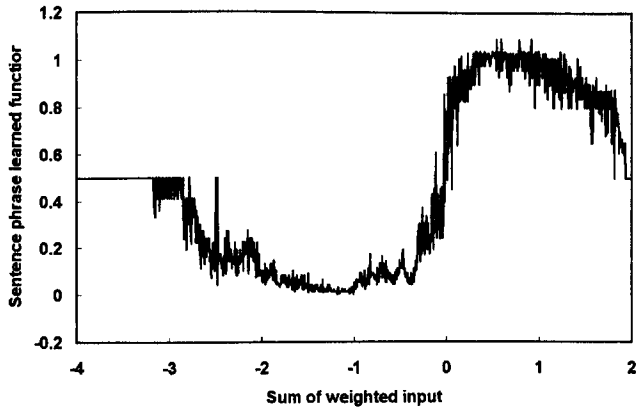


Figure 36: Sentence phrase learned function

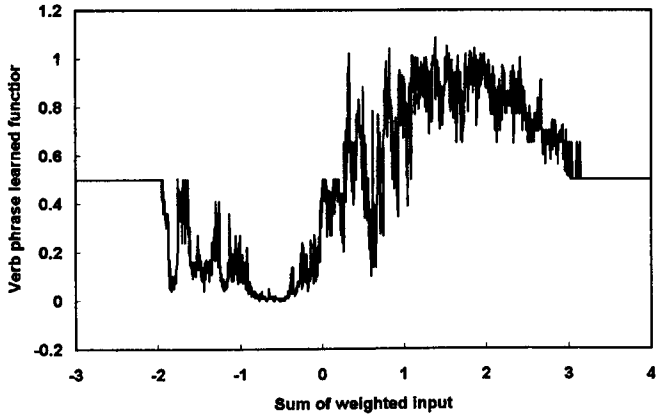


Figure 37: Verb Phrase learned function

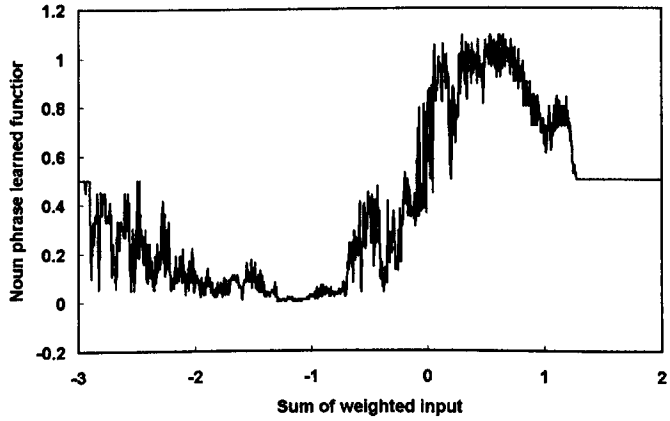


Figure 38: Noun Phrase learned function

3.5.3.5 Generalisation Ability

Table 3: Generalisation of ADFUNN on Phrase Recognition (254 patterns, 30 Runs)

Generalisation	Best (%)	Average (%)	Worst (%)
54 test patterns	100%	97.2%	94.4%
104 test patterns	90.2%	86%	81.2%
154 test patterns	81.1%	73.7%	66.2%

This experiment is evaluated by independent test with running for 30 times (30 simulations). 100% generalisation cannot be achieved using 150 training patterns (out of 254), however

with 200 training patterns, 100% generalisation can be achieved in most of the cases. The functions are obviously non-linear and non-monotonic, illustrating once again the value of the adaptive function method which is able to acquire these functions.

3.5.3.6 Comparison of MLP with Back-Propagation Applied on this Task

The performance of ADFUNN is compared with the MLP connectionist parser phrase recogniser from [143] and a simple back-propagation network, in table 4. TD is the result for the training data, ND is the result for testing using natural test data and PD is the result for testing using just pure test data. Natural test data is where patterns that have already occurred in the training are left in the test dataset, whereas for the pure test data they are removed.

The most obvious advantage of ADFUNN is the lack of hidden neurons, whereas 50 hidden layer nodes are required in the other two networks. Additionally, ADFUNN achieves 100% correct classification compared to 98.76% and 89.01% for the best of MLP and back-propagation networks.

Table 4: Comparisons between ADFUNN and MLP with back-propagation

Network Type	Total No. of input	No. of patterns per epoch	No. of epochs	Total no. of patterns presentation	No. of hidden layer nodes	No. of corresponding Tags	Correct Classifications	Overall performance (%)
Connectionist parser MLP phrase recogniser	2588 (TD)	2588	800	2070400	50	72	-	-
	2765(ND)	-	-	-	50	72	2461	89.01
	2433 (PD)	-	-	-	50	72	2132	87.63
Back propagation	127 (TD)	127	~128	16256	50	14	125	98.76
	154 (ND)	-	-	-	50	14	135	87.80
	89 (PD)	-	-	-	50	14	74	83.05
ADFUNN (30 simulations)	**127 (TD)**	**127**	**~400**	**50800**	**0**	**14**	**127**	**100.0**
	154 (ND)	-	-	-	**0**	**14**	**142**	**92.21**
	89 (PD)	-	-	-	**0**	**14**	**78**	**87.6**

3.5.4 ADFUNN vs. Adaptive Cubic Spline Activation Fucntion

ADFUNN is compared with an adaptive spline activation function neural network is investigated, as well as their performances and recent developments. The adaptive cubic spline neural network is in some ways the closest relative of ADFUNN. In this section, ADFUNN and the adaptive spline neural network are applied to a number of datasets and performances are compared, based on their network structures, learning methodologies, and hardware costs.

3.5.4.1 Comparison of Performance on Continuous XOR (cXOR) Problem

In Chapter 2.4, an adaptive spline activation function neural network was introduced [110, 111, 112, and 113]. The Catmull-Rom cubic splines are used as the activation function in the network. The spline parameters and weights are adapted together by the back-propagation learning method.

An evolution of this cubic spline activation function is reported by Mayer and Schwaiger [149]. In [149], experiments were set up to address the performance of the cubic spline network in comparison to a generic MLP with conventional sigmoid activation function. Intuitively, the decision boundaries in classification can be modelled much easier by cubic splines than by the logistic function.

A simple continuous XOR problem with piece-wise linear decision boundaries is used to evaluate and compare the single layer ADFUNN with this adaptive cubic spline function neural network (as well as to compare to the MLP, although better performance by ADFUNN has already been reported in this thesis). This continuous XOR problem is applied to ADFUNN under the same conditions as it applied to the other two.

The contiuous XOR (cXOR) is defined by a division of the unit square into four squares of identical area separated by line $x = 0.5$ and line $y = 0.5$. The lower left and the upper right square are labelled with binary value 1 and the two other squares represent the class with binary value 0. The training set is composed of 100 randomly selected points in each unit

square and the test set is made of 10,000 equally spaced points covering the complete unit square.

Two input and one output neuron are required for all the networks. The binary output value 1 is defined by an activation of the output neuron $a_o > 0.5$; otherwise the output is mapped to 0.

In [149], the experiment on cXOR with cubic spline activation function was set with a maximum of three hidden neurons and the sensitivity interval of the neurons' cubic spline to [-10, 10]. In 4 runs out of the total 20 runs, 1 hidden neuron was used to get the best performance. For the other 16 runs, better performance is achieved without a hidden neuron. They then summarised that "a single perceptron cannot learn the XOR function [10]" but in fact the result they achieved (which is: there are 16/20*100% = 80%) indicates an 80% chance that a cubic spline activation function neural network can solve the XOR problem without a hidden neuron.

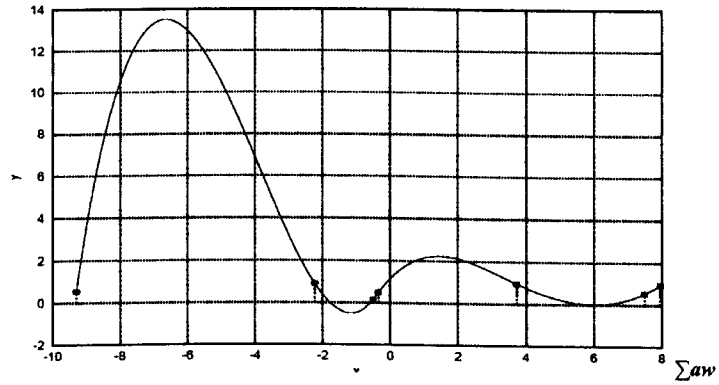


Figure 39: Cubic spline activation function for cXOR in the 2-1-1 network [149]

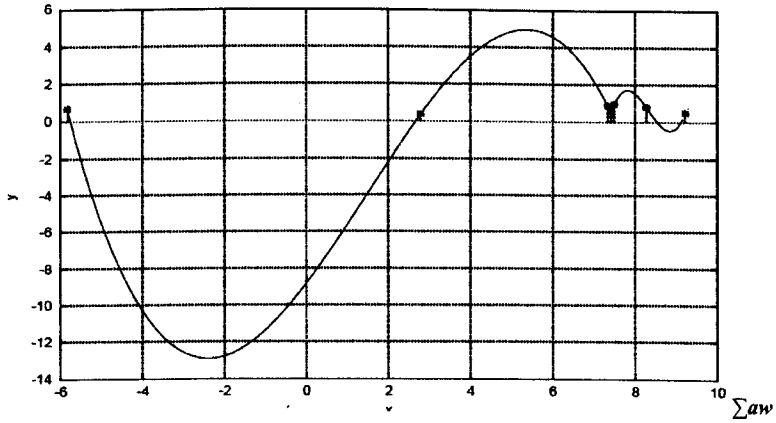


Figure 40: Cubic spline activation function for cXOR in the 2-1 network [149]

Table 5. Chance that a Hidden Neuron is Not Needed

	ADFUNN	***Cubic Spline***
Chance of No Hidden Neuron	100%	80%

In contrast, ADFUNN always solves cXOR (without a hidden neuron). In ADFUNN, there are two inputs and one output. The following is a comparison table for ADFUNN, cubic spline activation function neural network and a generic MLP on the cXOR problem. It is obvious that both the MLP with Sigmoid and Cubic Spline require hidden neurons, whereas ADFUNN can learn the cXOR problem in a single weighted neural network. Furthermore, neither of these two methods achieved better performance than ADFUNN, with in average 95.8% to 96.7% compare to ADFUNN's 98.9% classification.

Table 6. Comparison between ADFUNN, Cubic Spline and MLP on cXOR Problem

		cXOR	
Activation function	***hidden***	***Test Set***	
		Average	***Best***
(MLP)Sigmoid	2.8	0.9679	0.9752
Cubic Spline	2.5	0.9583	0.9767
(ADFUNN)Piecewise linear	**0**	**0.989**	**0.9994**

3.5.4.2 Comparison of Performance on Iris Dataset

Catmull-Rom spline activation function is applied to the Iris dataset [133] in [150]. The activation function proposed is based on the Catmull-Rom spline function where each patch curve is a polynomial of degree three. Four control points for the x-axis and four for the y-axis are used to control the slope of the activation function. For the Iris dataset, the network is constructed with four inputs, four hidden neurons, three outputs and error back propagation is used as the learning algorithm.

Although 100% successfully trained within 30,000 epochs, the simplicity of network implementation, learning speed and even hardware implementation of this Catmull-Rom spline activation function are not comparable with ADFUNN. ADFUNN achieved 100% classification within 200 epochs and without a hidden neuron (as introduced in section 3.5.2). The following compares the performance between them:

Table 7. Comparison between ADFUNN and Catmull-Rom Spline on Iris Problem

	Hidden Neurons	**Epochs**	**Classification**
Catmull-Rom Spline	4	30,000	100%
ADFUNN	0	200	100%

3.6. CONCLUSION

This chapter explained how and why the system was chosen and the learning algorithms of ADFUNN were introduced, as well as a general learning rule for ADFUNN.

ADFUNN was then applied to a range of linearly inseparable problems which normally require a hidden layer to solve. Although these tasks can be solved by introducing a multi-layer structured neural network, the success of ADFUNN is its clear advantages in training speed and computational requirements. A multi-layer perceptron is in the inferior position compared to ADFUNN in the sense of much higher computation and architecture complexities, when being attempted on the same task.

As compared in table 4, the single-layer ADFUNN, with 735 inputs and 41 outputs, has more computational efficiency than a hierarchically structured multi-layer back-propagation (735 inputs, 50 hiddens and 41 outputs), in terms of much less computer memory and CPU usage and less time consumption.

ADFUNN is also more efficient and powerful than the adaptive cubic spline function. Order expressions can be defined as follows, in order to compare the complexity of the order of computations for ADFUNN and adaptive cubic spline function.

E: Number of epochs
N: Number of neurons
X: Number of f-points or control points which need to be calculated when a pattern is passed to a neuron
C: Calculation required per f-point (ADFUNN) or control point (cubic spline), e.g. f(x) = ax
M: Memory usage per calculation per f-point (ADFUNN) or control point (cubic spline)
O (nn): Order of computations of a neural network for all patterns
O (p): Order of computations of a pattern passing through a neural network
O (n): Order of computations of a neuron

$$O(nn) = E \cdot O(p) \cdot O(n)$$
$$O(p) = N$$
$$O(n) = X \cdot C \cdot M$$

To compare ADFUNN and adaptive cubic spline function on the Iris dataset, the order of calculations per neuron is related to the number of f-points (ADFUNN) or control points (adaptive cubic spline function) which need to be calculated when a pattern is passed to the neuron, the calculation of each point and the memory cost per calculation. For ADFUNN, only two points which bound the input value and the input point itself need to be calculated, however, for the adaptive spline function, there are 8 control points (4 for x-axis and 4 for y-axis) need to be calculated using $f_i(x) = f_i(x_i) + a_i(x-x_i) + b_i(x-x_i)^2 + c_i(x-x_i)^3$. Therefore

For ADFUNN $O(n) = 3 \cdot C \cdot M$, for adaptive cubic spline $O(n) = 8 \cdot C \cdot M$

The order of computations of each pattern going through the networks $O(p)$ is related to the number of neurons. For ADFUNN, as there is no hidden neurons, the calculation only happens in the output neuron, $O(p) = 1 \cdot O(n) = 1 \cdot 3 \cdot C \cdot M = 3 \cdot C \cdot M$. Whereas for the

adaptive cubic spline function, there are 4 hidden neurons and an output neuron, so $O(p) = 5 \cdot O(n) = 5 \cdot 8 \cdot C \cdot M = 40 \cdot C \cdot M$.

The overall order of computations of an algorithm is related to the number of epochs, order of computations per pattern and order of computations per neuron. For ADFUNN, the $\boldsymbol{O\ (nn)} = E \cdot O\ (p) \cdot O\ (n) = 200 \cdot 3 \cdot C \cdot M = \boldsymbol{600 \cdot C \cdot M}$ and for adaptive cubic spline, $\boldsymbol{O\ (nn)} = E \cdot O\ (p) \cdot O\ (n) = 30000 \cdot 40 \cdot C \cdot M = \boldsymbol{1200000 \cdot C \cdot M}$. The calculation for each control point in the adaptive spline function is a three degree polynomial function; for ADFUNN it is a linear function for the neighbouring two points and a two degree polynomial function for the input point. And obviously, ADFUNN consumes less memory per calculation per point. Thus, ADFUNN scales up computational over 2000 times (1200000/600 = 2000) more efficiently than the adaptive cubic spline function.

In addition, ADFUNN is extremely easy to implement, very high in training speed, and requires much less hardware memory to learn.

CHAPTER 4 A MULTI-LAYER ADAPTIVE FUNCTION NEURAL NETWORK (MADFUNN)

4.1 INTRODUCTION

In the previous chapter, the single layer ADFUNN has been shown to be effective on some linearly inseparable datasets. These datasets are small or medium sized data. As the size and complexity of the model required increases, at some point ADFUNN's performance is limited.

Multi-layer ADaptive Function Neural Network (MADFUNN) is introduced to solve the practical problem of finding a suitably restricted subset of functions f which would have good representational capacity but would also form a space with a structure regular enough to enable efficient learning for complex models with large datasets.

In this chapter, a multi-layer adaptive function neural network extended from ADFUNN is introduced, a new learning rule for MADFUNN and its applications are also introduced.

MADFUNN's introduction is to enable efficient learning for more complex classifications with large datasets. Although ADFUNN has exhibited a highly effective generalisation ability, if the dataset itself is too complex, a single layer ADFUNN is not capable of dealing with the problem. Therefore, a multi-layer ADFUNN emerges with the ability to encode more functions and information.

4.2 THE MADFUNN ARCHITECTURE

MADFUNN is composed of a layer of input neurons which are fully forward connected to a layer of hidden neurons. The hidden layer is then fully connected to an output layer of neurons. Like the back-propagation algorithm, MADFUNN propagates inputs forward to a hidden layer and then to the output layer in the usual way. All outputs are computed using a linear piecewise function and MADFUNN propagates the errors backwards by apportioning them to each unit according to the amount of this error the unit is responsible for. After feeding the sum of the weighted value into a hidden or an output neuron, this value is passed through a piecewise linear activation function. Like the activation function in ADFUNN, the piecewise linear function is a continuous piecewise curve that limits the node's output. In simple words, the activations are propagated from the input to the output layer, and the error between the observed actual and the requested nominal value in the output layer is propagated backwards in order to modify the weights and functions.

4.2.1 Input Layer

The input layer in MADFUNN is composed of neurons. The number of these neurons is specified by the dimension of the data. Data can be any format of numbers, and they are normalised wherever needed to keep the sum input within range. The input layer is fully connected to the hidden layer. The calculated sum inputs will be presented as the input to the hidden layer.

4.2.2 Hidden Layer

The hidden layer is composed of a set of neurons and is fully connected to the output layer. The number of hidden neurons depends on the data complexity. It is the general critical problem a multi layer neural network always faces. Too few hidden neurons, too little "brain" available to learn the problem. If there are too many neurons, the network will memorise the problem instead of learning it. Taking the sum inputs as inputs, the hidden layer is able to produce outputs from the piecewise linear function. These outputs will be passed to the output layer.

4.2.3 Output Layer

Taking the outputs from the hidden layer as inputs, the output layer is also able to produce the results which are the actual outputs from MADFUNN. The number of neurons in the output layer is the number of classes of the data. These actual outputs will be compared with the desired outputs to produce the errors. In MADFUNN, errors from the output layer are sent back to the hidden layer for weight adaptation.

4.3 THE SYSTEM LEARNING

4.3.1 Learning Rates for Weights and Functions

As described in 3.4.2, different learning rates are needed for weight and function adaptations. This is mainly because of the different factors the weights and functions depend on. As there are different input data ranges for the hidden layer and the output layer, in the sense that the input data to hidden layer depends on the dataset whereas the input to output layer mostly depends on the number of hidden neurons. Therefore different weight learning rates are required for hidden neurons and output neurons, as well as different function learning rates.

4.3.2 Errors in Output Neurons and Hidden Neuron

The errors in the output layer are simply the difference between the actual output and the expected output. For instance, the error from output neuron *o* is:

$$E_o = t_o - y_o$$

Where t_o is the target output and y_o is the actual output. The errors from the output nodes are propagated back to the hidden neurons. For each hidden neuron, the error is calculated as:

$$E_H = \sum_{Y=1}^{n} E \bullet W_{HY} \bullet FSLOPE_Y$$

It sums the product of error, weight and slope from each output neuron.

4.3.3 Weight Normalisation and Weight Limiter

Weight normalisation is done by calculating the weights' vector's magnitude and dividing each of the weights by this magnitude as in ADFUNN. The result of this process is a weights vector whose magnitude is 1, but whose direction remains the same as the original vector. The aim of this is to keep the weights in range without losing any information. Another method of doing this is to use a weight limiter. The reason to introduce a weight limiter is to make the function more stable. Having said that, as MADFUNN is particularly introduced to solve complex problems which normally have large data ranges and overlapping classes. Therefore small f-point intervals are required for the activation functions. For example, if the f-point interval is 0.001, because the functions are adapted between {0, 1}, function slopes can be any value between {0, 1000}. Large slopes can make large swing errors and therefore make the learning unstable. Thus the weight limiter is used in MADFUNN to limit the weights between {-1, 1}. Using a weight limiter requires the weights to be initialised in very small numbers or as zero, and they are adapted slowly in each epoch.

4.3.4 Weight Adaptation

In MADFUNN, the output neuron weights are adapted according to the delta weight. The delta weight is calculated by multiplying the hidden neuron activation, the output error, the function slope and the weights learning rate together.

$$\Delta W_{HY} = WL \bullet FSLOPE_Y \bullet A_H \bullet E$$

Each weight from a hidden neuron to an output neuron is adapted by this delta weight and if no change is made beyond the weight limitation.

$$W_{HY}' = W_{HY} + \Delta W_{HY}$$

Likewise, the hidden neuron weights are adapted in the same way except the error and slope is from the hidden neuron and A is the input node activation.

$$\Delta W_{IH} = WL \bullet FSLOPE_H \bullet A \bullet E_H$$

Each weight from an input neuron to a hidden neuron is adapted within the weight limiter by:

$$W_{IH}' = W_{IH} + \Delta W_{IH}$$

4.3.5 Functions Adaptation

As introduced in 3.4.5, the piecewise linear activation functions in both hidden and output neurons are adapted by finding the two neighbouring f-points that bound Σaw and adapting them separately, on a proximal-proportional basis. To adapt each output function. Delta F is calculated by multiplying the function learning rate and the output error. Two f-points which bound the input activation are adapted separately based on the proximal proportional basis.

$$\Delta F_Y = FL \bullet E$$

$$F_{Y1}' = F_{Y1} + \Delta F_Y \bullet P_{Y1},$$

$$F_{Y2}' = F_{Y2} + \Delta F_Y \bullet P_{Y2}$$

The same with the hidden neuron functions. They are adapted in the same way but the delta F is calculated from the hidden neuron error and learning rate.

$$\Delta F_H = FL \bullet E_H$$

$$F_{H1}' = F_{H1} + \Delta F_H \bullet P_{H1},$$

$$F_{H2}' = F_{H2} + \Delta F_H \bullet P_{H2}$$

4.3.6 MADFUNN General Learning Rule

In this general learning rule, instead of normalising the weights, a weight limiter is introduced which will limit all weights within the range of [-1, 1]. Weight and activation functions are adapted in parallel, using the following algorithm:

A = input node activation,

A_H = hidden node activation,

E = output node error,

E_H = hidden node error,

$FSLOPE_Y$ = slope of output neuron functions,

$FSLOPE_H$ = slope of hidden neuron functions,

P_{Y1}, P_{Y2}:the two proximal-proportional values for the two neighbouring f-points for the output that bound $\sum aw$

P_{H1}, P_{H2}:the two proximal-proportional values for the two neighbouring f-points for the hidden neuron that bound $\sum aw$,

WL, FL: learning rates for weights and functions.

Step 1: calculate output error, E

Step 2: calculate hidden error, E_H

$$E_H = \sum_{Y=1}^{n} E \bullet W_{HY} \bullet FSLOPE_Y$$

Step 3: adapt weights to each output neuron

$$\Delta W_{HY} = WL \bullet FSLOPE_Y \bullet A_H \bullet E$$

$$W_{HY}' = W_{HY} + \Delta W_{HY}$$

Step 4: adapt function for each output neuron

$$\Delta F_Y = FL \bullet E$$

$$F_{Y1}' = F_{Y1} + \Delta F_Y \bullet P_{Y1},$$

$$F_{Y2}' = F_{Y2} + \Delta F_Y \bullet P_{Y2}$$

Step 5: adapt function for each hidden neuron

$$\Delta F_H = FL \bullet E_H$$

$$F_{H1}' = F_{H1} + \Delta F_H \bullet P_{H1},$$

$$F_{H2}' = F_{H2} + \Delta F_H \bullet P_{H2}$$

Step 6: adapt weights to each hidden neuron

$$\Delta W_{IH} = WL \bullet FSLOPE_H \bullet A \bullet E_H$$

$$W_{IH}' = W_{IH} + \Delta W_{IH}$$

Step 7: $F_Y = F_Y'$, $F_H = F_H'$

$$W_{HY} = W_{HY}',$$

$$W_{IH} = W_{IH}'$$

Step 8: randomly select a pattern to train

Step 9: repeat step 1 to step 8 until the output error tends to a steady state.

4.4 ANALYTICAL FUNCTION RECOGNITION

4.4.1 Motivation for Investigation

The learned function curves from XOR, Iris dataset and natural language phrase recognition tasks which ADFUNN has been applied to, are very well-regulated. It should be possible, for a given set of analytical function prototypes to determine which analytical function matches best to a given smoothed curve from the learned function output. Some points on the

activation function have never been adapted even though they are within the active range of adaptation, in which case they are in effect noise. To smooth these curves, which include the smoothing of unlearned points, a simple moving average method, least-squares polynomial smoothing, and a new self-sizing moving window method are applied and compared. The smoothed curves are then available for function recognition. The implementation of this function recognition task also helps to interpret the hidden features of data and understand what happens inside the network.

4.4.2 Patterns Generation

Six commonly used analytical functions classes plus one random numbers curve class are selected as the prototypes to train the network. The random number class is used for any case where the testing curves do not resemble one of the given function classes. Each analytical function has 200 training patterns generated, which makes 1400 training data in total.

The analytical functions selected in the training data are based on the general shapes of ADFUNN learned function curves in different applications. Piecewise linear function is the activation function which is being adapted during the learning of ADFUNN. The supervised ADFUNN always adapts its functions towards 1 or 0. Therefore, the analytical functions whose shapes most closely resemble the ADFUNN's learned function curves are: pulse function, step function, sine function, sigmoid function, straight line and RBF.

200 patterns are generated for each of the six analytical functions and one random function. These patterns are transformed and normalised within the range of {0, 1} in the two dimension space. There are 101 points used as the input data within this range. Having looked at the general shapes generated from ADFUNN's learned functions in the XOR problem, Iris dataset, the phrase recognition task and considered the nature of ADFUNN being adapted either towards 0 or 1, the training patterns are generated as follows:

For Step and Pulse function classes, patterns are generated using the piecewise constant function $y=1$ moving with different size of intervals on the range of x belongs to {0, 1}.

Because the learned functions in ADFUNN normally have one or two peak (where $y=1$) intervals, the patterns generated also have one or two peaks.

For Sine function class $y = sin\ (bx+c)$, the patterns are generated with angle b varies between $\{0, 2\pi\}$ and c varies between $(0, \pi)$. For Sigmoid function class $y = \frac{1}{1+e^{-(x+a)}}$, a varies between $\{-1, 0\}$. The Straight line function $y=ax+b$ class patterns are generated based on the gradient a varies between $\{0, 1\}$ and intercept b varies between $\{0, 1\}$ where the incremental change for a is $a = a+0.2$ and for b is $b = b+0.1$. And the RBF function $y = e^{-a(x-b)^2}$ class patterns are generated based on a varies between $\{0, 10\}$ and b varies between $\{0, 1\}$. For the Random Number class, the 101 input points are randomly selected between $\{0, 1\}$ for every training pattern.

For instance, the following is a Sine function, $f(x) = sin(5x)$, on the x range of $\{0, 1\}$, its original curve is illustrated in figure 41.

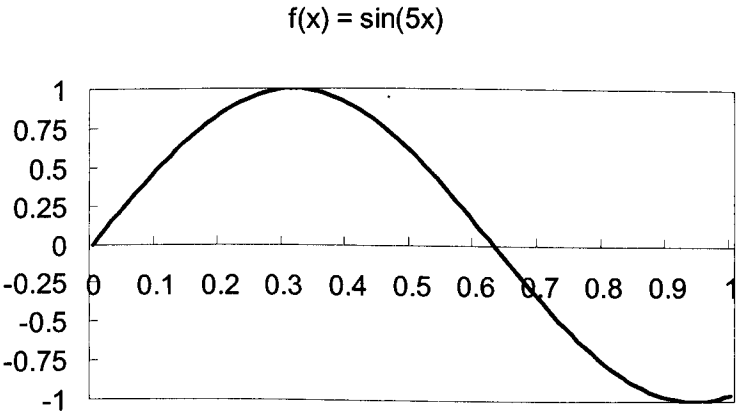


Figure 41: Sine function $f(x) = sin\ (5x)$

After normalisation to the range of $\{0, 1\}$, it has a pattern as shown in figure 42:

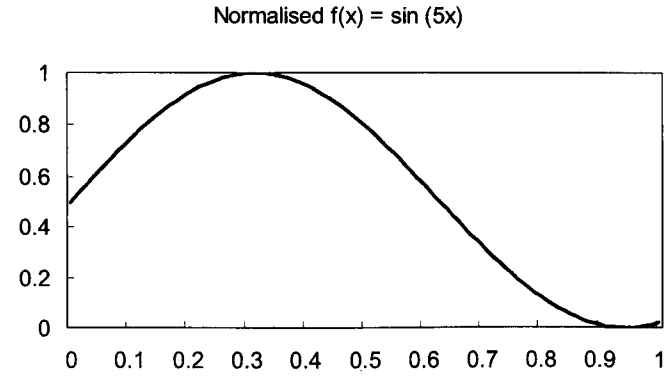


Figure 42: Normalised sine function f(x) = Sin (5x)

Figure 43 and 44 are examples of pulse function and step function.

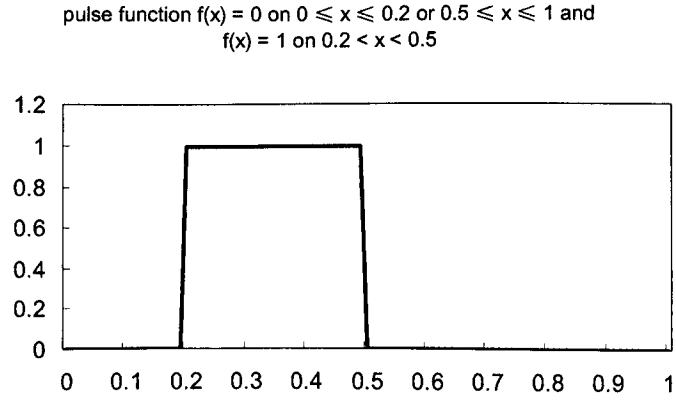


Figure 43: Pulse function example

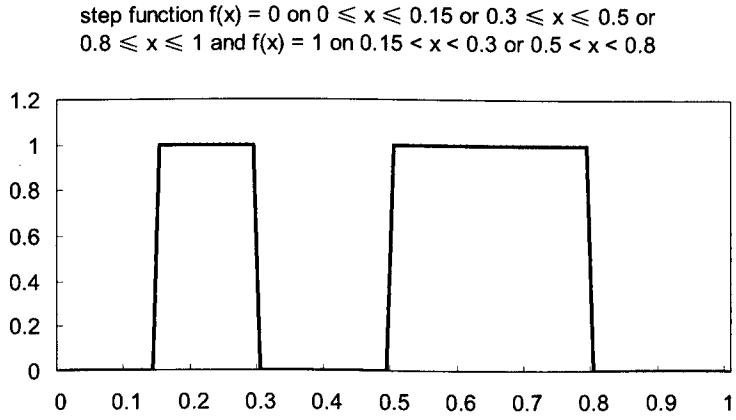


Figure 44: Step function example

Patterns are generated and then normalised in the same way, whereas random f(x) values are generated for the random analytical function class as illustrated in figure 45.

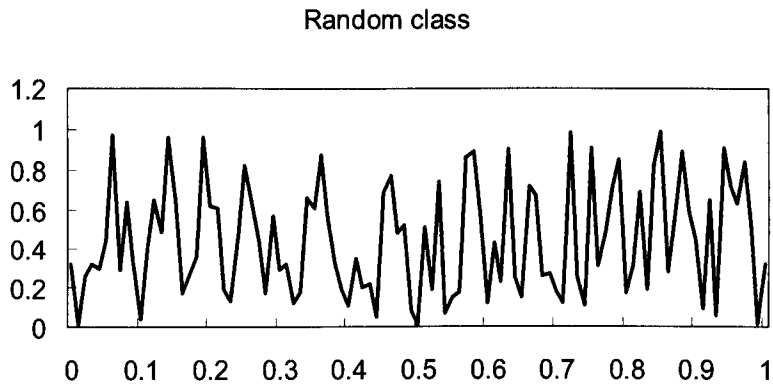


Figure 45: Random analytical function example

4.4.3 Analytical Function Recognition Insolvable Using a Single Layer ADFUNN

A single layer ADFUNN is trained with the generated 1400 patterns in order to test and recognise the best matched analytical function from the empirical ADFUNN learned function curves, such as learned ADFUNN for Iris dataset and natural language recognition.

Data are generated or normalised within the range of {0, 1} in the x-axis with precision of 0.01, hence the vector size is 101 for the input data. For the 7 typical analytical function outputs, 1260 (90% of the total patterns) training patterns are randomly selected from the total 1400 patterns in each run.

In this single layer network, weights are initialised randomly between -1 and 1, and then normalised. F point values are initialised to 0.5 so that the learned function curve can be clearly and easily seen. The network is adapted using the general learning rule of ADFUNN outlined in 3.4.6. However, the best generalisation ability is only 89.26% with 90% patterns used for training as shown in table 8.

Table 8: Generalisation of ADFUNN on Analytical Function Recognition (1400 patterns, 30 Runs)

Generalisation	Best (%)	Average (%)	Worst (%)
140 test patterns	89.26%	83.75%	78.25%
280 test patterns	78.92%	74.28%	69.64%

4.4.4 Complexity and Availability of the Task

As experimental results show table 8, 100% correct classification is not possible to be achieved with these 1400 patterns using the single layer ADFUNN. The task of replacing learned functions with matched analytical ones is an undeveloped field and challenging topic in artificial neural network research. Identifying the closet analytical function to an empirical one is a complex task. Firstly, not every empirical output belongs to an analytical function. Secondly, the empirical output is sometimes a combination of two or more analytical ones.

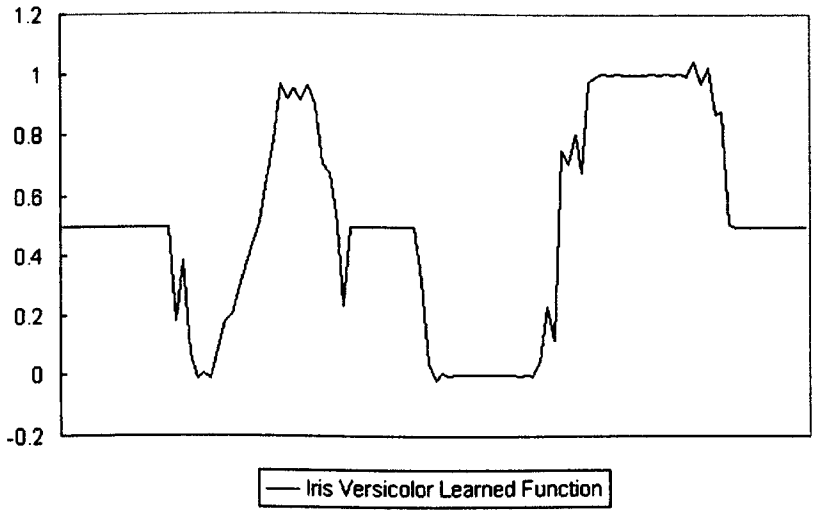


Figure 46: Example of an empirical output that contains approximations to two analytical functions

As illustrated in figure 46, the learned function of Iris Versicolor class contains two separate parts of a learned curve. It intuitively looks like a combination of a sine and a pulse function when the noise is ignored. Sometimes, the learned function can be more complicated than this illustration. Therefore, it is rather difficult to classify an experimental learned function curve to an analytical function.

However, the implementation of this function recognition task is significant because it helps to interpret the hidden features of data and understand what happens inside the network.

4.4.5 MADFUNN on Analytical Function Recognition

As shown in Table 8, ADFUNN is not able to successfully classify the analytical function recognition task due to the highly complex dataset and insufficient network structure. In this case, to deploy ADFUNN with a layer of hidden neurons will enable the network to encode more functions and information. This may be more efficient than other networks in terms of the dual-modes of learning. By adding a hidden layer of neurons to ADFUNN, MADFUNN is constructed by 101 input neurons, 30 hidden neurons and 7 output neurons to solve this function recognition problem. Weights are initialised as small random numbers between [-0.1, 0.1] for output neurons and [-0.4, 0.4] for the hidden neurons limited by a weight limiter. F-points are initialised to 0.5 in order to easily reveal the learned function range. In the same way as ADFUNN, F-points are equally spaced on the function and the function value between points is on the straight line joining them. A slope limiter is also applied to ensure that the adaptation to weights will not be too large which would produce instability. The two learning rates FL and WL are equal to 0.1 and 0.0001 respectively. The weights learning rate WL for output neurons is chosen on the following basis:

The change of a weight is related to several factors: net input, corresponding output function's slope and error. When a slope limiter is applied, a relatively smaller WL (0.0001) can help to balance any over-adaptation caused by these factors and thus produce a more stable weights adaptation. In contrast, function learning rate FL just depends on the required output range [0, 1]. After testing, these training patterns' $\sum aw_j$ have a known range of [-1, 1] for output neurons and [-10, 10] for hidden neurons. It has a precision of 0.01, so 201 and 2001 points

are enough to encode all training patterns for output and hidden neurons respectively. Different weights initialisation range and f-point ranges are being set for output and hidden neurons because the maximum range of sums of weighted inputs to output and hidden neurons are different.

4.4.6 Simulation Result

The network is trained using the general learning rule of MADFUNN. Within 50 epochs in each run (experimented for 30 runs), 100% correct classification can be achieved. The following function outputs can be obtained. Only the sigmoid function in figure 47, RBF function in figure 48 and pulse function in figure 49 are listed here.

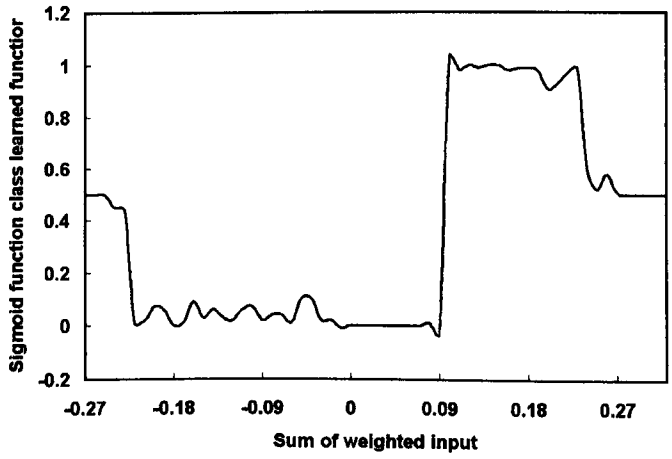


Figure 47: Sigmoid function recognition output using MADFUNN

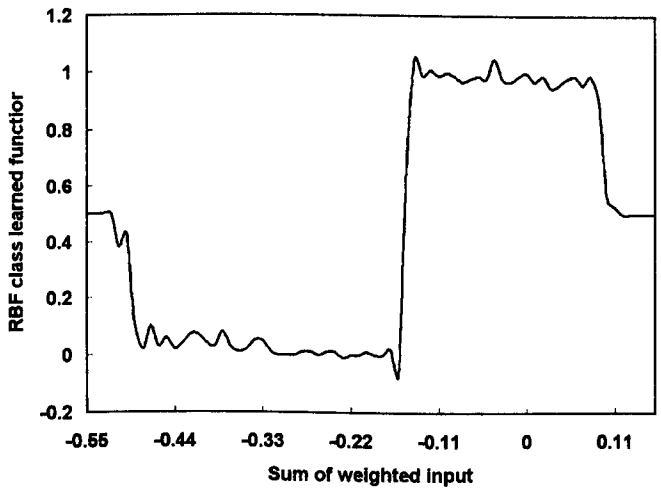


Figure 48: RBF function recognition output using MADFUNN

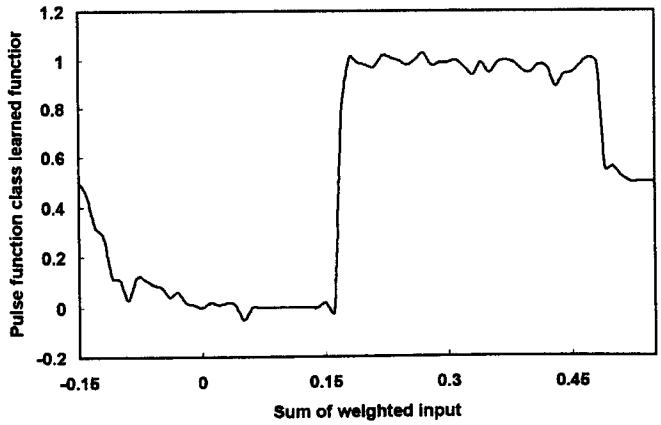


Figure 49: Pulse function recognition output using MADFUNN

Apparently, the function outputs in figures 47 - 49 are very well regulated in the sense that the learned region has very clear and sharp steps. For sigmoid function in figure 47, a characteristic curve is learned. The curve range shown is the learned region, within which adaptation has occurred. All the data are projected onto this range, and points outside the range are not relevant in the final analysis.

In the region projected onto between [0.09, 0.23], the slope of the activation is nearly 0 and f = 1, the sigmoid function is activated in this range. Beyond this region, the function slopes drop down towards 0. In conjunction with the weights on the input to this neuron, it is able to recognise input patterns that resemble Sigmoids.

Figure 50 illustrates is a typical form of learned hidden neuron function output. It is a kind of cluttered and characterless curve between two straight lines, except that there are clear spikes.

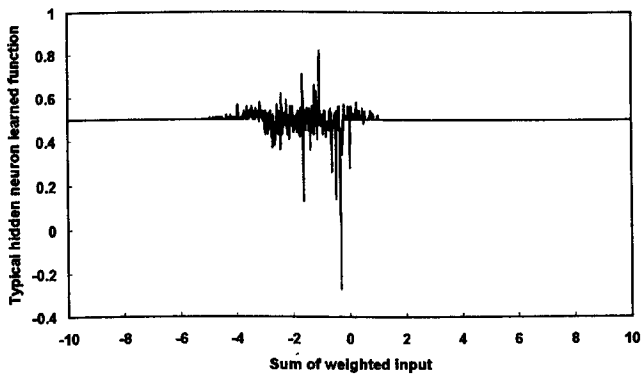


Figure 50: Typical form of a hidden neuron function output

4.4.7 Generalisation Ability

The analytical function recognition task is learned with 100% successful classification when all patterns are used in training for 30 runs, and also with 1300 (the other 100 unseen patterns are used for independent test) patterns, whilst 1100 and 900 patterns were insufficient to achieve complete generalisation. Results are tabulated in table 9.

Table 9: Generalisation ability of MADFUNN on analytical function recognition

Generalisation	Best (%)	Average (%)	Worst (%)
100 testing 1300 training	100.0	99.8	99.7
300 testing 1100 training	91.7	86.7	81.7
500 testing 900 training	82.2	79.6	77

4.4.8 Comparison of a Simple Back-Propagation with MADFUNN

A simple back-propagation network is also built to solve this analytical function recognition task to compare with MADFUNN. This network propagates inputs forward in the usual way. All outputs are computed using sigmoid function and it propagates the error backwards by apportioning them to each neuron according to the amount of this error the nueron is responsible for. Table 10 is a comparison table between a simple back-propagation network and MADFUNN. Both of the two programs have been run for 30 simulations.

Table 10: Comparison of MADFUNN with a simple Back-Propagation(BP) neural network

Network Type	Number of training patterns	Number of epochs each run	Number of hidden nodes	% Correct Classification
Simple BP	1400	~1000	30	63.3%
	1400	~1000	80	100.0%
	1300	~1000	80	91.0%
	1100	~1000	80	85.3%
MADFUNN	1400	~100	30	100.0%
	1300	~100	30	100.0%
	1100	~100	30	91.3%

As seen from the comparison, this simple back-propagation exhibits poorer performance than MADFUNN. With the same number (30) of hidden neurons, it can only achieve 63.3 % correct classification with 1000 epochs. The best performance using back-propagation network is with 80 hidden neurons, the correct classification can reach 100%. In contrast, MADFUNN can solve this problem with 100% correct classification efficiently within 100 epochs using only 30 hidden neurons.

4.5 LETTER IMAGE RECOGNITION

The letter image recognition dataset [151] from UCI repository [152] provides a complex pattern recognition problem which is to classify distorted raster images of English alphabetic characters. This is an extremely complex dataset that even a 4 layer fully connected MLP of 16-70-50-26 topology [153] can obtain 98% correct classification with AdaBoost, but required 20 computers to implement the system. In this thesis, a Multi-layer ADFUNNs (MADFUNNs) is applied and tested in this UCI distorted character recognition task [154]. A system with two parts is constructed, letter feature grouping and letter classification. This type of system is used to cope with the complexity of the wide diversity among the different fonts

and attributes. Here 'grouping' means letters are randomly categorised into different groups initially and during the learning they will be regrouped according to the common group features.

4.5.1 Letter Image Recognition Dataset

This complex task is that of letter recognition (in figure 51) as presented by D.J. Slate [151]. The 20,000 character images consisting of on average 770 examples per letter, are based on 20 different fonts and each letter within these 20 fonts is randomly distorted to produce a file of 20,000 unique stimuli. Sixteen numerical attributes (statistical moments and edge counts) were defined to capture specific characteristics of the letter images. Each of these attributes is then scaled to fit into a range of integer values from 0 through 15. Each of the letter images is thus transformed into a list of 16 such integer values

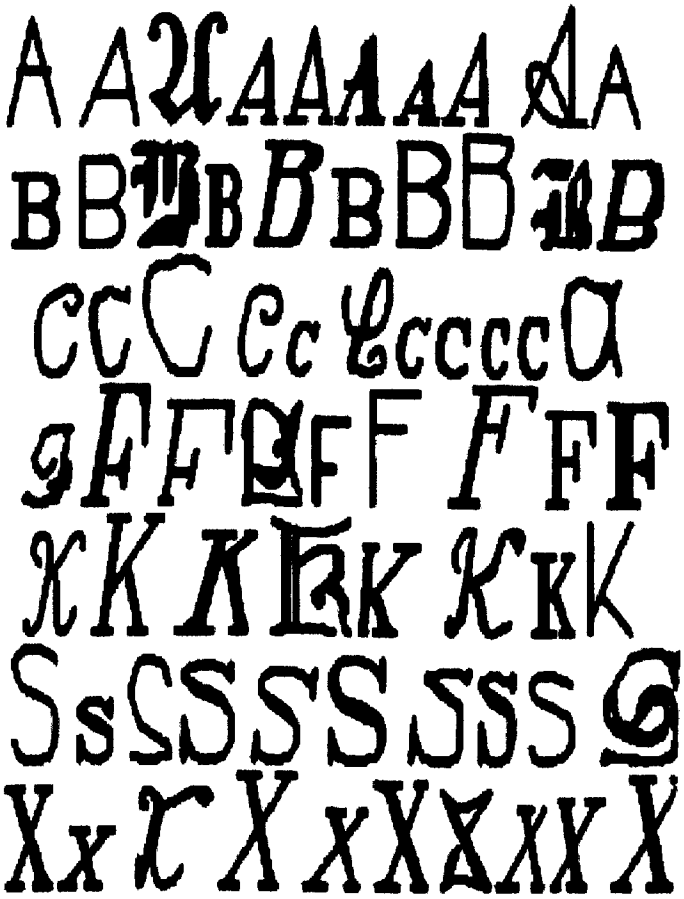


Figure 51: Examples of the character images generated by "distorted" parameters

4.5.2 MADFUNN for Letter Image Recognition

As explained in 6.3.1 this letter image recognition task contains extremely complicated data. In order to achieve the best classification performance, a group of optimised networks with letter feature grouping and letter classification are developed and investigated. The letter feature extraction is applied to the data prior to the letter classification stage, in order to simplify the complexity of the data and thus produce more accurate generalisation results. Due to the complex nature of this dataset, extracting general features of letters into groups (e.g. the common feature between letters C, O, G and even Q) can help to reduce the classification complexity, in terms of reduced number of patterns, features and classes that need to be classified in each feature group and powerful learning from the combination of networks for all groups.

The construction of these networks utilises the classic model of feature extraction followed by classification [155] which effectively balances the classification load. Theoretically, the more the feature group number is, the easier the letters will be classified in each group. On the other hand, a large number of feature groups will also increase the complexity of feature extraction. For this dataset, there are 26 letters to be classified. In order to find an optimal number of feature groups and initial number of letters in each group of features, experiments are performed with different number of groups (2, 13 and 26 which are also the factors of number 26). The results show that with 13 feature groups, the feature extraction can get the best performance. Having said that, technically 15 groups probably will get better performance than 13 groups. However 15 groups will result the unbalanced distribution of letters in each group, for example it will have 11 groups having 2 letters and 4 groups having only 1 letter.

Therefore, to extract common features, letters are firstly categorised and grouped using a supervised learning method using MADFUNN_1 (as in figure 52).

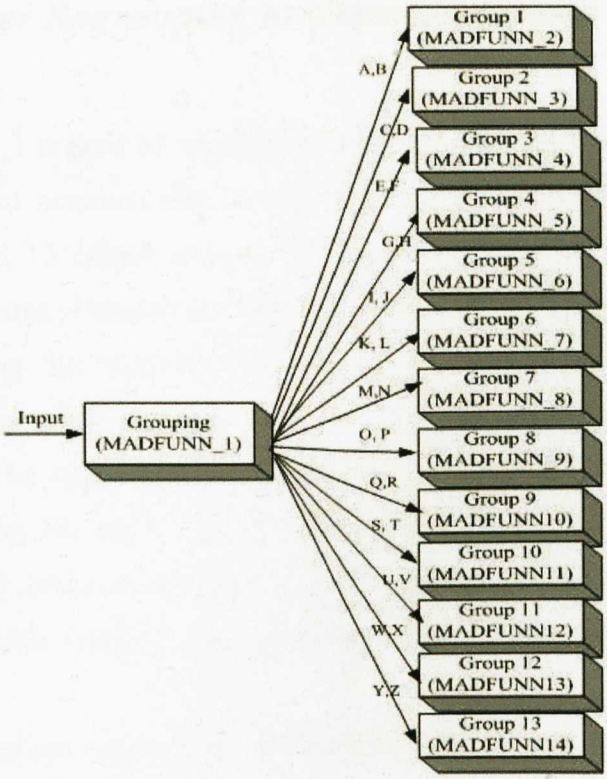


Figure 52 Networks applied to letter image recognition task

Letters are assigned and initialised to 13 groups followed by their original sequence as in the English alphabet:

Group1: A B

Group2: C D

Group3: E F

Group4: G H

Group5: I J

Group6: K L

Group7: M N

Group8: O P

Group9: Q R

Group10: S T

Group11: U V

Group12: W X

Group13: Y Z

4.5.3 Letter Regrouping to Extract Features

MADFUNN_1 is used for classifying common features of these letters. It is a neural network with 16 input neurons, 100 hidden neurons (experimentally optimised number of hidden neurons) and 13 output neurons (13 feature groups). It is trained to learn the above 13 assigned groups. Patterns are passed to MADFUNN_1 where the network is then being adapted using the MADFUNN's general learning rule. A pattern will be passed to its corresponding group (e.g. letter A goes to group1, letter M goes to group7) after the feature extraction. The stage of letter classification is performed in MADFUNNs 2-14 and they are adapted using the same general learning rule. In MADFUNNs 2-14, each has 16 input neurons, 100 hidden neurons and 4 output neurons in stage1 and 6 output neurons in stage2 (will be explained later). They are used to classify each individual letter.

In the beginning, the groups are assigned without any human judgment. When error rate reduces to a steady state, the results are analysed by a confusion matrix which contains information about actual and predicted classifications. The letters which have big confusions will be regrouped according a predefined rule. The principle of this predefined rule is to regroup a letter to the group where it will share common features with other letters. Ideally, if all letters are assigned to the correct groups in the first step, all samples for each letter should all be classified to the current group this letter is assigned to. However, because letters are assigned to these 13 groups without any human judgement, different samples for one letter may share different features with other letters. In the experiment, all letter samples are passed to MADFUNN_1 to learn these 13 groups. The result will be drawn in a confusion matrix with the distribution of samples in each group for each letter after learning. The network is trained to learn these 13 groups with all samples. The number of misclassified samples in other groups for each letter should suggest a potential feature sharing in these groups. In the first round, the majority samples for each letter should be correctly classified to its initially assigned group as being trained. However, the number of misclassified samples in one particular group class may be relatively high, say greater than nearly half of the number of samples in the expected class (empirically). That means this letter has a lot of common features with letters in the misclassified group and it should be regrouped to that group.

However, to balance the number of letters in each group, the maximum number of letters allowed to be grouped in each group should not exceed the twice of the average number of letters in each group (e.g. if there are on average 2 letters assigned in each group, the maximum number of letters allowed to be regrouped to one group should be, empirically say, no more than 4).

4.5.4 Confusion Matrix for the Regrouping Analysis

Therefore, in order to identify letters which have been mis-grouped, a confusion matrix is used to help find letters which have the highest confusion values in groups and regroup them. In neural networks, a confusion matrix is used to evaluate the performance of a classifier during supervised learning. It is a matrix plot of the predicted versus the actual classes of the data. In the confusion matrix for regrouping the mis-grouped letters in MADFUNN1, each row of the matrix is set to represent the instances of a predicted group, while each column represents the instances of an actual letter.

One benefit of this confusion matrix is that it is easy to see if the system is confusing letters in grouping.

4.6.4.1.Rules of Letter Regrouping

After each round of learning, the letters with high confusion values in groups will be regrouped according to the following rule. The regrouping method has two stages; in the first stage the following rule is applied:

N_ω(X): the number (N) of correctly grouped patterns in group X where letter ω is currently assigned to (for the following case, e.g. N_B (1) = 479).
N_ω(Y): the number (N) of mis-grouped patterns in group Y for letter ω (for the following case, e.g. N_B (2) =16, N_B (3) =81... ...N_B (13) =1).
N(Z): the number(N) of letters which have been assigned to group Z. e.g. if AB were assigned to group1 then, N(1) = 2.
***Universal set** U =G_1 (group1), G_1(group2)... ... G_{12} (group12), G_{13} (group13).*

*The **relative complement** of group G in U is denoted by G^C.*

$\forall$: any

N_{letter}: number of letters = 26

N_{group}: number of groups = 13

For example: lettter B is assigned to Group 1 initially. After one round of learning, the correctly classified patterns in Group 1 for B is N_B (1) = 479. And the mis-classified patterns in Group 2 for B is N_B (2) = 16, in Group 3 is N_B (3) = 81, in Group 4 is N_B (4) = 32, in Group 5 is N_B (5) = 5, in Group 6 is N_B (6) = 10, in Group 7 is N_B (7) = 13, in Group 8 is N_B (8) = 18, in Group 9 is N_B (9) = 71, in Group 10 is N_B (10) = 14, in Group 11 is N_B (11) = 3, in Group 12 is N_B (12) = 23, and in Group 13 is N_B (13) = 2.

The relative component of group 2 is $G_2{}^C$ = {G_1, G_3, G_4, G_5, G_6, G_7, G_8, G_9, G_{10}, G_{11}, G_{12}, G_{13}}

Rules (R1):

R1.1: if ($N_\omega(\forall Y) > 1/2\ N_\omega(X)$) *// if the Number(N) of misclassified letter ω patterns in group **Y** is more than half of the Number(N) of correctly classified letter ω patterns in group **X**.*

{

R1.2: if ($N_\omega(Y) > N_\omega(\forall Z{:}Z \subset (X \cup Y)^C$ *// if group **Y** contains the largest Number (N) of letter ω patterns among all misclassified groups.*

&& R1.3: ($N(Y) \leq 2 \cdot N_{letter}/N_{group}$ *//AND if there are fewer than 2 · N_{letter}/N_{group} different letters already assigned in group **Y***

{

This letter ω will be regrouped to group Y.

}

}

For each specific letter **ω**, after each round of learning, the rule checks the number of correctly classified patterns and the number of misclassified patterns in a particular group in which the number of misclassified patterns is significant (more than at least half of the number of correctly classified patterns). The rule then confirms if this group contains the largest number of patterns among all misclassified groups and the current number of letters

assigned to this group is no more than the maximum allowed number. If so, the letter will be regrouped to the new group. Examples are given in the following sections.

Rule1.1 looks for a group which has a significant number of misclassified patterns. The amount to judge for such group is set to "1/2 of the correctly classified patterns" in Rule1.1. This amount does not have to be 1/2, and theoretically it can be any value less than 1. However, this amount does require a significant number of patterns to help the learning. If the value is too small, the rule will pass through too many qualified groups, hence too many directions are given to the network. Whereas if the value is too big, a qualified group will be difficult to find. "1/2" is an experimentally selected reasonable number of value being used in this rule. Rule 1.3 checks whether the selected group (with large number of misclassified patterns) has already contained too many letters. The average number assigned in each group initially is 2. In order to classify letters with similar features together, the rule allows more letters to be added into one group. However, to avoid too many letters from going into one group, a maximum number of letters allowed in each group is set to 4 in this rule. It is experimentally optimised and selected in Rule 1.3.

4.6.4.2.Letter Classification

Initially the groups are assigned without human judgment into thirteen groups. When error rate becomes steady, the system generates the following confusion matrix in figure 53. Taking letter *H* as an example, $(N_H(7) = 161) > (1/2 \bullet (N_H(4) = 295))$ which satisfies *R1.1*. And $N_H(7)$ contains the largest number of mis-grouped H letter patterns which satisfies *R1.2*. $(N(7) = 2) \leq (2\ N_{letter}/N_{group}) = (2 \bullet 26/13 = 4)$ which satisfies *R1.3*. Thus letter *H* should be regrouped to group 7 from group 4.

For each epoch in this stage, each input training pattern is randomly selected from the whole 20,000 patterns, e.g.: (2,8,3,5,1,8,13,0,6,6,10,8,0,8,0,8) is a letter *T* pattern. It should go to group10. If correctly classified in PART1, pass this pattern to Group10.

In this round (as shown in figure 53), e.g., for letter *S*, the system predicted that group 3 mis-grouped 167 patterns which is than half the number of correctly classified patterns in

group 10 (323) where *S* is currently assigned. This satisfies *R1.1*. Group 3 is the largest misclassified group and it has only 2 letters, which satisfies *R1.2* and R1.3. Hence letter S is regrouped from group 10 to group 7. For letter *X*: group13 mis-grouped 143 patterns which is more than at least half number of correctly classified patterns in group12 (254) where *X* was assigned, this satisfies *R1.1*. Group 13 is the largest mis-classified group and it has 2 letters now, which satisfies *R1.2* and *R1.3*. Letter *X* is therefore regrouped from group 12 to group 13.

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13
A	692	8	3	11	3	20	12	0	17	10	0	10	3
B	479	16	81	32	5	10	13	18	71	14	3	23	1
C	2	485	57	53	4	54	3	19	14	8	17	17	3
D	56	496	37	32	14	12	23	45	13	15	5	56	1
E	20	21	409	66	3	62	6	10	54	18	0	45	54
F	20	8	508	9	9	5	5	87	13	50	10	27	24
G	41	23	42	415	1	20	10	36	124	5	9	42	5
H	24	45	46	295	2	50	161	46	33	5	9	16	2
I	20	3	16	9	611	10	0	11	19	14	2	13	27
J	11	4	8	5	624	3	9	9	37	4	1	21	11
K	15	9	60	121	2	396	17	3	62	3	7	39	5
L	24	3	36	19	5	632	2	0	19	0	0	12	9
M	8	1	13	5	5	7	731	4	1	5	2	10	0
N	6	15	11	14	8	17	662	19	5	5	7	13	1
O	15	55	13	100	4	12	6	404	54	13	19	58	0
P	11	12	61	14	5	2	2	666	5	4	1	14	6
Q	31	14	15	45	1	9	3	28	576	11	2	38	10
R	74	11	41	25	9	20	29	7	526	3	2	9	2
S	64	9	167	31	18	18	0	20	32	323	2	39	25
T	5	5	54	27	7	6	0	7	4	616	15	13	37
U	1	15	10	51	14	20	34	7	20	5	612	20	4
V	10	7	28	10	2	3	0	23	17	19	616	15	14
W	5	3	3	16	0	1	50	12	9	1	6	646	0
X	72	11	84	26	6	103	0	7	49	24	8	254	143
Y	3	3	12	7	1	4	9	17	20	47	102	8	553
Z	21	9	29	11	6	11	0	1	31	40	3	11	561

Figure 53: Confusion matrix generated in the first round

Thus letters are regrouped as follows:

Group 1: A B

Group 2: C D

Group 3: E F S

Group 4: G

Group 5: I J

Group 6: K L

Group 7: M N H

Group 8: O P

Group 9: Q R

Group 10: T

Group 11: U V

Group 12: W

Group 13: Y Z X

MADFUNN_1 is then trained to learn the above rearranged groups again and at the same time the other 13 MADFUNNs are adapted to classify letters. After another round of learning, the groups are regrouped as follows according to the rule R1:

Group1: A

Group2: C

Group3: E F S B

Group4: P R

Group5: I J

Group6: K L

Group7: M N H

Group8: O D

Group9: Q G

Group10: T

Group11: U V

Group12: W

Group13: Y Z X

Following another round of learning (third round), the confusion matrix is generated in figure 54:

	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13
A	668	0	27	2	4	10	24	4	3	1	13	5	28
B	0	1	670	48	1	3	9	16	4	1	0	1	12
C	0	493	79	4	4	51	15	20	38	2	26	1	3
D	2	2	101	30	2	19	42	581	0	0	3	0	23
E	0	15	568	9	8	23	6	3	29	3	6	1	97
F	1	9	548	72	13	6	7	10	3	18	6	1	81
G	1	15	141	7	0	37	15	35	498	2	6	0	16
H	2	3	129	23	0	24	382	93	23	0	24	0	31
I	0	0	50	10	615	14	2	6	5	1	0	0	52
J	1	2	45	8	619	0	15	29	2	1	3	0	22
K	0	9	72	27	1	477	38	5	15	1	26	0	68
L	4	2	53	11	6	605	2	4	28	0	7	1	38
M	4	0	22	2	1	4	745	1	3	1	3	3	3
N	1	5	12	0	4	8	687	30	1	1	12	3	19
O	0	0	25	17	0	7	58	555	62	0	16	6	7
P	0	3	147	546	13	5	3	29	13	1	1	10	32
Q	3	0	99	5	12	31	7	47	560	0	7	0	12
R	0	1	151	460	4	46	57	18	4	1	0	0	16
S	2	0	505	10	24	25	8	7	13	3	5	2	144
T	0	1	57	3	5	15	6	9	14	615	9	2	60
U	0	3	17	0	2	18	79	27	7	0	647	1	12
V	0	3	53	11	1	7	8	4	2	9	697	34	35
W	1	0	21	5	0	6	65	0	5	1	16	625	7
X	2	0	110	12	9	24	4	7	13	4	8	0	594
Y	0	0	40	1	6	9	6	5	24	18	86	1	590
Z	0	6	124	7	5	4	3	1	2	1	2	0	579

Figure 54: Confusion matrix generated in the third round

4.6.4.3. *One-shot Multi-grouping and Rules*

As shown in figure 54, no more regrouping is required based on the regrouping rules. Essentially, this means that the groupings are now stable. Thus regrouping stage 2 with a one-shot multi-grouping is introduced. After classification of the groups and letters, ideally, all letters should have mostly been classified to their corresponding groups. But actually there are still some misclassified patterns for each letter. So to follow stage1, a one-shot

multi-grouping is performed. This allows letters to reside in more than one group where necessary. As theoretically, letters do share common features with other letters, especially when patterns for each letter are generated with very distinct features.

Multi-grouping Rule (R2):

R2.1: if (($N_\omega(\forall A) > 10\% N_\omega(X)$) && R1.2: ($N(A) \leq 3 \cdot N_{letter}/N_{group}$) *// if the Number (N) of misclassified letter ω patterns in group A is more than 10% of correctly classified patterns in group(X) AND if the number of letters in group A are less than 6*

{

This letter will be multi-grouped to both group A and X.

}

The maximum allowed number of letters in each group has been increased to 6 ($3 \cdot N_{letter}/N_{group}$) from 4 ($2 \cdot N_{letter}/N_{group}$) in this stage. This is to increase the number of different letters which can be catergorised into each group. Therefore the whole network's output number in PART2 will be increased from 4 to 6, as mentioned in the beginning of this section. According to R2, letters are regrouped as follows:

Group1: A

Group2: C_1

Group3: B E_1 F_1 P_1 R_1 S_1

Group4: F_2 P_2 R_2

Group5: I J

Group6: C_2 K L R_3

Group7: H_1 M N O_1 R_4 U_1

Group8: D H_2 O_2

Group9: G O_3 Q

Group10: T

Group11: U_2 V Y_1

Group12: W

Group13: E_2 F_3 S_2 X Y_2 Z

4.5.5 Simulation Result

As described in figure 50, there are 14 MADFUNNs in this system. One is used to do the grouping and the others are for letter classification. The only difference between MADFUNN_1 and the other MADFUNNs is the number of outputs. MADFUNN_1 has 13 and MADFUNN_2 – MADFUNN_13 has 4 in stage 1 and 6 in stage 2. They are all adapted according to the general learning rule. For each MADFUNN, weights are initialised to small random numbers which are between [-0.1, 0.1] for output neurons and [-0.4, 0.4] for hidden neurons. F-points are initialised to 0.5. A slope limiter is also applied to weight adaptation in order to ensure stability.

There are still two learning constants for each MADFUNN, WL and FL. WL depends on input data range and the F-point interval (0.1 in this case), whereas FL depends solely on the required output range. However, MADFUNN_1 has a more difficult (large amount of data to learn and 13 classes to be classified) problem domain than the other MADFUNNs (only data to letters assigned to each group and 4 to 6 classes to be classified). Therefore, a smaller FL is required for MADFUNN_1 than the others to enable sufficient learning. The learning rate WL is equal to 0.00001 for both MADFUNN_1 and other MADFUNNs, FL is 0.005 for MADFUNN_1 and 0.05 for other MADFUNNs. These training patterns' $\sum aw$ have a known range [-10, 10] for output neurons and [-50, 50] for hidden neurons. They have the precision of 0.02 and 0.1 respectively, so 1001 points are sufficient to encode all training patterns for output and hidden neurons.

The training takes about 500 epochs before the overall error rate reduces to a steady state. Some representative classes' learned functions are listed in figure 55, 56, 57 and 58 as an illustration.

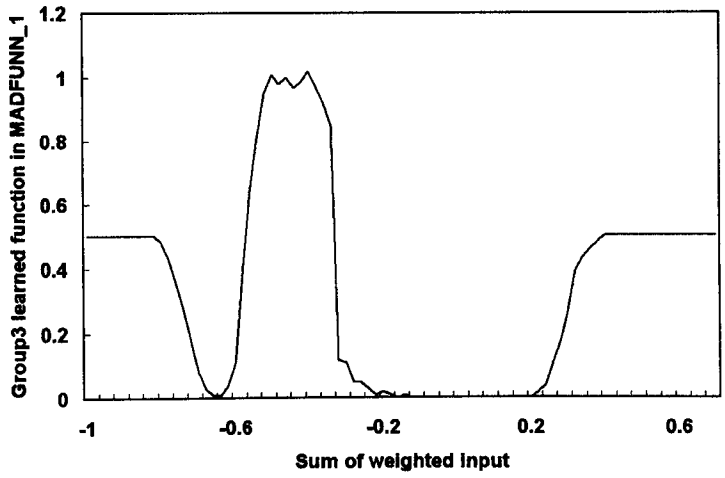


Figure 55: Group 3 learned function in MADFUNN_1

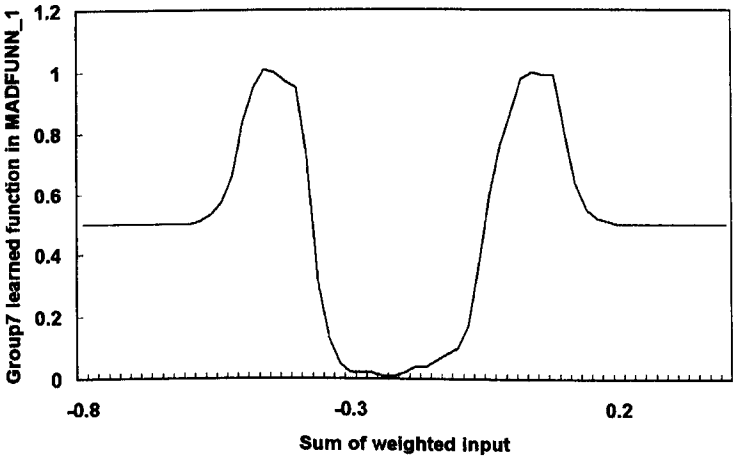


Figure 56: Group 7 learned function in MADFUNN_1

In figure 55 (as well as in 56, 57, and 58), the raised and decreased curves mark the learned region, within which adaptation has occurred. As showed in figure 55, the sum of weighted inputs ($\sum aw$) in the range [-0.54, -0.34] for group 3 function in MADFUNN_1 activates a group 3 response. Similarly in figure 56, there are two learned parts where the $\sum aw$ in the range [-0.5, -0.38] and [0, 0.1] for group 7 function in MADFUNN_1 activate a group 7 output.

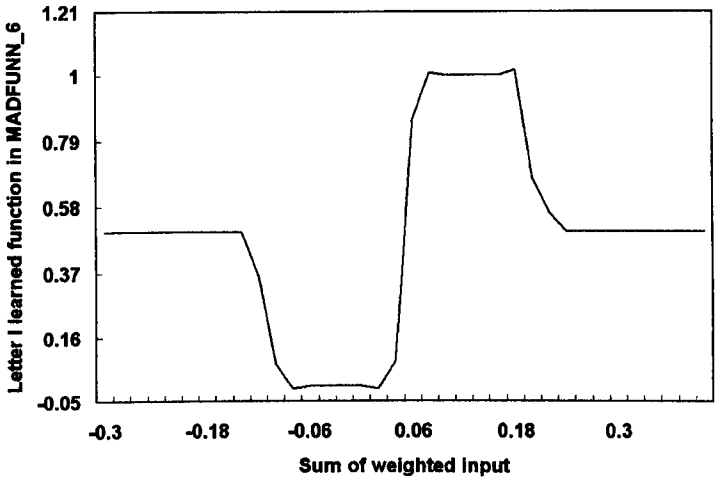


Figure 57: Letter I learned function in MADFUNN_6

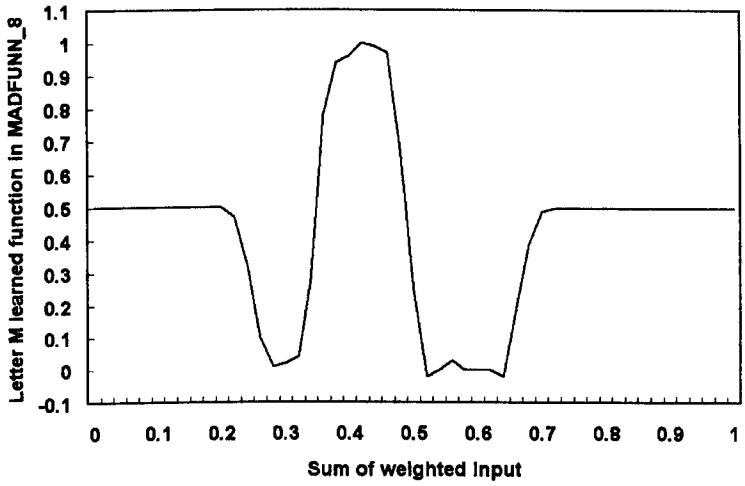


Figure 58: Letter M learned function in MADFUNN_8

Figure 57 and 58 are learned function curves for letter classification. Figure 57 is for letter I in group 5(MADFUNN_6) and figure 58 is for letter M in group 7(MADFUNN_8).

4.5.6 Generalisation Ability

The overall generalisation is the accuracy of correctly classified letters which come from the correctly classified groups, in other words, it equals generalisation of PART_1 (group classification) times generalisation of PART_2 (letter classification).

The performance of this system is tested both by the independent testing data (pure test data) and non-independent testing data (natural test data). Natural test data is all testing patterns including any that already participated in training whereas for the pure test data they are never used in training. With 4,000 pure test data, 87.60% generalisation can be achieved compared to only 79.3% [156] generalisation using a simple back-propagation MLP. And 93.77% accuracy can be achieved if testing with the 4,000 natural test data.

Table 11 Comparison of MADFUNN with MLP on the Letter Image Recognition

Generalisation	MADFUNN	MLP
4000 test patterns	87.6%	79.3%

4.5.7 Performance Comparison with other Methods Applied to this Problem

Using a Holland-style adaptive classifier and a training set of 16,000 examples, the authors of this dataset reported the classifier accuracy [154] is a little over 80% (actually, 80.8%, 81.6% and 82.7% are the three best) based on the independent testing data (compared to 87.60% correct classification by MADFUNN).

Philip and Joseph [157] introduced a Difference Boosting (DB) less intensive Bayesian Classifier algorithm. Using the first 16,000 patterns from the dataset as the training set resulted in 85.8% accuracy (compared to 87.60% by MADFUNN) on the independent test set of remaining 4,000 patterns. They can get 94.1% accuracy (compared to 93.77% by MADFUNN) on the entire 20,000 data. However, this can only be achieved by repeated reordering of the training and test sets (this is to say, the training and test sets are selected). Also, a second guess is allowed on the test set if the first prediction fails. This is not the case with the MADFUNN tests.

Partridge and Yates [158] used a selection procedure known as pick heuristic with three different measures: CFD (coincident-failing diversity measure), DFD (distinct-failure diversity measure) and OD (overall diversity which is the geometric mean of CFD times DFD) as three multi-version systems, pickOD, pickCFD and pickDFD. By exploiting distinct-failure diversity in these three ways on the 4,000 test patterns, they achieved 91.6% generalisation using pickCFD, 91.07% generalisation using pickDFD and 91.55% using pickOD. However, implementation of these three multi-version systems is complex with each containing nine networks. For instance, the pickDFD system was composed of 4 RBF networks and 5 MLPs.

With AdaBoost on the C4.5 algorithm [156], 96.9% correct classification can be achieved on the 4,000 test data. However computational power is required over 100 machines to generate the tree structure [156]. And with a 4 layer fully connected MLP of 16-70-50-26 topology [153], 98% correct classification with AdaBoost can be achieved but required 20 machines to implement the system. In contrast, all the MADFUNN tests have been carried out on one PC.

4.6 CONCLUSION

In this chapter, a multi-layer ADFUNN is deployed in order to solve more complex nonlinear models. The structure of this multi-layer ADFUNN is like the back-propagation algorithm but with activation function adaptable. The activations are propagated from the input to the output layer through a hidden layer, and the error between the actual value and the desired nominal value in the output layer is propagated backwards in order to modify the weights and adjust the activation functions.

The reason to introduce MADFUNN is to overcome the limitations that ADFUNN faces in some large highly non-linear complex datasets to establish whether the hidden layer supports better learning. Even so ADFUNN has exhibited extraordinary performance as compared to other single layer and many multi-layer neural network methods by combing two modes of learning into one network.

The structure of MADFUNN -- a multi-layer ADFUNN, is similar to the structure of a Multi-layer perceptron, but with only one layer of hidden neurons. Just like ADFUNN, it

combines the mode of weight adaptation and the mode of function adaptation within one network. The only difference between ADFUNN and MADFUNN is the adaptation of hidden neurons because ADFUNN does not have a hidden layer. The adaptation in the hidden layer of MADFUNN has the same objective to reduce output errors based on the Delta Rule, by distributing the error of each output neuron to all the hidden neurons that is connected to it. The error in a hidden neuron is therefore the weighted sum of all output errors which have connections to it. Both weights and functions for hidden neurons are adapted to reduce these errors.

In this chapter, a dataset contains a large size of patterns, from the UCI repository is investigated for MADFUNN. This letter image recognition task has 20,000 different examples collected from a large number of human subjects. The performance of MADFUNN is significant compared to other methods.

The experiment has not only shown that MADFUNN is very efficient in extracting common features from letter examples, but also shown that MADFUNN is very powerful in classifying each featured group of letters into their correct classes.

To achieve more accurate feature detecting result, a regrouping rule and a one-shot very last regrouping rule are introduced based on the confusion matrix obtained during learning. The aim of these rules is to categorise the letter to a group in which other letters all share similar features. The categorised letters are then very easy to be classified.

CHAPTER 5 SNAP-DRIFT ADAPTIVE FUCTION NEURAL NETWORK (SADFUNN)

5.1 INTRODUCTION

Another modal learning method S*nap-drift* was first introduced by Palmer-Brown and Lee [159, 160, and 161] as: "an attempt to simplify and modify Adaptive Resonance Theory (ART) learning in non-stationary environments where self-organisation needs to take account of periodic or occasional performance feedback"[160, 161]. Since then, the snap-drift algorithm has been applied in many different fields and applications and proved extremely valuable for continuous on-line learning. The reason to combine Snap-drift with ADFUNN into a one neural network echoes the classic model of feature extraction followed by classification [155]. Snap-drift is very effective at finding appropriate features that, for example, Learning Vector Quantization (LVQ) [162] cannot find and ADFUNN is highly efficient single layer neural network in pattern classification. The combination provides a general method suitable for feature extraction followed by classification.

The snap-drift learning method utilises a mode of fast, minimalist learning (snap) and a mode of slower drift learning. Snap is based on the logical intersection method from ART and is implemented as a fuzzy AND; and drift is based on LVQ. It enhances the strengths of the two modes of learning in a rapid form of adaptation that balances minimalist pattern intersection learning with LVQ.

Snap-drift has been effectively applied on a range of databases [159, 160 and 161] and proved to be a very fast unsupervised method suitable for real-time learning and non-stationary environments where new patterns are continually presented to the network. It is also very effective in extracting distinct features from the complex cursive-letter datasets. The combining of one modal learning Snap-drift with another modal learning single layer

supervised ADFUNN (i.e. SADFUNN) offers effective and simple learning strategies and therefore produce a powerful supervised method.

5.2 THE SNAP-DRIFT ALGORITHM

Snap-Drift system architecture is shown in figure 59. The first layer, a distributed snap-drift neural network (dSDNN) learns to group the input patterns according to their features. In this case, 10 F1 nodes whose weight prototypes best match the current input pattern, are used as the input data to a selection snap-drift neural network (sSDNN) module for feature classification. In the dSDNN module, the output nodes with the highest net input are accepted as winners (only 3 winners shown in figure 59). In the sSDNN module, a 'quality assurance threshold' is used. When the net input of a sSDNN node is greater than the threshold, the output node is accepted as strong enough to be the winner; otherwise another uncommitted output node will be selected as the new winner and initialised with the current input pattern. In general, the snap-drift algorithm can be described as: $w = \alpha(snap) + \sigma(drift)$, where α and σ are toggled between (0, 1) and (1, 0) at the end of each epoch. The overall effect is to perform two complementary forms of feature discovery within one system.

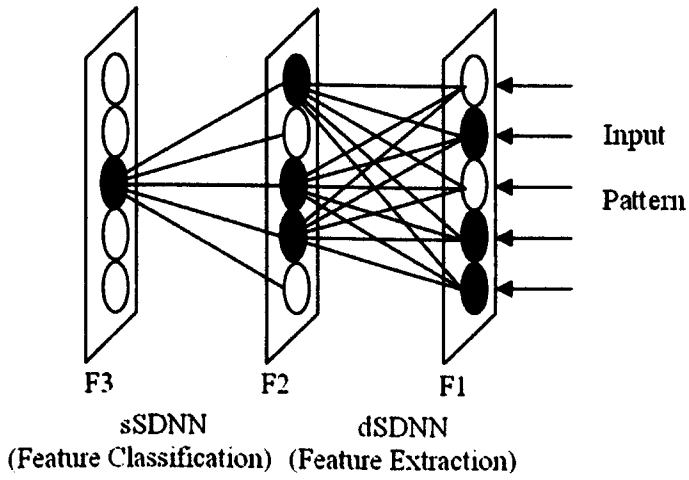


Figure 59: Snap-Drift Neural Network (SDNN) architecture

5.2.1 Weights Initialisation

The weights are initialised to small random numbers in snap-drift in the beginning of simulations. Top-down weights, w_{ji} are initialised using a randomly selected input pattern data.

The bottom-up weights w_{ij} are initialised corresponding to the initial values of the top-down weights w_{ji}. They are:

$$w_{ij}(0) = \frac{w_{ji}(0)}{1+N}$$

where N = number of input nodes

5.2.2 Distributed Snap-Drift Neural Network (dSDNN) for Feature Extraction

The purpose of the distributed SDNN (dSDNN) module is to learn and detect key features, which can then be used by the selection Snap-Drift Neural Network (sSDNN) to select an appropriate output. When an input pattern is presented, the network attempts to categorise the input pattern by comparing it against the stored knowledge of the existing distributed output categories of the F2 layer in figure 59. dSDNN is based on the concept of distributed ART (dART) [163], there is more than one winning node, in this case D (the number of winning nodes) = 3. The three F2 nodes with the highest bottom-up activations are selected. Generally speaking, the dSDNN learns to group input patterns according to their features using snap-drift. The neurons whose weight prototypes result in them receiving the highest activations are adapted. Weights are normalised weights so that in effect only the angle of the weight vector is adapted, meaning that a recognised feature is based on a particular ratio of values, rather than absolute values.

5.2.3 The Selection Snap-Drift Neural Network (sSDNN) for Feature Classification

The winning neurons from dSDNN output act as input data to the selection SDNN (sSDNN) module for the purpose of feature grouping and it is also subject to snap-drift learning [159]. The distributed output representations of category, produced by dSDNN acts as inputs to the sSDNN. The architecture of the sSDNN is the same as that described dSDNN but only one final wining node with the highest activation is for learning and classification.

5.2.4 Snap-Drift Learning Rule

The learning process in SDNN is unlike the traditional error minimisation in MLPs and other kinds of networks which optimise the classification by forcing the features in a way that minimises errors, and does not consider the feature's significace within the input data. In contrast, SDNN toggles its learning mode to find a rich set of features in the data and uses them to group the data into categories.

Each weight vector is bounded by snap and drift in which snapping gives the angle of the minimum values and drifting gives the average angle of the patterns grouped under the neuron [78, 79 and 80].

The following is a summary of the steps that occurs in SDNN ([159, 160 and 161]):

Step 1: Initialise parameters: (a = 1, s = 0. a = 1 will invoke a snap learning whereas s = 1 will invoke a drift learning)

Step 2: For every input pattern in every epoch (t)

Step 2.1: Find D (D is the number of) winning nodes at F2 with the largest net inputs

Step 2.2: Weights of dSDNN are adapted according to the alternative learning procedure: (a, s) becomes Inverse (a, s) after every successive epoch, i.e. (0, 1)

Step 3: Process the winning nodes as an input pattern of sSDNN

Step 3.1: Find the node at F3 with the largest net input

Step 3.2: Test the threshold condition:

if (the net input of the node > the threshold)

Weights of the sSDNN output node adapted according to the alternative learning procedure: (a, s) becomes inverse (a, s) after every successive epoch

else An uncommitted sSDNN output node is selected and its weights are adapted according to the alternative learning procedure: (a, s) becomes Inverse (a, s) after every successive epoch.

5.3 THE SYSTEM LEARNING

By combining an unsupervised distributed snap-drift neural network (dSDNN) for feature detection and a supervised ADFUNN for classification, a powerful network SADFUNN is

constructed. SADFUNN combines two modal learning methods, with one plays the role in feature extraction and the other plays the role in pattern recognition. SADFUNN is particularly suitable for relatively complex tasks which contain high dimensional input data many hidden features among the data.

5.3.1 Unsupervised Distributed Snap-Drift Neural Network (dSDNN)

As described in 5.2.2, in SADFUNN, only the distributed snap-drift neural network (dSDNN) is used for the key features detection. Simply speaking, the snap-drift weights adaptation can be described as:

Snap-Drift = α *(Fast_Learning_ART)* + σ *(LVQ)*[159]

The top-down weights are adapted using:

$$w_{Ji}^{(new)} = \alpha\,(I \cap w_{Ji}^{(old)}) + \sigma(w_{Ji}^{(old)} + \beta(I - w_{Ji}^{(old)}))[159]$$

where w_{Ji} is the top-down weights vectors, I is the input vector and β is the drift speed constant. When $\alpha = 1$, a fast minimalist snap learning is invoked, otherwise, a drift learning is activated.

$$w_{Ji}^{(new)} = \alpha\,(I \cap w_{Ji}^{(old)})$$

And when $\sigma = 1$, $w_{Ji}^{(new)} = w_{Ji}^{(old)} + \beta\,(I - w_{Ji}^{(old)})$ which employs a clustering at β speed. Whereas, the bottom-up learning is just a normalised version of the top-down learning.

$$w_{iJ}^{(new)} = w_{Ji}^{(new)} / |\, w_{Ji}^{(new)}|$$

where $w_{iJ}^{(new)}$ is the top-down weights after learning. From the predefined number of winning features (for example 10), 10 winning F2 nodes with the highest bottom-up activations are selected by:

$T_J = max \{T_j \mid j = 1, 2... M)\}$

These 10 F2 nodes will be passed to ADFUNN to perform pattern classification. In SADFUNN, patterns are applied to ADFUNN only after snap-drift has converged. This is because ADFUNN can only optimise once snap-drift has successfully extracted the features.

5.3.2 Supervised ADFUNN

Instead of using a selection snap-drift (sSDNN) for pattern classification, ADFUNN is applied to classify the feature categories. ADFUNN has the same number of dSDNN F2 nodes as the inputs. When patterns pass through the learned dSDNN to ADFUNN, only the inputs corresponding to the dSDNN top winning nodes (say 10 for example) in ADFUNN are activated and calculated. The inputs applied to these 10 input nodes in ADFUNN are the weighted sums from the dSDNN top wining features. The input data for the inactivated inputs are all set to zero. Weights and functions in ADFUNN are adapted in parallel to reduce the output errors. The number of outputs is the number of classes of the dataset.

5.3.3 SADFUNN Architecture

By combining two modal learning methods: an unsupervised snap-drift and a supervised ADFUNN together, SADFUNN is shown in figure 60. As input patterns are introduced at the input layer F1, the distributed SDNN (dSDNN) learns to group them. The winning F2 nodes, whose prototypes best match the current input pattern, are used as the input data to ADFUNN. For each output class neuron in F3, there is a linear piecewise function. Functions and weights are adapted in parallel using a gradient descent supervised learning algorithm. The output error is obtained in each output neuron and the two nearest f-points on the piecewise activation function are adapted separately, using a function modifying version of the delta rule on a proximal-proportional basis, as described in section 3.4.6. Patterns are being applied to ADFUNN after snap-drift has converged.

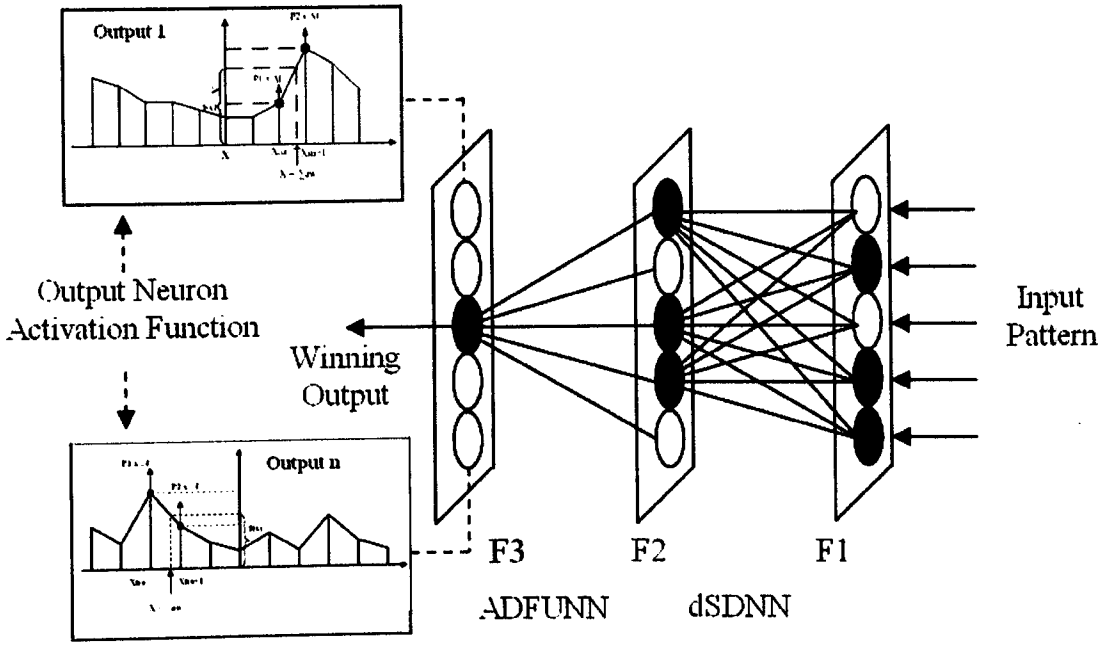


Figure 60: Snap-drift ADFUNN (SADFUNN) Neural Network architecture

5.4 SADFUNN ON OPTICAL AND PEN-BASED HANDWRITTEN DIGIT RECOGNITION

5.4.1 Optical and Pen-Based Handwritten Digit Recognition Datasets

These two complex datasets are those of handwritten digits presented by Alpaydin et.al [164, 165]. They are two different representations of the same handwritten digits. 250 samples per person are collected from 44 people who filled in forms which were then randomly divided into two sets: 30 forms for training and 14 forms by distinct writers for writer-independent testing.

The optical one was generated by using the set of programs available from NIST [166] to extract normalised bitmaps of handwritten digits from a pre-printed form. Its representation is a static image of the pen tip movement that have occurred as in a normal scanned image. It is an 8 x 8 matrix of elements with each element in the range of 0 to 16. Thus the 8 x 8 = 64 input dimensions are needed for this dataset. There are 3823 training patterns and 1797 writer-independent testing patterns in this dataset.

The Pen-Based dataset is a dynamic representation of the movement of the pen as the digit is written on a pressure-sensitive tablet. It is generated by a WACOM PL-100V pressure sensitive tablet with an integrated LCD display and a cordless stylus. The raw data consists of integer values between 0 and 500 at the tablet input box resolution, and they are then normalised to the range 0 to 100. This dataset's representation has eight(x, y) coordinates as shown in the spatial resampling image in dynamic representation in figure 61. The eight coordinates are connected in sequence to form the dynamic digit representation. To represent these eight coordinates, 8 (coordinates) · 2(for x and y) = 16 input dimensions are needed for the dataset. There are 7494 training patterns and 3498 writer-independent testing patterns. The diagram in figure 61 shows how these two datasets are generated from the raw data.

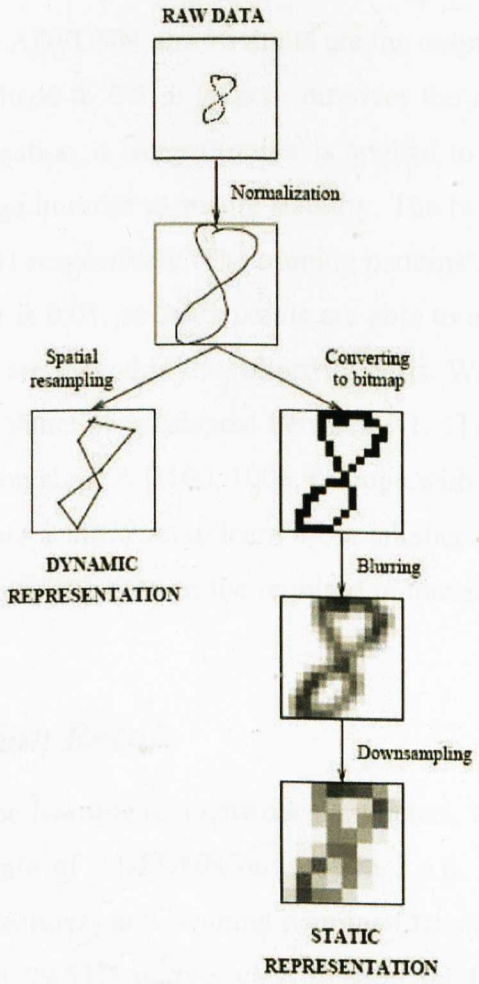


Figure 61: The processing of converting the dynamic (pen-based) and static (optical) representations (image adapted from [164, 165])

5.4.2 SADFUNN on the Two Datasets

In ADFUNN, the activation functions are adapted in parallel with the weights using a function modifying version of delta rule. All the inputs are scaled from the range of {0, 16} to {0, 1} for the optical dataset and from {0, 100} to {0, 1} for the pen-based dataset. Training patterns are passed to the Snap-Drift network for feature extraction. After a couple of epochs (feature extraction learned very fast in this case, although 7494 patterns need to be classified), the learned dSDNN is then ready to supply ADFUNN for pattern recognition. The training patterns are applied to dSDNN again but without learning. Results are calculated using the learned dSDNN weights. Winning F2 nodes (whose prototypes best match the current input pattern) are used to form the input data for ADFUNN.

In this single layer ADFUNN, the 10 digits are the output classes. Weights are initialised to 0. F-points are initialised to 0.5 to help to discover the active learned function range Again instead of normalisation, a weight limiter is applied to ensure that the adaptation to weights will not be too large in order to ensure stability. The two learning rates FL and WL are equal to 0.1 and 0.000001 respectively. The training patterns' $\sum aw_j$ has a known range of [-10, 10]. The precision of it is 0.01, so 2001 points are able to encode all training patterns for output. The learning rates are picked in the following basis. WL mainly depends on input data range and f-point range. Function is adapted between [-1, 1] and the f-point interval is 0.01, hence the range of function slope is [-100, 100]. To cope with weights adaptation involving function slope, and to enable a more acute learning, a smaller learning rate 0.000001 is selected for WL. Whereas FL just depends on the required output range [0, 1] and a normal learning rate can be applied.

5.4.3 Simulation Result

After specifying the learning and network parameters, the network is ready to learn using the general learning rule of ADFUNN outlined in 3.4.6. By varying the number of snap-drift neurons (learned features) and winning neurons (detected features) in F2, within 200 epochs in each run, about 99.53% correct classification for the optical dataset and 99.2% correct classification for the pen-based dataset can be obtained for the training data. The output

neuron functions (as in figure 62 – 65) can be obtained (only a few of the learned functions listed here):

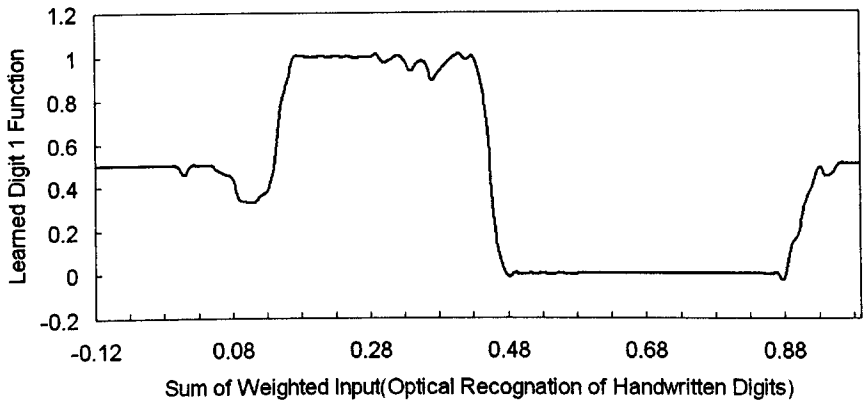


Figure 62: Digit 1 learned function in optical dataset using SADFUNN

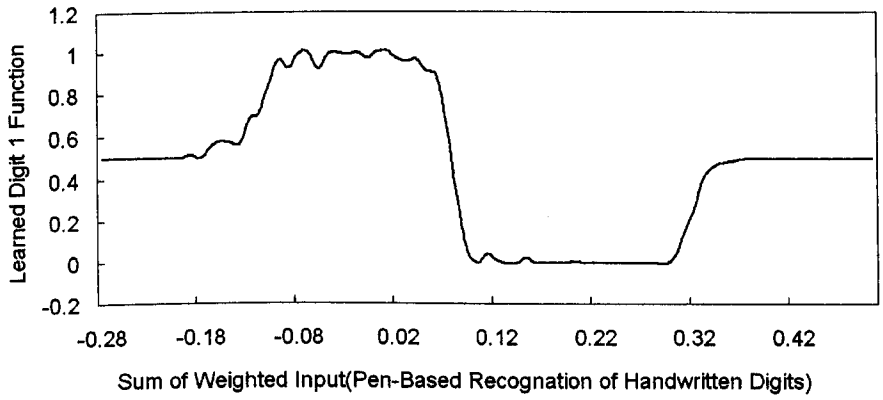


Figure 63: Digit 1 learned function in pen-based dataset using SADFUNN

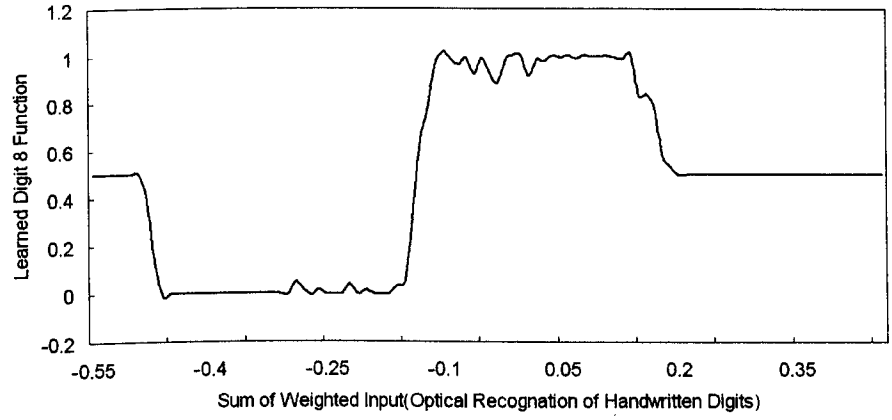


Figure 64: Digit 8 learned function in optical dataset using SADFUNN

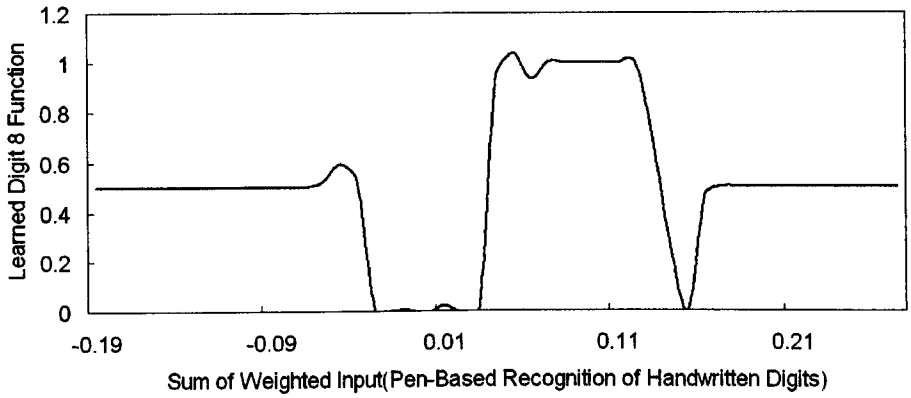


Figure 65: Digit 8 learned function in pen-based dataset using SADFUNN

5.4.4 Generalisation Ability

The learned network is tested using the two writer-independent testing data for both of the optical recognition and pen-based recognition tasks. Performance varies with the varying of a small number of parameters, including learning rates FL, WL, the number of snap-drift neurons (features) and the number of winning features.

Based on the experimental results, a large total number of features has a positive effect on the overall performance, however too many may limit generalisation if there is too much memorisation. The performance charts in figure 66 and 67 show how the generalisation changes along with the total number of features

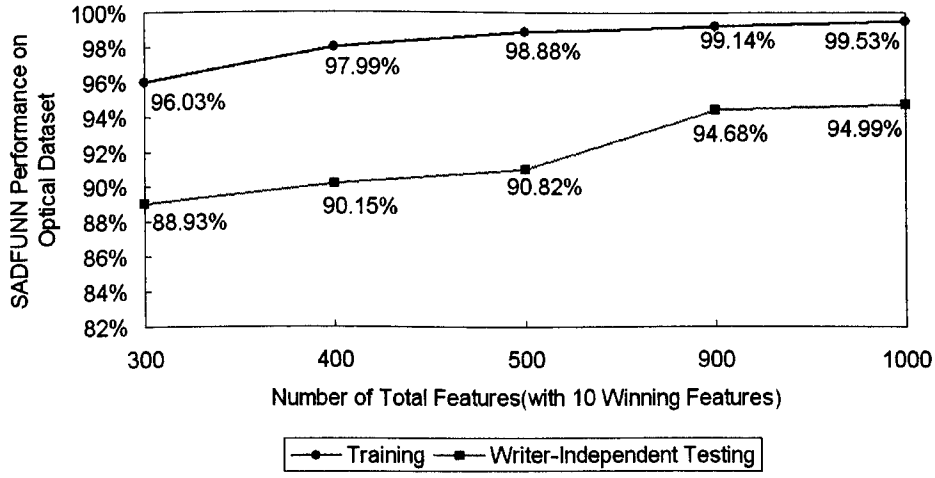


Figure 66: The performance of training and testing for optical dataset using SADFUNN

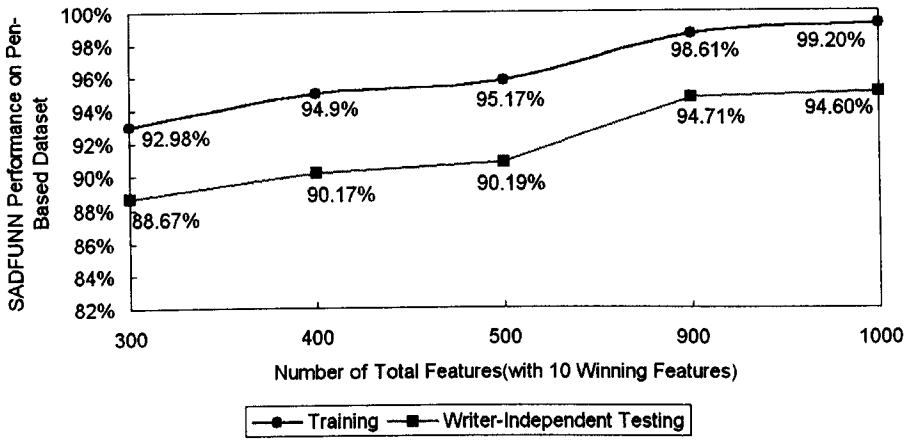


Figure 67: The performance of training and testing for pen-based dataset using SADFUNN

Figure 68 to 70 are examples of some misclassified patterns from SADFUNN for optical recognition case.

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Actual output	0.0	0.0	0.0	0.11	0.0	**0.82**	0.0	0.0	0.0	**0.35**
Expected output	0	0	0	0	0	0	0	0	0	1

Figure 68: Digit 9 misclassified to digit 5

As can be seen from above figure, a digit 9 pattern was misclassified to digit 5 which has the largest output. In this misclassified example, the upper part of digit 5 must have been almost looped, making the 5 similar to a 9.

The following figure 69 and 70 are another two cases which illustrate similar confusions, where digit 2 is misclassified to 8 and digit 3 is misclassified to 9.

Digit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Actual output	0.0	0.0	**0.49**	0.0	0.0	0.0	0.0	0.12	**0.52**	0.1
Expected output	0	0	1	0	0	0	0	0	0	0

Figure 69: Digit 2 misclassified to digit 8

Digit	0	1	2	3	4	5	6	7	8	9
Actual output	0.0	0.0	0.0	**0.25**	0.0	0.0	0.0	0.1	0.0	**0.82**
Expected output	0	0	0	1	0	0	0	0	0	0

Figure 70: Digit 3 misclassified to digit 9

5.4.5 Related Work on Methods of Handwritten Cursive Letter Recognition

Handwritten letter recognition task has a long history and always been a challenging problem encountered in many real-world applications, such as postal mail sorting, bank cheque recognition, and automatic data entry from business forms. A good computational solution must have the ability to recognise complex cursive letter patterns and represent commonsense knowledge of them.

Handwritten cursive letter recognition problem is made complex by the fact that the writing is fundamentally ambiguous as the strokes in the letter are generally linked together. The recognition techniques can be classified according to two criteria: the way preprocessing is performed on the data and the type of the decision algorithm.

Artificial neural network can finely approximate complicated decision boundaries. It is one of the most popular methods which has been applied to this area as far back as to the 1950's. Letter recognition was included in Frank Rosenblatt's [167] perceptron. It was one of the first computers based on the idea of a neural network, which is a simplified computational model of neurons in a human brain. It was the first functioning neurocomputer, and it was able to recognise a fixed-font character set. Since then, many neural network learning methods like (Multi Layer Perceptron)MLP, k Nearest Neighbours (kNNs), RBF etc have been applied to this area and the difficulty of handwriting recognition has been always underestimated.

Zhang and Li [168] propose an adaptive nonlinear auto-associative modelling (ANAM) based on Locally Linear Embedding (LLE) for learning both intrinsic principal features of each concept separately. The LLE algorithm is a modified k-NN developed to preserve local

neighbourhood relation of data in both the embedded Euclidean space and the intrinsic one. In ANAMs, training samples are projected into the corresponding subspaces. Based on the evaluation of recognition criteria on a validation set, the parameters of inverse mapping matrices of each ANAM are obtained adaptively. The forward mapping matrices are calculated based on a similar framework. 1.28% and 4.26% error rates can be obtained by ANAM for optical recognition and pen-based recognition respectively. However, given its complex calculation of forward mapping and inverse mapping matrices, many subspaces are needed and also suboptimal auto-associate models need to be generated.

A feed forward neural network trained by Quickprop algorithm, which is a variation of error back propagation is used for on-line recognition of handwritten alphanumeric characters by Chakraborty [169]. Some distorted samples in numerals 0 to 9 are used as experiment which are different from the dataset used in this section. Good generalisation capability of the extracted feature set is reported.

SADFUNN is computationally much more efficient, simpler and achieves similar results. It will be a straight forward process to apply it to many other domains.

5.4.6 Comparison with other Methods Applied to the Datasets

Using multistage classifiers involving a combination of a rule-learner MLP with an exception-learner k-NN, Alpaydin et al. [83 and 84] reported 94.25% and 95.26% accuracy on the writer-independent testing data for optical recognition and pen-based recognition datasets respectively. Patterns are passed to a MLP with 20 hidden layers, and all the rejected patterns are passed to a k-nearest neighbours with k = 9 for a second phase of learning.

For the optical recognition task, 23% of the writer-independent test data were not classified by the MLP. They were passed to k-NN to give a second classification. In the single network combination of a single layer Snap-Drift and a single layer ADFUNN (SADFUNN) only 5.01% patterns were not classified on the testing data. SADFUNN proves to be a highly effective network with fast feature extraction and pattern recognition ability. Similarly, with the

pen-based recognition task, 30% of the writer-independent test data are rejected by the MLP, whereas only 5.4% of these patterns were misclassified by SADFUNN.

The original intention of Alpaydin et al. [83 and 84] was to combine multiple representations (dynamic pen-based recognition data and static optical recognition data) of a handwritten digit to increase classification accuracy without increasing the system's complexity and recognition time. By combing the two datasets, they get 98.3% accuracy on the writer-independent testing data. However, this combined dataset was not tested using SADFUNN as Apaydin et al. [83 and 84] have already proved the combination of multiple presentations work better than a single one, because SADFUNN has already exhibited extremely high generalisation ability compared to a MLP.

5.5 CONCLUSION

In this chapter, a single layer modal learning supervised adaptive activation function neural network learning method is combined with another single layer modal learning unsupervised distributed snap-drift neural network learning method, to perform pattern classification and feature extraction respectively.

The combination of these two highly effective modal learning methods proved to be very powerful in extracting and detecting features, and being accurate in classification. This was shown for a complex large dataset. When compared with other work on the same data it was shown that SADFUNN is rather effective in terms of: very fast training time (a couple of epochs for dSDNN to perform feature extraction and no more than 100 epochs for ADFUNN to perform classification), higher performance and simpler network structure. In addition, for extremely complex or special data, it is possible to combine the unsupervised Snap-Drift learning method with the Multi-layer ADFUNN which integrates more neurons to the network to learn the problem.

CHAPTER 6 ADAPTIVE FUNCTION NEURAL NETWORKS FOR INTELLIGENT DATA ANALYSIS

6.1 INTRODUCTION

Intelligent data analysis with neural networks requires analysis of the weights to establish the most important factors and generate simplified equations to explain network decisions [117]. It is to discover the information contained in the data and helps finding intelligent solutions and new ways of looking at problems. There are software tools for intelligent data analysis which combine statistical methods with neural networks and fuzzy technologies. However, these tools mostly depend on the statistical methods to analyse weights and retrieve information. The activation function itself is not used to inform the analysis. However, the learned activation function curves in ADFUNN (MADFUNN and even SADFUNN) can reveal much more useful information about the data, especially when considering and analysing the learned weights and functions together. In this chapter, the learned weights and functions from different applications of ADFUNN are analysed.

Some learned ADFUNN (as well as in MADFUNN and SADFUNN) functions contain discontinuities (steps) in the activation function curves. In fact, many learning algorithms have been investigated for neural networks with discontinuous activation functions [170, 171]. However, these works all employed pre-defined shapes of the discontinuous activation functions to better suit the problems rather than let the network produce a meaningful curve. In contrast, discontinuous activation functions produced by the learned ADFUNN contain very important information about data, e.g. class boundaries. In order to remove curve noise, analyse the useful information and yet keep the important discontinuous characteristic of the learned activation functions, a self-sizing moving window is introduced. The moving window is used to filter the learned function curve in order to keep useful sharp edges (discontinues) and remove the unrelated information (noise). It is compared with some well known

smoothing methods, like simple moving average and the least square polynomial smoothing method.

This section looks at intelligent data analysis for different datasets learned by ADFUNN, and conducted by analysing the learned functions and weights together, as well as a complementary self-sizing smoothing method introduced to assist with removing noise, thereby helping to reveal important information from learned function curves.

6.2 Learned Functions and Weights Analysis

6.2.1 Retrieve Important Inputs Variables and Features for Each Class from the Learned Weights

Every neural network possesses knowledge which is contained in the values of the connections weights. This is to say the effect that each input has in decision making is dependent on the weight of that particular input. A larger learned weight for a specific input neuron indicates this input neuron is more important than other input neurons to in predicting a specific output neuron.

Because the learned weights contain very important information on the data. By analysing the learned weights, the relationships between input and output data are revealed, identifying which inputs are decisive for a particular output.

6.2.2 Retrieve Important Information for Each Class from the Learned Functions and Weights

The piecewise linear activation functions in ADFUNN are adapted either towards 1(true) or 0 (false). The learned curve of the activation function for an output class will have an obvious learned range, to indicate which parts of the learned curve will active this class and which parts will not. Combining these learned functions with the learned weights for each class, it is then very easy to generate some inequality rules as a learned solution for the data.

These inequality rules can be rather efficient and accurate if the input dimension is low (as the example for Iris dataset in the following). For medium or high dimensional inputs, the inequality rules can still be calculated easily and quickly by software programs. These rules are generic and easy to apply. One generated rule for the Iris dataset is introduced in section 6.2.4. Iris dataset has a 4 inputs dimension and the rule can be easily applied.

6.2.3 Analysis Result for Iris Dataset

As described in section 3.5.2, the current statistical analysis for Iris dataset is limited to plotting the dataset onto scatter plots to determine patterns in the data in relation to the Iris classifications However, from the learned ADFUNN, Iris data can be interpreted by considering weights and functions together. The following are examples of learned weights and functions in ADFUNN for Iris dataset.

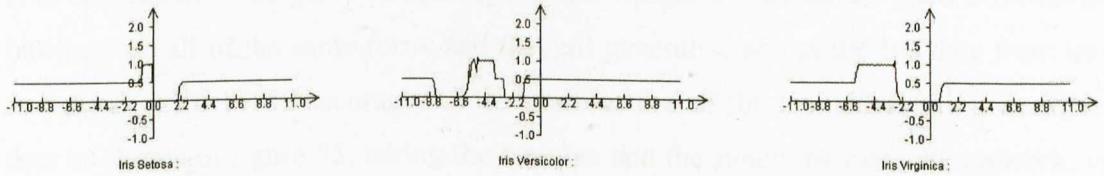


Figure 71: One example of learned functions of ADFUNN on the Iris dataset

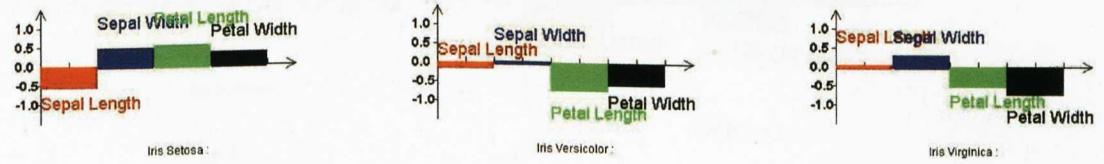


Figure 72: The learned weights with corresponding learned ADFUNN functions in figure 71

The following is another example of the learned functions and weights.

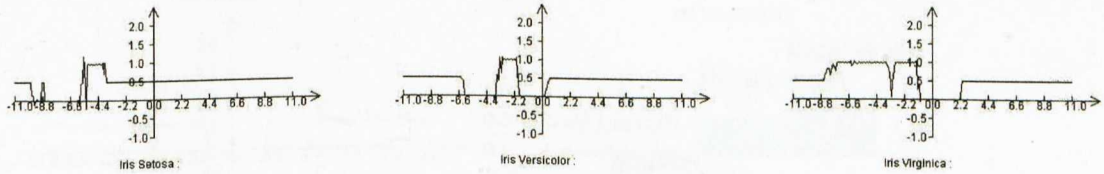


Figure 73: Another example of learned functions of ADFUNN on the Iris dataset

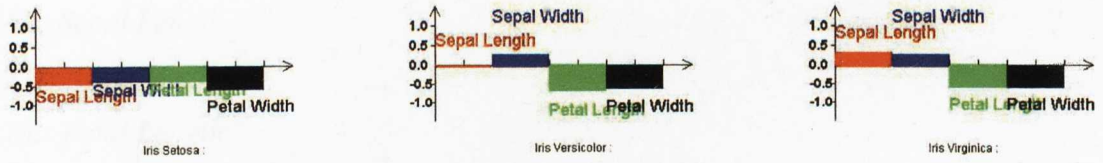


Figure 74: The learned weights with corresponding learned ADFUNN functions in figure 73

By considering and analysing a number of examples like those above, a rule can be summarised. To identify Iris Setosa, sepal length, sepal width, petal width and petal length, all must be taken into account as they have similar weights. In contrast, Iris Versicolor and Iris Virginica are most dependent on petal length and petal width.

6.2.4 Inequality Rule for Iris Dataset

It is obvious that in figures 71 to 74, different weight initialisations yield different solutions, but they are all of the same form, and they all generalise across the Iris data from training set to test set, with 100% accuracy. Close examination of the learned solution example for Iris data as shown in figure 75, taking the weights and the functions from the network, yields the following inequality rules for the three types of Iris.

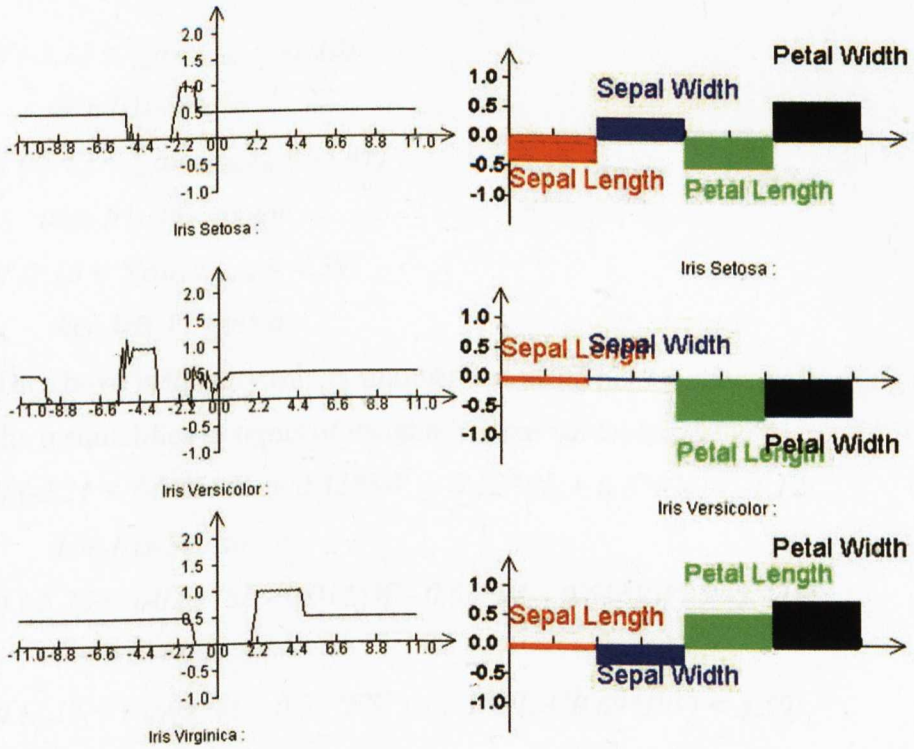


Figure 75: The learned activation functions and weights for Iris dataset using ADFUNN

SL: Sepal Length

SW: Sepal Width

PL: Petal Length

PW: Petal Width

w_{setosa} = *(-0.45, 0.31, -0.52, 0.6)*

$w_{versicolor}$ = *(-0.09, -0.01, -0.63, -0.61)*

$w_{virginica}$ = *(-0.09, -0.32, 0.51, 0.69)*

a = *(SL, SW, PL, PW)*

$\sum aw_{setosa} = a \bullet w_{setosa}$

$\sum aw_{versicolor} = a \bullet w_{versicolor}$

$\sum aw_{virginica} = a \bullet w_{virginica}$

Taking any example (SL, SW, PL, PW) from Iris dataset, the inequality rule can be summarised as follows, in which the three statements are equally conditioned without any particular priority in order. If the ∑aw satisfies any statement, the corresponding class is selected and the execution of the inequality rule can terminate.

if $(-2.21 < \sum aw_{setosa} < -1.12)$

then Iris-Setosa

if $(-5.72 < \sum aw_{versicolor} < -3.41)$

then Iris-Versicolor

if $(2.18 < \sum aw_{virginica} < 4.59)$

then Iris-Virginica

The above inequality rule is summarised using neural network learning algorithm. Expressing the inequialities in terms of weights * *input variables*:

*if (-2.21 < (-0.45*SL + 0.31*SW _- 0.52*PL + 0.6*PW)< -1.12)*

then Iris-Setosa

*if (-5.72 < (-0.09*SL - 0.01*SW - 0.63*PL - 0.61*PW) < -3.41)*

then Iris-Versicolor

*if (2.18 < (-0.09*SL - 0.32*SW + 0.51*PL + 0.69*PW) < 4.59)*

then Iris-Virginica

This inequality rule is much more efficient and accurate than the rule summarised statistically in section 3.5.2.1. The above rule can produce 100% accurate result for Iris dataset; whereas the statistical rule in [134] produces 10% misclassified patterns.

6.2.5 Analysis Result for Natural Language Phrase Recognition

From the learned ADFUNN on the natural language phrase recognition task as described in Section 3.5.3, locating the strongest 100 or so weights for sentence(S), verb phrase (VP) and noun phrase (NP), it is possible to see some inputs performed more important than others for that output and only these input tags will activate that output neuron. Therefore by looking at the weights it is possible to tell which input tags are most important for each neuron (See figure 77 for sentence, figure 78 for verb phrase and figure 79 for noun phrase). The tags summarised for each output neuron are not all of those that can activate this neuron but they have a higher chance of triggering a specific class because of stronger weights compared to other input tags.

For example, several combinations of four phrasal tags, will have a higher chance of producing a sentence phrase according to figure 77. For verb phrase (figure 78), a combination of four look-back tags, four phrasal tags and one look-ahead tag, will produce a verb phrase. There are many examples can be used to test the analysing result.

Take a combination input tags of NP VP PP from figure 77, the sentence example in figure 76 can be produced:

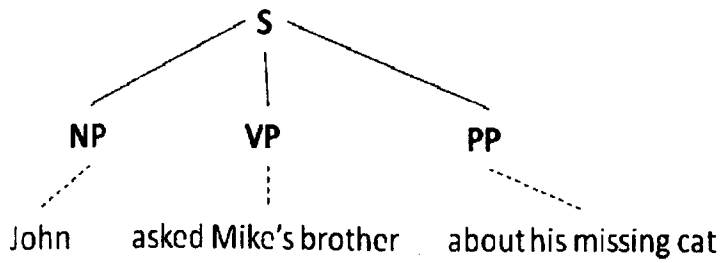


Figure 76: A sentence which is composed by NP + VP _+ PP

Dominant tags summarised are according to the experimental results for sentence, verb phrase and noun phrase. Although only 4 phrasal tags positions are dominant for these three output

phrases, there are in total 41 output classes and tags in other columns are important for other class below sentence such as 'phrase beginning with wh-word', or 'finite clause'. The proposed 15 input symbols: 4 look-back tags, 10 phrasal tags and 1 look-ahead tag in [145] are experimentally picked for the best performance.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				NP	NP	NP	NP							
				VP	VP	VP	JP							
				RP	RP	RP	RP							
				PP	PP	PP	PP							
				Rq		JP	Ti							
				E		Ti								
				CC										

Figure 77: Input tags for sentence

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NP	VP	NP	NP	V	V	V	V							N
Na		Na	Na	VP	VP	VP	VP							V
		RP	RP											P
		CC	VP											JP
			CC											CC
			Rq											
			E											

Figure 78: Input tags for verb phrase

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NP	NP	NP	VP	N	N	N	N							N
VP	VP	VP	RP	NP	NP	NP	NP							V
RP	RP	Rq	P	V	Po	P	Po							P
P	Na	Na	V			Po								CC
N	N	N	CC											
CC	CC	CC												
	Vi	E												
		Vi												

Figure 79: Input tags for noun phrase

The tags in figure 77 to figure 79 are:

NP: noun phrase
VP: verb phrase
RP: adverbial phrase
JP: adjectival phrase
PP: prepositional phrase
Rq: an adverb phrase beginning with a wh-word, e.g. `How do you feel'
Ti: to-infinitive clause
E: label used for existential there' i.e. `There' is nothing wrong
CC: conjunction e.g. AND, AND/OR, BUT, OR, YET
Na: a noun phrase marked as subject of verb
Po: a prepositional phrase beginning with the preposition of'
N: nouns
V: verbs
P: prepositions
ATI: singular or plural article (THE, NO)
NN: singular common noun

6.3 WELL-REGULATED LEARNED FUNCTIONS WITH NOISE

Along the lines of previous chapters, it is apparent that the learned ADFUNN (MADFUNN and SADFUNN) function curves are very well-regulated and characteristic curves. It has very clear edges for the learned range and also obvious steps within the learned range.

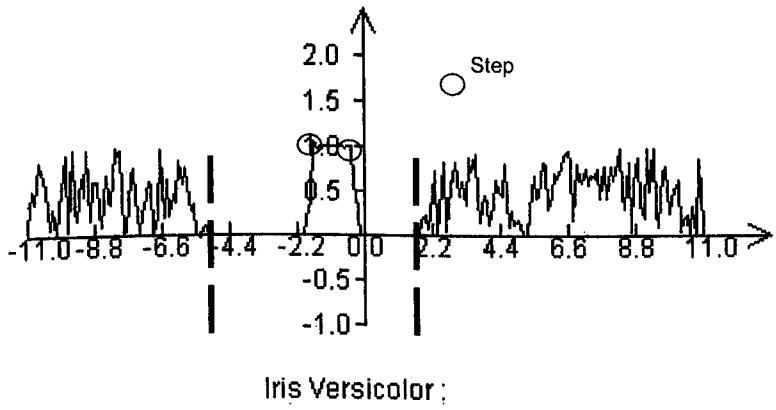


Figure 80: Easily identified learned range

Figure 80 is from learned Iris versicolor function which is initialised randomly between 0 and 1 (for better contrast result to highlight the active learning range). It can be clearly seen that

the learned range of function is very easily to be revealed among the random numbers (The learned range is the regulated curve within the two dashed lines). Also the steps are very clear which indicate the start/end point from/to which an iris versicolor class is activated.

6.4 LEARNED FUNCTION CURVE SMOOTHING

From a given set of analytical function prototypes, ADFUNN and MADFUNN have been developed to determine which analytical function matches best a given smoothed curve from the learned function output in chapter five. The number of f-points mostly depends on the complexity of the dataset. If a large number of f-points are needed for a complex dataset, some points may have never been adapted even though they are within the active range of adaptation; in this case they are in effect noise.

For a better analytical function classification result, the learned function curves are smoothed. Two mathematical methods the simple moving average method, and, least-squares polynomial smoothing are applied, as well as a more efficient method self-sizing moving window is introduced.

6.4.1 Function Range Transaction using Min-Max Normalisation

The learned functions' active ranges are different for each neuron if f-points or weights are initialized randomly. To transform the functions into a desired range, the method of min-max normalisation is ued to stretch or contract the range. The transformation formula is:
$v'(i) = [(v(i) - minA) / (maxA - minA)] \bullet (new\_maxA - new\_minA) + new\_minA$,
where *[minA, maxA]* is the active range of the learned function and *[new_minA, new_maxA]* is the desired range. In this thesis, active range is transformed into *{0, 1}*.

6.4.2 Simple Moving Average Smoothing

A simple moving average is formed by computing the average (mean) of a set of points over a specified window. The calculation is repeated for each point. The averages are then joined to form a smooth curve. The formula for a simple moving average is:

$$\overline{y_t} = (y_t + y_{t-1} + \ldots + y_{t-n-1}) / n$$

Where y is the variable, t is the current point, and n ($n=5$ in this case) is the window width for the average. For instance, if there are 100 points, the window width is 5, as the calculation continues, the newest point is added and the oldest point is subtracted.

6.4.3 Least-squares Polynomial Smoothing

The least squares curve fitting technique is the simplest and most commonly applied form of linear regression. The technique is to find the best fitting curve to a given set of points by minimizing the sum of the squares of the residuals (offsets) of the points from the curve. By implementing the least-squares linear regression analysis, it is easy to fit any polynomial of m degree $Y = a_0 + a_1x + \ldots + a_mx^m$ to experimental data $(x_1, y_1), (x_2, y_2)\ldots, (x_n, y_n)$, (provided that $n \geq m+1$) so that the sum of squared residuals S is minimized:

$$S = \sum_{i+1}^{n} [y_i - \hat{y}_i]^2 = \sum_{i+1}^{n} [y_i - (a_0 + a_1x_i + \ldots + a_mx_i^m)]^2$$

After getting the partial derivatives of S with respect to $a_0, a_1, .., a_m$ and equating these derivatives to zero, the following system of m-equations and m-unknowns $(a_0, a_1, .., a_m)$ is defined: $s_0a_0 + s_1a_1 + \ldots + s_ma_m = t_0, \ldots\ldots\ldots\ldots, s_ma_0 + s_{m+1}a_1 + \ldots + s_{2m}a_m = t_m$

where:

$$s_k = \sum_{i+1}^{n} x_i^k, \quad t_k = \sum_{i+1}^{n} y_i x_i^k$$

Thus it is easy to calculate the set of coefficients $(a_0, a_1, \ldots, a_m)$ which gives the resulting smoothed curve.

6.4.4 A New Self-Sizing Moving Window

This new method is introduced here because the two smoothing methods above both smooth the curves without keeping the original features of the function. Step changes in the curve are removed. Therefore this work has derived a novel approach to function smoothing, that preserves steps changes (discontinuities) whilst smoothing between them. This is achieved by means of a self-sizing smoothing window. The window expands over regions that do not contain large variations; and contracts in order to zoom in and jump over on regions that do contain large variations.

6.4.4.1 *Deviation within the Window*

A self-sizing moving window is formed by calculating the average deviation (δw) of all the points in the window, then adapting the size of it and moving to the next point. Small deviation indicates that there are no large steps in the window so it can be smoothed safely and the size of the window can be enlarged. In contrast, large deviation shows that there are big steps, the window size must be reduced and a standard deviation recalculated. If the large deviation still exists, the size of the window is reduced again (no smoothing).

Deviation is calculated using the following formula:

$$\phi = \sqrt{\frac{\sum (x - \bar{x})^2}{n-1}}$$

where ϕ, indicates the standard deviation, x is each point on the curve, and x-bar, denoted by the sign $\bar{x}$ is the average of the point set of the curve. And n is simply the number of values in the data sample

6.4.4.2 *Self-sizing moving window smoothing method*

T: *the threshold to set level of smoothing. T=0.2 in this case.*

k: *the level of resize. k=1.2 in this case.*

δw: *the standard deviation in current window*

S_w: *the size of the window*

X: *a set of f points within the window*

x: *the mid point of current window*

N_p: *the total number of f points on curve*

S_{min}: the minimum size of the window, S_{min}= 3 which is the minimum size a moving window can be reduced

for (int *$i = 0.5 \bullet S_w$; $i < N_p$; i++)*

{

//smooth the mid point in the window at one time

while *((δw >T) && (S_w >= S_{min}))*

```
{
    // reduce the size of the window
    Sw = Sw / k;
}
if (Sw > Smin)
{
    x = avg(X);
    //paint this new calculated mid point on curve
    paint(x);
}
// enlarge the size of the window
Sw = k • Sw;
}
```

This smoothing method loops and smoothes through all the f-points on the curve, starting from the middle point of the initial window size. If the standard deviation in current window has exceeded the threshold and the current windows size is bigger than 3, it means there are sharp steps in the current window and the window size must be reduced. When the window size cannot be reduced any more, the big step will be kept and the window moves to the next point.

6.4.5 Smoothing Experiments

After transforming the active range the following curves below can be obtained. Figure 81 is the original active range transformed from the learned verb phrase function. Figure 82 is the smoothed curve using the simple moving average and figure 83 is the best fitting 8 degree polynomial function using the least-squares method.

It is apparent that after smoothing by the simple moving average (result in figure 82); much of noisy data has been removed. Similarly, the polynomial smoothing removes more noise (figure 83), but it requires human intervention to choose the degree of the polynomial function for the best fitting curve. The best degree in this case is 8.

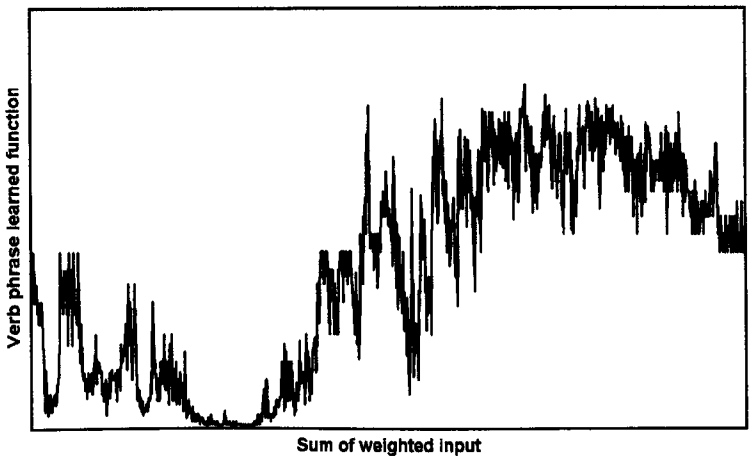


Figure 81: Original learned function of verb phrase

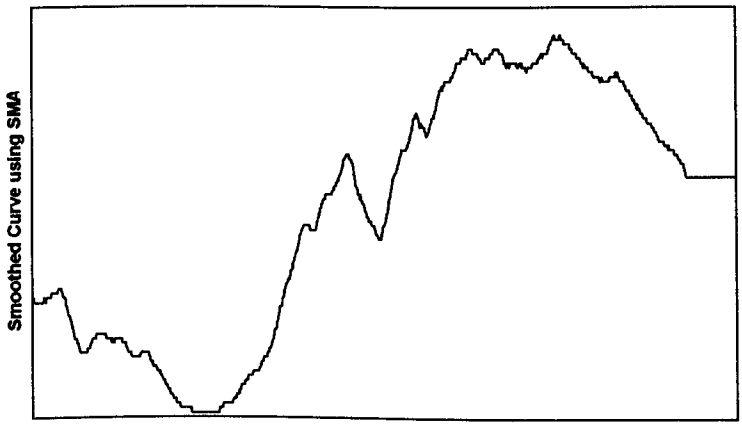


Figure 82: Smoothed curve of verb phrase learned function

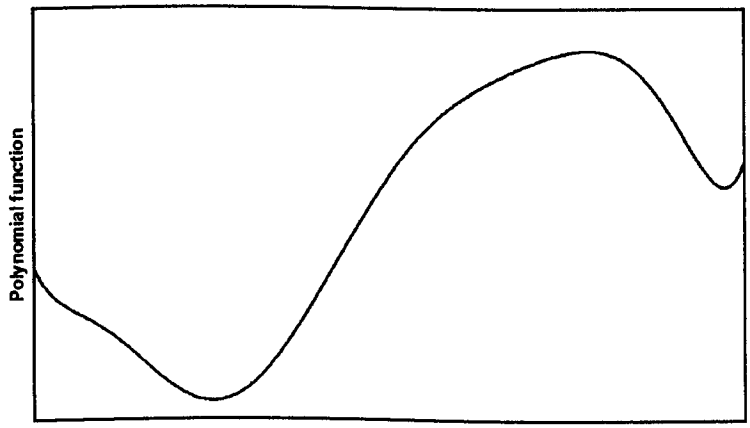


Figure 83: The best fitting 8 degree polynomial function using the least-squares method

In contrast, by using the self-sizing moving window to smooth a learned sentence function (with noise) in phrase recognition case, the important features of the original sentence (figure

84) learned function has been kept and the noise has been removed (figure 85). However, the features can not be kept by using the simple moving average method (figure.86).

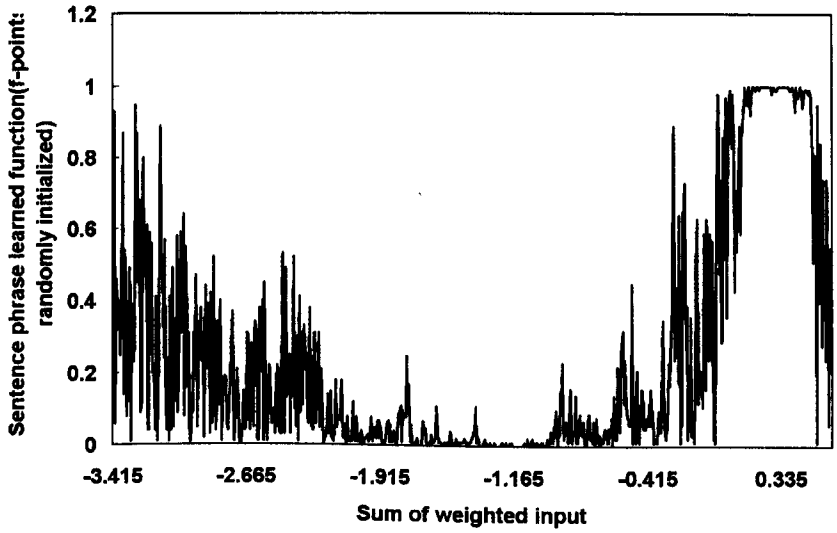


Figure 84: Normal learned sentence function

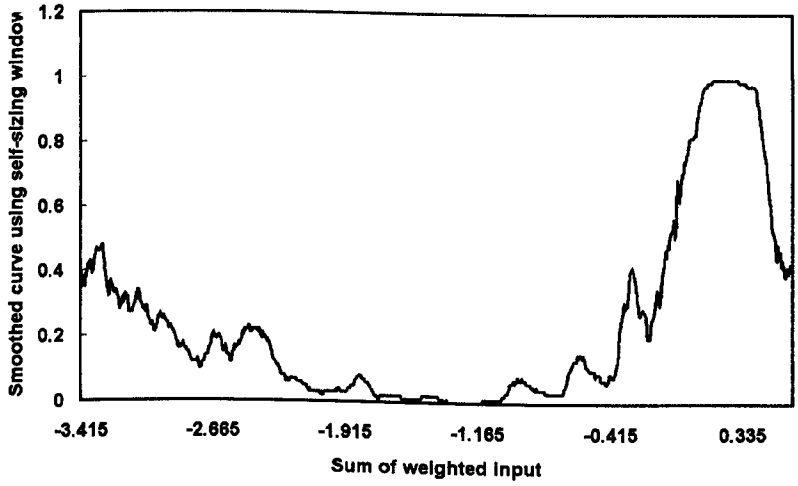


Figure 85: Smoothed curve of learned sentence function using this self-sizing window

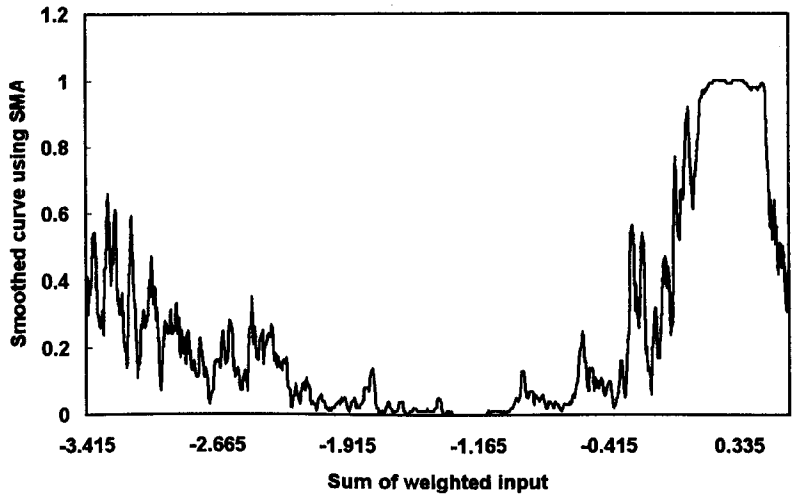


Figure 86: Smoothed curve of the learned sentence function using simple moving average

More examples have been tested and results show that the self-sizing window can get better smoothing results than the simple moving average in terms of removing noise whilst keeping the useful information. Figure 87 is a function curve learned for pulse function class in the analytical function recognition case using ADFUNN.

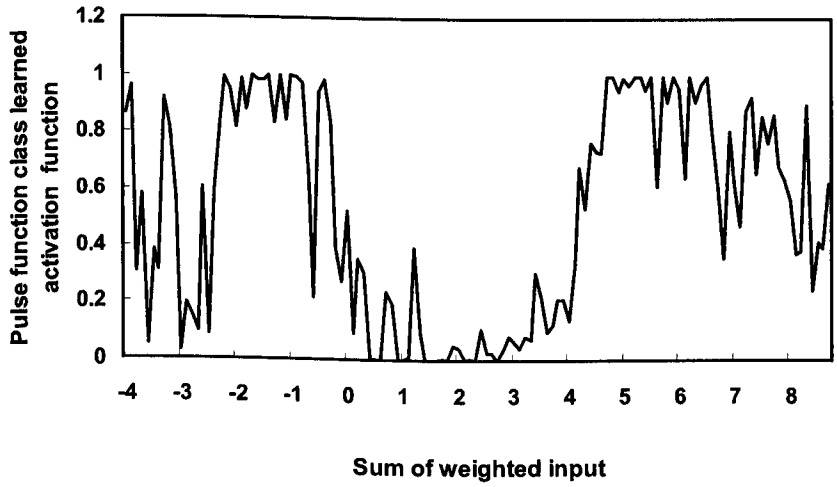


Figure 87: Pulse function class learned function curve

The smoothed curve in figure 88 is the result by applying simple moving average method on the above learned function curve.

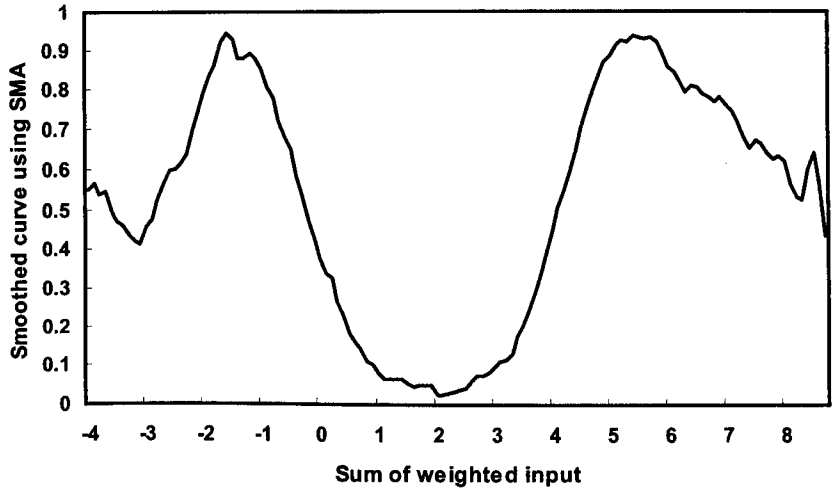


Figure 88: Smoothed function curve using simple moving average

The above curve is smoothed, however it is obvious that the important information of the learned function, like the discontinuities, have also been removed. In contrast, the smoothed curve in figure 89 using the self-sizing moving window method can not only get rid of noise, but also it can keep the sharp steps, which may indicate the starting/stopping of a learning range in learned ADFUNN function, or a learned feature from the data/problem.

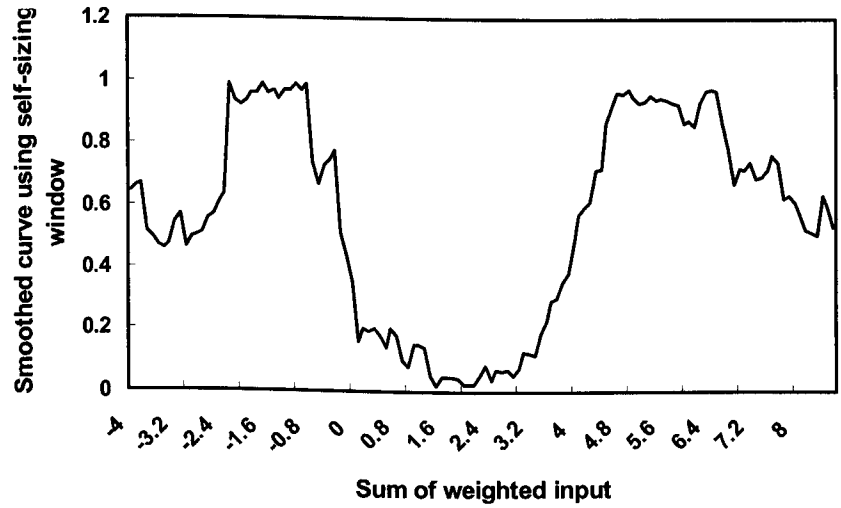


Figure 89: Smoothed function curve using self-sizing window

Figure 90 compares the difference between the original learned ADFUNN curve for analytical function recognition task pulse function class and the smoothed curves. The dark blue curve is the learned ADFUNN curve for pulse function class, cyan curve is the smoothed curve using simple moving average and the yellow curve is the smoothed curve using self-sizing moving window. If substituting the smoothed curves to all learned classes, 82.3% generalisation can be obtained from the smoothed curves using simple moving average and 85.1% generalisation is achieved from smoothed curves using our self-sizing window smoothing method, compared to 85.7% generalisation produced by the original learned function curves.

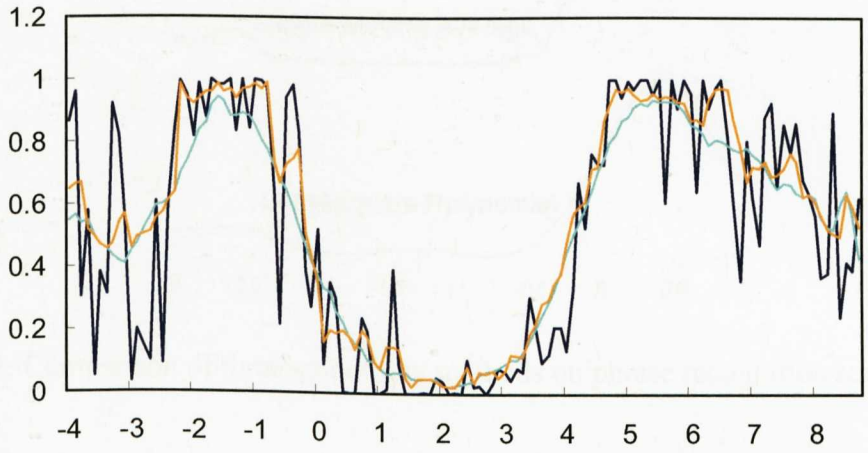


Figure 90: Learned function with smoothed curves

The following example of sentence class illustrates a very good example of the advantages of the self-sizing window. After transforming the active range from the learned sentence class the original learned sentence curve is shown at the top in figure 91. The curve below it in the same figure are the smoothed curve using the new proposed self-sizing window; the smoothed curve using the simple moving average; and, polynomial function by the least-squares method.

It is apparent that after smoothing, a large amount of noisy data have been removed. Most importantly, the example illustrates a very obvious advantage of self-sizing smoothing method compare to other smoothing methods. The self-sizing window keeps discontinuities as well as removing noise.

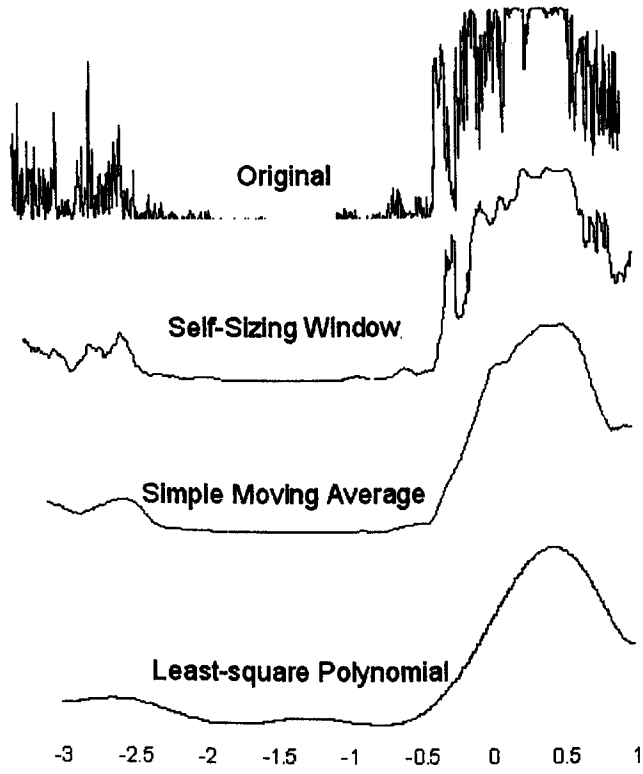


Figure 91: Comparison of three smoothing methods on phrase recognition sentence class

6.4.6 Smoothed Curves Performance

How do these smoothed curves work when substituted back into the neural network? The simple moving average method causes less distortion than the least square polynomial smoothing. For the natural language processing case, experiments were performed to substitute simplified curves smoothed by the simple moving average method for empirical ones for all of the 41 constituent tag output classes. The simplified curves work well, the correct classification is still 100% for all patterns used for training (254/254*100% = 100%) e.g. for the verb phrase neuron as shown in figure 92 and 93. Figure 92 is the original learned verb function and figure 93 is the smoothed one.

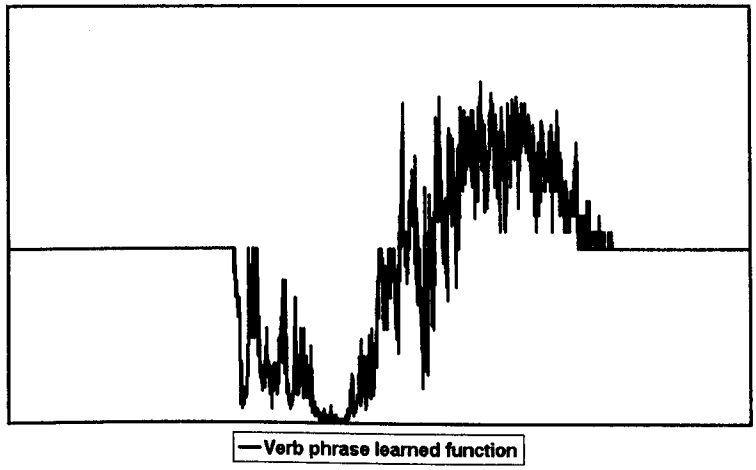


Figure 92: Verb phrase learned function

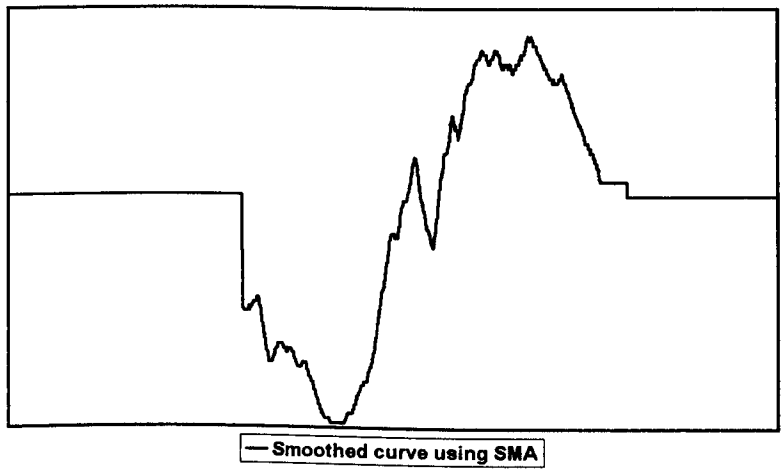


Figure 93: Substituting this smoothed curve to the verb phrase neuron

However, the best approach is the self-sizing moving window smoothing method, because it keeps the important features of the original learned functions and increases the interpretability of the functions. For example, when applied to the XOR adapted function, it preserves the discontinuities, and equally it preserves the smoothing of an underlying sinusoid whilst removing noise. Four phrase recognition function output curves are replaced with the self-sizing moving window smoothed curves and the average correct classification is reduced, but only very slightly from 100% to 98.7%.

The diagram in figure 94 shows a very obvious advantage of self-sizing smoothing method compare to other smoothing methods. The self-sizing window keeps discontinuities as well as removing noise. In this case, the original learned function has a small discontinuities near x =

0 around 0.5. However, only the self-sizing window method kept this discontinuity, SMP and least-square polynomial smoothing methods both ignored and smoothed this part of the curve.

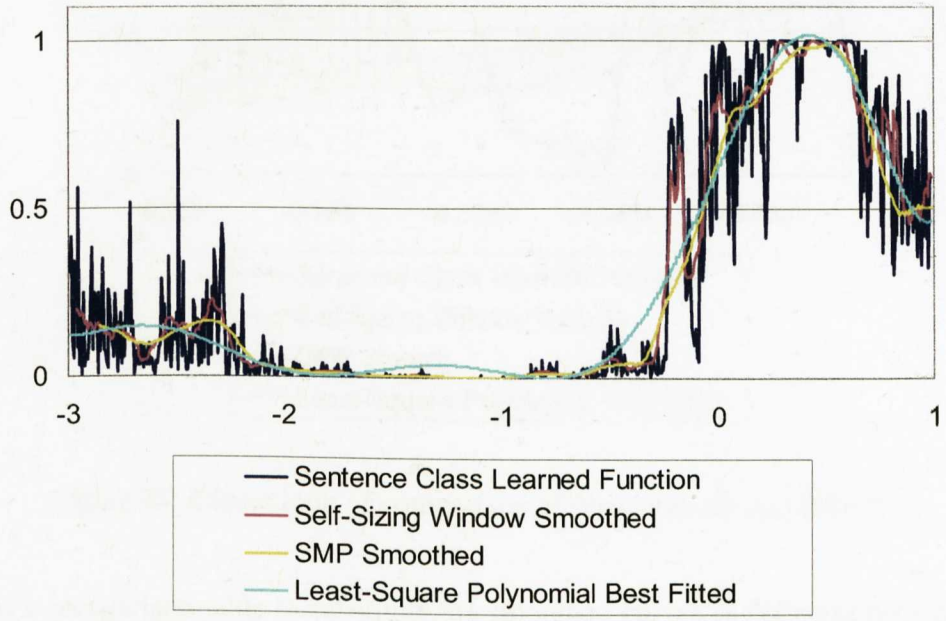


Figure 94: Comparison of three smoothing methods on phrase recognition sentence class

Figure 95 takes a closer look of this part of the graph. The input pattern presenting to the range between {-0.235, -0.135} expect output above 0.5 which is the class threshold; a result above 0.5 indicates a winning node as seen from the original learned function. However, only the smoothed curve using self-sizing window can give a similar result, whereas in contrast, the other two smoothing curves all give output below 0.5.

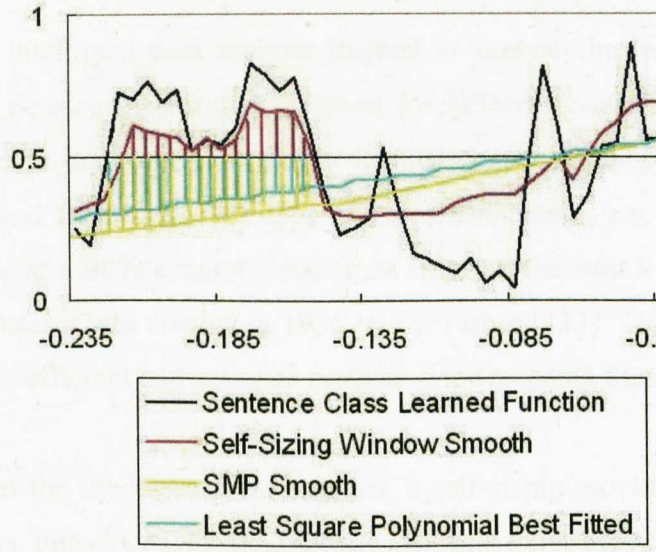


Figure 95: Closer look of comparison of three smoothing methods

Tabel 11 is a comparison table to substitute the smoothed curves to different learned function curves. The self-sizing window smoothed curves work perfectly when substitute into the learned network, the performance is much better than with the simple moving average smoothed curves.

Table 12: Performances comparison between smoothed curves and original curves

	Smoothed using SMA	Smoothed using self-sizing window	Original function curves
XOR	86.7 %	100 %	100 %
Iris Dataset	94.2 %	99.1 %	100 %
Phrase Recognition	93.6 %	98.7 %	100 %
Analytical Function Recognition	82.3 %	85.1 %	91.6 %

6.5 CONCLUSION

This section firstly looks at intelligent data analysis for different datasets learned by ADFUNN as examples (datasets learned by MADFUNN or SADFUNN can also be used for experiments).

Using traditional intelligent data analysis method to analyse the learned weights, important input factors can be identified and established for different classes. More significantly, this Chapter introduces an accurate and efficient inequality rule can be produced by analysing the learned weights and functions. This type of rules can obtain, e.g. for Irish Dataset, 100% accuracy compare to a 90% accuracy statistical rule. As the best known pattern recognition dataset, the Iris Dataset was created in 1936 by Mr Fisher [133]. This type of inequality rules are by far the most efficient and accurate method in the research literature.

To further support the intelligent data analysis, a self-sizing moving window is proposed in this chapter. It is introduced to smooth learned functions produced from ADFUNN (or MAD|FUNN or SADFUNN). The self-sizing moving window expends and smoothes over points that do not contain large deviations, zooms in and jumps over on points that do contain large deviations. In this way, the class active boundaries are kept and noise data are removed. Other smoothing method like the simple moving average and the least square polynomial smoothing method will smooth both of the active boundaries and noise data.

CHAPTER 7 CONCLUSIONS

7.1 INTRODUCTION

In this Chapter, a general conclusion of the whole thesis is summarised in section 7.2. It reviews the methodologies, experimental datasets, data analysis used in this thesis. The performance results from experiments by applying the methodologies are also concluded.

It then discusses and summarises the key findings of this project and main contributions to knowledge of this research in 7.3. At the end of this chapter, recent projects are reviewed and future projects are proposed. Examples are: a proposed unsupervised ADFUNN (or MADFUNN or SADFUNN) development, the application of ADFUNN (or MADFUNN or SADFUNN) on Enabled Self Procurement (ESP) project to help people build sustainable communities, a proposed handwritten electronic signature authentication project; and, an assortment of collected data projects.

7.2 SUMMARY

The research in this thesis investigated a novel adaptive function neural network (ADFUNN) and its two forms of extensions MADFUNN and SADFUNN. ADFUNN is based on a linear piecewise artificial neuron activation function which is modified by a gradient descent supervised learning algorithm. Its function adaptation Δf process is carried out in parallel with the traditional weights adaptation Δw process. Several linearly inseparable problems, as proved in Chapter 3, like XOR, Iris dataset, natural language processing phrase recognition task, were learnt rapidly and without a single hidden neuron with ADFUNN.

To support more complex datasets and to achieve more efficient generalisation abilities, a multi-layer ADFUNN (MADFUNN) was introduced in Chapter 4. MADFUNN was applied to two complex datasets, the letter image recognition tasks dataset and analytical function recognition dataset. Experimental results showed that MADFUNN exhibited higher

generalisation ability than most of the other methods which have been applied on the letter image task and required less complexity in the system implementation. On the analytical function recognition task, the analytical functions which MADFUNN has classified the smoothed learned function curves to, work very well if substituted back to its corresponding output neuron. For a more accurate and efficient smoothing result, a self-sizing moving window was introduced in Chapter 6. It smoothes the learned function curves yet keeps the important information on step changes/discontinuities.

In Chapter 5, an existing unsupervised Snap-Drift was combined with a supervised ADFUNN acting on the activation functions, to perform classification. Snap-Drift is very effective in extracting distinct features from the complex cursive-letter datasets. Experiments show only a couple of epochs are enough for the feature classification. It helps the supervised single layer ADFUNN to solve these linearly inseparable problems rapidly without any hidden neuron. From the experimental results, it is clear that when combined within one network (SADFUNN), the two methods exhibited higher generalisation abilities than MLPs even though the learning process is simpler. An additional benefit of ADFUNN is that the learned functions can support intelligent data analysis. In Chapter 6 the learned weights and functions from different applications of ADFUNN were analysed and some important information revealed.

This chapter presents some specific conclusions from the discussion and summary of the earlier chapters, but mainly on the novelty of the whole research. The powerful performance and simple structure of ADFUNN can be extended and combined with other methods to create new form of modal learning. Furthermore; the unsupervised learning method can be deployed on ADFUNN to perform a fast and simple unsupervised learning. Therefore, further work of this research is highly recommended.

7.3 DISCUSSIONS

One of the main novelties of this research is the new Modal Learning method introduced in this thesis. It adapts the activation function inside each neuron in parallel with the traditional weights adaptation between neurons in a single layer neural network structure.

To outline the key findings of this research, the core new approach introduced in this project is ADFUNN which has proven to be able to overcome linear inseparability limitations in a single layer network. It successfully solved non-linear problems in a single layer structure in which no such method has ever been able to overcome in neural computing history. More significantly, ADFUNN has been proved to be more powerful than related works (computationally more than 2,000 times powerful than the closest work), yet faster and simpler (a great contribution to speed up learning process and simplify hardware requirements).

Another successful finding of this project is a Multi-layer extension of ADFUNN, MADFUNN which was introduced for extremely large and complex datasets. It exhibited highly efficiency comparing to related works (computationally much more powerful and simpler than other works).

SADFUNN is a feature extraction version of ADFUNN combined with an unsupervised Snap-Drift method. SADFUNN was introduced for extremely complex datasets with large dimensional data. It has also proven to be able to obtain similar results comparing to related works but with significantly simplified network structure and faster speed.

By utilising the novel learned functions and traditional learned weights from ADFUNN (or MADFUNN or SADFUNN), this research can also produce highly accurate and efficient inequality rules to perform intelligent data analysis. The rules offer a more intelligent way to analyse data. Moreover, no such accurate or efficient rules have ever been invented in similar research literature.

Another significant contribution of this research is the introduction of a self-sizing smoothing moving window which was introduced to assist with data analysis. To the auther's best knowledge, no such function curve smoothing method has been reported or published, which is able to smooth function curves whereas preserving useful information of the data at the same time.

7.4 FUTURE WORK

7.4.1 Unsupervised ADFUNN

As unsupervised learning involves no target values, by changing the rules of function learning, it is possible for ADFUNN to observe only the features rather than to describe how the data are organised or clustered.

The idea is basically to find the tendencies of adaptation directions of adaptive functions and make these tendencies more obvious and clear. As the functions are adapted either towards 0 or 1 and are initialised randomly between middle point 0.5, if the distance between maximum output and the middle point is bigger than the distance between the middle point and the minimum output, the maximum output neuron shows a tendency to go up and therefore it is adapted upwards, whereas other neurons are adapted downwards. Otherwise, the minimum neuron is adapted downwards and other neurons are adapted upwards.

A variety of parameters can affect the performance of this proposed unsupervised single layer adaptive function neural network. For instance: the number of output nodes; the number and initialisation range of f-points; initialisation of weights; learning rates for both functions and weights. Further work based on these ideas are worthy of investigation, since the single layer ADFUNN performed effectively whilst simpler than many other supervised learning methods, this single layer unsupervised ADFUNN may achieve better performance than other unsupervised methods.

7.4.2 Apply ADFUNN (or MADFUNN or SADFUNN) to UrbanBuzz ESP Project

The Enabled Self Procumbent (ESP) is an UrbanBuzz funded project to help people design their own houses in order to build sustainable communities in the Thames Gateway [178] area. Each house is procured by an individual rather than delivered by a speculative volume house builder on the open market. The ESP project is a multi-user platform where people can select a plot, build their own house from a database of pattern book houses and even see their neighbours' designs. User can specify the design brief to perform design evaluation, scoring and house type matching.

ADFUNN (or MADFUNN, SADFUNN) is going to be applied to the ESP project as a neural network matching system. By specifying the design brief, such as number of bedrooms, size of the garden, number of car parks, total budget, the neural network matching system will be able to help the user to find an appropriate house type or suitable pattern book from the database. The strategy is explained in figure 96.

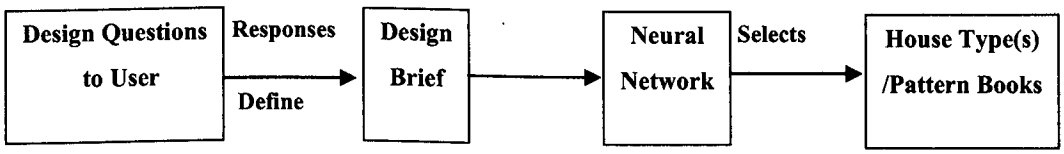


Figure 96: Strategy of the neural network matching system for ESP project

7.4.3 Apply ADFUNN (or MADFUNN or SADFUNN) to Handwritten Electronic Signature Authentication

Electronic signature scheme notion was introduced by Whitfield Diffie and Martin Hellman since 1976 [179], however it still has not become popular due to the lack of technology with verification tools. An important feature of paper based signatures is that they can be individually studied and analysed by handwriting experts, by comparing with other existing samples for authentication. Whereas, this is the most significant challenge for authenticating electronic signatures, which could make it become worthless if cannot be associated with signers.

ADFUNN (or MADFUNN or SADFUNN) is proposed to be applied to an electronic signature authentication project; specifically a computer mouse based electronic signature verification which is more efficient, cost effective and easy to use. The proposed project will be set up as a case study in a small to medium sized company. Signature samples will be collected from the company and verification of their signatures will be performed for any new signed document.

7.4.4 Apply ADFUNN (or MADFUNN or SADFUNN) to More Collected Data

ADFUNN (or MADFUNN or SADFUNN) also is going to be applied to more collected data. Complex data was used by Roadknight et. al [117] before. They collected data from open topped chamber experiments, outdoor experiments, and closed-chamber experiments for ozone related injury and the accompanying levels for a variety of pollutants and climatic factors [117]. The methodology for data collecting used in ADFUNN (or MADFUNN or SADFUNN) may be similar with the above, but the data type may vary. There are some prospective projects where ADFUNN (or MADFUNN or SADFUNN) could be applied to, such as a water hazard related project with the International Centre for Water Hazard and Risk Management (ICHARM) in Japan, in order to help people better predict water-related disasters.

7.4.5 Apply ADFUNN (or MADFUNN or SADFUNN) to Fuzzy Neuron System

As introduced in section 2.3.1 in the beginning of this thesis, the combination of fuzzy logic system with artificial neural network can help solve the inherited limitations of each isolated paradigm and therefore produce a more powerful and efficient model. Because the new methods proposed in this thesis have exhibited faster learning ability and produced more powerful results with much simpler network structure and less hardware requirements. Combining ADFUNN (or MADFUNN or SADFUNN) with a fuzzy logic system could possibly produce more accurate and faster learning results, yet reduce the hardware costs in a fuzzy neuron control system.

7.5 FINAL THOUGHTS

The research presented in this thesis has led to the development of a single weight layer supervised network to overcome linear inseparability limitations. The experimental results demonstrate that a single network becomes more effective (than clustering or perceptron in this thesis) by integrating two learning modes into one network. The novel learning feature in ADFUNN (as well as in MADFUNN and SADFUNN) is that of simultaneous adaptation between (weights) and within (functions) neurons, which results in highly effective performance in a simple network structure. The outcome of the research has provided significant and immediate benefits to a range of applications. The research has also provided opportunities for further development and implementations of modal learning methods. The author believes that the *ADFUNN (MADFUNN, SADFUNN)* algorithms will be widely explored and succeed in a wider range of applications.

REFERENCES

1. Bloch, G. and Denoeux, T., 2003. Neural Networks for Process Control and Optimization: Two Industrial Applications. *ISA transactions,* vol. 42, pp.39-51.

2. Roadknight, C. M., Palmer-Brown, D. and Al-Dabass, D., 2003. Simulation of Correlation Activity Pruning Methods to Enhance Transparency of ANNs. *International Journal of Simulation*, vol. 4 (2), pp. 68 - 74.

3. Kaastra, I. and Boyd, M., 1996. Designing a Neural Network for Forecasting Financial and Economic Time Series. *Neurocomputing,* vol. 10, pp. 215 - 236.

4. Pattichis, C. S., Schizas, C. N. and Middleton, L., 1995. Neural Network Models in EMG Diagnosis. *IEEE Transactions on Biomedical Engineering,* vol. 42 (5), pp. 486 - 496.

5. Palanivel, S., Venkatesh, B. S. and Yegnanarayana, B., 2003. Real Time Face Authentication System using Autoassociative Neural Network Models. *ICME,* vol. I, pp. 257-260.

6. Lee, S. W. and Palmer-Brown, D., 2006. Phonetic Feature Discovery in Speech using Snap-Drift, *International Conference on Artificial Neural Networks (ICANN'2006)*, pp. 952 – 962.

7. Johnson, R. C. and Brown, C., 1998. *Cognizers: Neural Networks and Machines that Think.* New York, John Wiley & Sons.

8. McCulloch, W. S. and Pitts, W., 1943. A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics.*

9. McCulloch, W. S. and Pitts, W., 1947. How We Know Universals: The Perception of Auditory and Visual Forms. *Bulletin of Mathematical Biophysics.*

10. Minsky, M. L. and Papert, S. A., 1969. *Perceptron,* MA, MIT Press, Cambridge.

11. Rosenblatt, F., 1962. *Principles of Neurodynamics.* New York, Spartan.

12. Rosenblatt, F., 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, vol. 65, pp. 386 - 408.

13. Rosenblatt, F., 1958. Two Theorems of Statistical Separability in the Perceptron. *In: Proceedings of Mechanization of Thought Processes.* National Physical Laboratory, London: HM Stationery Office, pp. 421 - 456.

14. Efe, M. Ö., 2002. A Novel Error Critic for Variable Structure Control with an ADALINE. *Transactions of the Institute of Measurement & Control*, vol.24, no.5, pp.403-415.

15. Hamilton-Wright, A. and Stashuk, D. W., 2005. Comparing "Pattern Discovery" and Back-Propagation Classifiers. *International Joint Conference on Neural Networks (IJCNN 2005)*, Montreal, Canada, pp. 1286-1291.

16. Widrow, B. and Lehr, M. A., 1990. 30 Years of Adaptive Neural Network: Perceptron, Madaline and Backpropagation. *In: Proceedings of IEEE*, vol. 78, pp. 1415 - 1442.

17. Huang, X., 2005. A New Kind of Hopfield Networks for Finding Global Optimum. *International Joint Conference on Neural Networks (IJCNN 2005)*, Montreal, Canada, pp. 764-769.

18. Nishikawa, I., Sakakibara, K., Iritani, T. and Kuroe, Y., 2005. Two Types of Complex-Valued Hopfield Networks and the Application to a Traffic Signal Control. *International Joint Conference on Neural Networks (IJCNN 2005)*, Montreal, Canada, pp. 770-775.

19. McClelland, J. L. and Rumelhart, D. E., 1998. Explorations in Parallel Distributed Processing: *MIT Press*.

20. Hinton, G.E., 1992. How Neural Networks Learn from Experience. *Scientific American*, vol. 267, pp. 144-151.

21. Kurzynski, M., Puchala, E. and Rewak, A., 2006. The Bayes-Optimal Feature Extraction Procedure for Pattern Recognition using Genetic Algorithm. *16th International Conference on Artificial Neural Networks (ICANN 2006)*, Part I, Athens, Greece, pp.21-30.

22. Suykens, A. K., et al., 2003. Advances in Learning Theory: Methods, Models and Applications. *NATO Science Series III: Computer & Systems Sciences*, vol.190, IOS Press Amsterdam.

23. Schlesinger, M. I. and Hlavác, V., 2002. *Ten Lectures on Statistical and Structural Pattern Recognition*, Kluwer Academic Publishers.

24. Webb, A., 2004. *Statistical Pattern Recognition*, Wiley.

25. Stork, D. G. and Yom-Tov, E., 2004. *Computer Manual in MATLAB to accompany Pattern Classification*, Wiley Interscience.

26. Carreras, X., Collins, M. and Koo, T., 2008. TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-rich Parsing, *Proceedings of CONLL08*.

27. Smart, M. H. W. and Murray, A. F., 1996. Multilayer Perceptron for Rotationally Invariant Feature Extraction and Classification. *Proceedings of Applications and Science of Artificial Neural Networks.*

28. Palubinskas, G., Descombes, X. and Kruggel, F., 1998. An Unsupervised Clustering Method Using the Entropy Minimization. *14th International Conference on Pattern Recognition*, Vol.2.

29. Yager, R. R., 2006. An Extension of the Naive Bayesian Classifier. *Information Sciences*, vol. 176, issue. 5, pp. 577-588.

30. Shawe-Taylor, J. and Cristianini, N., 2004. *Kernel Methods for Pattern Analysis*, Cambridge University Press.

31. Lee, S. W. and Palmer-Brown, D., 2006. Modal Learning in A Neural Network. *1st Conference in Advances in Computing and Technology, London, United Kingdom,* pp. 42 – 47.

32. Kohonen, T., 1990. Improved Versions of Learning Vector Quantization. *International Joint Conference on Neural Networks*, vol. 1, pp. 545-550. San Diego, CA.

33. Callan, R., 1999. *The Essence of Neural Networks*, Prentice Hall Europe, pp. 36-41.

34. Hebb, D. O., 1961. Distinctive Features of Learning in the Higher Animal. *Brain Mechanisms and Learning*, London: Oxford University Press.

35. Carpenter, G. A. and Grossberg, S., 2003. Adaptive Resonance Theory. *The Handbook of Brain Theory and Neural Networks,* Second Edition. Cambridge, MA: MIT Press, pp. 87-90.

36. Quinlan, J. R., 1996. Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, vol. 4, pp. 77-90.

37. Mitchell, T. M., 1997. *Machine Learning*. McGraw Hill, Chapter 3.

38. Quinlan, J. R., 1986. *Machine Learning*, Springer.

39. Su, J. and Zhang, H., 2006. A Fast Decision Tree Learning Algorithm. *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06).*

40. Tong, W., Hong, H., Fang, H, Xie, Q and Perkins, R, 2003. Decision Forest: Combining the Predictions of Multiple Independent Decision Tree Models. *Journal of Chemical Information and Computer Science.*

41. Cios, K. and Liu, N., 1992. A Machine Learning Method for Generation of Neural Network Architecture: A Continuous ID3 Algorithm. *IEEE Transaction of Neural Networks*, vol. 3, no. 2, pp. 280-291.

42. Quinlan, J., 1986. Induction of Decision Trees. *Machine Learning.* vol. 1, pp. 81-106.

43. Drake, P. R. and Packianather, M. S., 1998. A decision tree of neural networks for classifying images of wood veneer. The *International Journal of Advanced Manufacturing Technology.* Springer London, vol. 14, no. 4.

44. Utgo, P. E., 1989. Incremental Induction of Decision Trees. *Machine Learning*, vol. 4, no. 161.

45. Cios, K. J. and Sztandera, L. M., 1992. Continuous ID3 algorithm with fuzzy entropy measures. *IEEE International Conference on Fuzzy Systems*, vol. 8, no. 12, pp.469 – 476.

46. Quinlan, J. R., 1987. Generating production rules from decision trees. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 304–307.

47. Liu, H., 1996. Efficient Rule Induction from Noisy Data. *Expert Systems with Applications.*

48. Clark, P. and Niblett, T., 1989. The CN2 Induction Algorithm. *Machine Learning,* vol. 3, no. 4, pp.261-283.

49, Clark, P. and Boswell, R., 1991. Rule Induction with CN2: Some recent improvements. *Lecture Notes in Computer Science, Machine Learning — EWSL-91*, vol. 482.

50. Kralik, P. and Bruha, I., 1999. Discretizing Numerical Attributes in A Genetic Attribute-based Learning Algorithm.

51. Fausett, L., 1994. *Fundamentals of Neural Networks,* New York: Prentice Hall.

52. Sutton, R. S. and Barto, A. G., 1998. Reinforcement Learning: An Introduction. *MIT Press*, Cambridge.

53. Crites, R. H. and Barto, A. G., 1998. *Elevator Group Control Using Multiple Reinforcement Learning Agents.* Machine Learning, Springer.

54. Littman, M. L., 2001. Value-Function Reinforcement Learning in Markov Games. *Cognitive Systems Research,* Elsevier.

55. Pednault, E., Abe, N. and Zadrozny, B., 2002. Sequential cost-sensitive decision making with reinforcement learning. *Proceedings of the 8th ACM SIGKDD.*

56. Peterson, C. and Soderberg, B., 1989. A New Method for Mapping Optimization Problems onto Neural Networks. *International Journal of Neural Systems.*

57. White, H., 1992. *Parametric Statistical Estimation with Artificial Neural Networks, Economics Working Paper Series with number 92-13.* University of California.

58. Fogel, G. B. and Corne, D. W., 2003. Evolutionary Computation in Bioinformatics. *Morgan Kaufmann.*

59. Haykin, S., 2002. Kalman Filtering and Neural Networks. *John Wiley and Sons Inc.*

60. White, H., 1992. Parametric Statistical Estimation with Artificial Neural Networks. *Economics Working Paper Series with number 92-5*, University of California.

61. Xiang, C., Fan, X. A. and Lee, T. H., 2004. Face Recognition using Recursive Fisher Linear Discriminate with Gabor Wavelet Coding. *2004 International Conference on Image Processing,* Singapore, pp.79 – 82.

62. Stanton, C., "Linear Regression Java Applet for Probability and Statistics", Department of Mathematics, University of Wisconsin-Madison, viewed 24 August 2010, <http://www.math.csusb.edu/faculty/stanton/m262/regress/index.html>.

63. Kotsiantis, S. and Pintelas, P., 2005. Logitboost of Simple Bayesian Classifier. *Computational Intelligence in Data mining Special Issue of the Informatics Journal,* vol. 29, no.1, pp. 53-59.

64. Boulle, M., 2009. A Parameter-Free Classification Method for Large Scale Learning. *Journal of Machine Learning Research*, vol. 10, pp.1367-1385.

65. Freund, Y. and Schapire, R. E., 1998. Large Margin Classification Using the Perceptron Algorithm. *In the Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT' 98).* ACM Press.

66. Vapnik, V. N., 1998. *Statistical Learning Theory.* New York, John Wiley and Sons.

67. Vapnik, V. N., 1995. *The Nature of Statistical Learning Theory.* Berlin, Springer.

68. Joachims, T., 2006. Training Linear SVMs in Linear Time. *KDD'06, 2006,* Philadelphia, Pennsylvania, USA.

69. Xiao, Y., Rao, R., Cecchi, G. and Kaplan, E., 2008. Improved Mapping Of Information Distribution Across The Cortical Surface With The Support Vector Machine. *Neural Networks, 2008 Special Issue: Advances in Neural Networks Research: IJCNN07*, vol. 21, pp. 341- 348.

70. Suykens, J. A. K., Horvath, G., Basu, S., Micchelli, C. and Vandewalle, J., 2003. Advances in Learning Theory: Methods, Models and Applications. *Computer and Systems Sciences,* vol. 190 of NATO-ASI Series III.

71. Tipping, M. E., 2000. The Relevance Vector Machine. *Advances in Neural Information Processing Systems.*

72. Gray, A., 1997. *The Intuitive Idea of Distance on a Surface.* Boca Raton, FL: CRC Press, pp. 341-345.

73. Gill, B., 2007. *Bayesian Methods: A Social and Behavioral Sciences Approach.* Chapman and Hall/CRC, 2nd edition.

74. Mason, L., Baxter, J., Bartlett, P. and Frean, M., 2000. Boosting Algorithms as Gradient Descent. *Advances in Neural Information Processing Systems,* vol. 12, pages 512--518, MIT Press.

75. Buf, H. D. and Bayer, M. M., 2002. Automatic Diatom Identification. *World Scientific, Series in Machine Perception and Artificial Intelligence*, vol. 51, pp.316.

76. Dowe, D. L., Gardner, S. and Oppy, G., 2007. Bayes not Bust! Why Simplicity is no Problem for Bayesians. *Brit. J. Phil. Science*, vol. 58, pp. 46.

77. Wallace, C. S. and Dowe, D. L., 1999. Minimum Message Length and Kolmogorov Complexity. *Computer Journal*, vol. 42, no. 4, pp270-283.

78. Newall, P., "Ockham's Razor", viewed 1 September 2010, <http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor>

79. Chickering, D. M., Heckerman, D. and Meek, C., 2004. Deciding Conditional Independence is Hard in Noncausal Directions. Exact Inference in Large Networks Takes a Very Long Time. *Journal of Machine Learning*, vol. 5, pp. 1287–1330.

80. Fukunaga, K., 1990. Introduction to Statistical Pattern Recognition (Second Edition). *Academic Press*, New York.

81. Gegov, A., 2007. Complexity Management in Fuzzy Systems - A Rule Base Compression Approach. *Studies in Fuzziness and Soft Computing 211*, 7–16, Springer-Verlag Berlin Heidelberg.

82. Liu, P. and Li, H., 2004. *Fuzzy Neural Network Theory and Application.* World Scientific Publishing Co. Pte. Ltd.

83. Abraham, A., 2001. Neuro *Fuzzy Systems: State-of-the-art Modelling Techniques.* Springer-Verlag, Germany, pp.269-276.

84. Jang, S. R., 1993. ANFIS: Adaptive-Network-Based Fuzzy Inference Systems. *IEEE Trans. Systems, Man & Cybernetics* Vol. 23, pp.665-685.

85. Halgamuge, S. K. and Glesner, M., 1994. Neural Networks in Designing Fuzzy Systems for Real World Applications. *Fuzzy Sets and Systems*, Vol. 65, pp.1-12.

86. Berenji, H. R. and Khedkar, P., 1992. Learning and Tuning Fuzzy Logic Controllers Through Reinforcements. *IEEE Trans. Neural Networks*, Vol. 3, pp. 724-740.

87 Lin, T. C. and Lee, C. S., 1991. Neural Network Based Fuzzy Logic Control and Decision System. *IEEE Transactions on Computers*, Vol. 40, no. 12, pp. 1320-1336.

88. Haykin, S., 1998. *Neural Networks a Comprehensive Foundation*, Prentic-Hall Inc, 2nd edition.

89. Widrow, B. and Hoff, M. E., 1960. Adaptive Switching Circuits. *IREWESCON Convention Record*, pp. 96-104.

90. Broomhead, D. S. and Lowe, D., 1988. Multivariate Functional Interpolation and Adaptive Networks. *Complex Systems*, vol. 2, pp. 321-355.

91. Park, J. and Sandberg, I. W., 1993. Approximation and Radial Basis Function Networks. *Neural Computation*, vol. 5, no. 2, pp. 305-316.

92. Ma, L., Xin, K. and Liu, S., 2008. Using Radial Basis Function Neural Networks to Calibrate Water Quality Model. *In: proceedings of World Academy of Science, Engineering and Technology.*

93. Martin, D., Buhmann, M. and Ablowitz, J., 2003. Radial Basis Functions: Theory and Implementations. *Cambridge University Press.*

94. Mandic, D. P. and Chambers, J. A., 2001. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability.* John Wiley & Sons Ltd, England.

95. Mandic, D. and Chambers, J., 2001. *Recurrent Neural Networks*. Wiley.

96. Elman, J. L., 1990. Finding Structure in Time. *Cognitive Science,* vol. 23, pp.417-437.

97. Williams, R. J. and Zipser, D., 1989. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computing*, vol. 1, pp. 270–280.

98. Rumelhart, D. E., Hinton, G. E. and Williams, R. J., 1986. Learning Internal Representations By Error Propagation. *In: D.E. Rumelhart, J.L. McClelland (Eds.), Parallel Distributed Processing*, vol. 1, MIT Press, Cambridge, MA.

99. Rumelhart, D. E., Hinton, G. E. and Williams, R. J., 1986. Learning Representations by Back-propagation Errors. *Nature*, vol. 323, pp. 533–536.

100. Catfolis, T., 1993. A Method For Improving the Real-Time Recurrent Learning Algorithm. *Neural Networks,* vol. 6, pp. 807–821.

101. Zipser, D., 1989. A Sub-Grouping Strategy that Reduces Complexity and Speed up Learning in Recurrent Networks. *Neural Computing*, vol.1, pp. 552–558.

102. Hopfield, J. J., 1982. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *In: Proceedings of the National Academy of Sciences of the USA*, vol. 79, no. 8, pp. 2554-2558.

103. Tzafestas, S. G., 2002. *Computational Intelligence in Systems and Control Design and Applications*, Spinger, pp. 331-332.

104. Jaeger, H., 2001. The 'Echo State' Approach to Analysing and Training Recurrent Neural Networks. *German National Research Centre for Information Technology*, Tech. Rep. 148.

105. Rumelhart, D. E., McClelland, J. L., and the PDP Res. Group, 1986. Parallel Distributed Processing (PDP): Exploration in the Microstructure of Cognition. Cambridge, MA: MIT Press, vol. 1.

106. Scheler, G., 2004. Regulation of Neuromodulator Efficacy: Implications for Whole-Neuron and Synaptic Plasticity. *Progress in Neurobiology*, vol.72, no.6.

107. Scheler, G., 2004. Memorisation in a Neural Network with Adjustable Transfer Function and Conditional Gating. *Quantitative Biology*, vol 1.

108. Chen, C. T. and Chang, W. D., 1996. A Feed-Forward Neural Network with Function Shape Autotuning. *Neural Networks*, vol. 9, no. 4, pp. 627–641.

109. Piazza, F., Uncini, A., and Zenobi, M., 1992. Artificial Neural Networks with Adaptive Polynomial Activation Function. *In: Proceedings of. IJCNN*, Beijing, China, pp. II-343–349.

110. Uncini, A., Piazza, F. and Vecci, L., 1998. Learning and Approximation Capabilities of Adaptive Spline Activation Function Neural Networks. *Neural Networks*, vol. 11, no. 2. pp. 259-270.

111. Fiori, S., 2002. Hybrid Independent Component Analysis by Adaptive LUT Activation Function Neurons. *Neural Networks*, vol. 15, pp. 85-94.

112. Piazza, F., Uncini, A. and Zenobi, M., 1993. Neural Networks with Digital LUT Activation Function. *In: proceedings of International Joint Conference on Neural Networks (IJCNN'93)*, Nagoya, Japan, pp. 1401-1404.

113. Uncini, A., Vecci, L., Campolucci, P., and Piazza, F., 1999. Complex-Valued Neural Networks with Adaptive Spline Activation Function For Digital Radio Links Nonlinear Equalization. *IEEE Trans. Signal Processing*, vol. 47, pp. 505--514.

114. Foor, W. E. and Neifeld, M. A., 1995. Adaptive, Optical, Radial Basis Function Neural Network for Handwritten Digit Recognition. *Appl. Opt.* vol. 34, pp. 7545.

115. Ozbay, Y. and Karlik, B., 2001. A Recognition of ECG Arrhythmias Using Artificial Neural Networks. *In: proceedings of the 23rd Aunnual International Conference of the IEEE,* vol 2. pp. 1680 - 1683.

116. Lewis-Beck, S. M., 1995. *Data Analysis: An Introduction,* Sage Publications Inc.

117. Roadknight, C. M., Balls, G. R., Mills, G. E. and Palmer-Brown, D., 1997. Modelling Complex Environmental Data. *Journal Transactions on Neural Networks, IEEE*, vol.8.

118. Sivia, D. S., 1996. *Data Analysis: A Bayesian Tutorial,* Published by Oxford University Press.

119. Gelman, A., Carlin, J. B. and Stern, H. S., 2003. *Bayesian Data Analysis.* Published by CRC Press.

120. Wolpert, D. H., 1996. The Lack of a Priori Distinctions between Learning Algorithms. *Neural Computation*, vol. 8, pp. 1341-1390.

121. Mehrota K., Mohan C.K. and Ranka S., 1997. Elements of Artificial Neural Networks, In: The MIT Press, Cambridge, Massachusetts.

122. Moyle, S. A. and Watts, M., 2003. *Fuzzy Neural Networks (FuNN) in the Palm Environment.* Department of Information Science, University of Otago, Dunedin, New Zealand.

123. Vielma, J. P., Ahmed, S. and Nemhauser, G., 2010. Mixed-Integer Models for Nonseparable Piecewise Linear Optimization: Unifying Framework and Extensions. *Institute for Operations Research and the Management Sciences (INFORMS)*, Linthicum, Maryland, USA, vol.58, pp.303-315.

124. Bai, Y., Zhang, H. and Hao, Y., 2009. The Performance of the Back-propagation Algorithm with Varying Slope of the Activation Function. *Chaos, Solitons & Fractals*, vol. 40, issue. 1, pp. 69-77.

125. Apostol, T., 1974. *Mathematical analysis (2nd edition).* Addison Wesley, 1974.

126. Widrow, B. and Hoff, M. E., 1960. Adaptive Switching Circuits. *IREWESCON Convention Record*, pp. 96-104.

127. McClelland, J. and Rumelhart, D., 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. *Cambridge, MA: MIT Press*, vol.1.

128. Bourbaki, N., 1987. *Topological Vector Spaces.* Elements of Mathematics, Springer.

129. Boné, R., Crucianu, M., and Asselin de Beauville, J. P., 1998. Yet Another Neural Network Simulator (YANNS). *Proceedings of the Conference, NEURal Networks and their Applications (NEURAP'98)*, Marseille, France. pp. 421–424.

130. Robert, G. B., 2007. *An Object-Oriented Analysis and Design with Applications.* 3rd Edition, Addison-Wesley.

131. Budd, T., 1991. *An Introduction to Object-Oriented Programming.* Addison-Wesley.

132. Culwin, F., 1998. *A Java GUI: Programmer's Primer*, Prentice Hall.

133. Fisher, R. A., 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics 7*, pp.178-188.

134. Hoey, P. S., "Statistical Analysis of the Iris Flower Dataset", Univeristy of Massachusetts st Lowell, viewed 10 Auguest 2010, <http://patrickhoey.com/papers/Computer_Science/03_Patrick_Hoey_Data_Visualization_Dataset_paper.pdf>

135 Moriarty, D. E., 1997. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks.* Ph.D. thesis, Department of Computer Science. The University of Texas at Austin.

136. Cantu-Paz, E., 2003. Pruning Neural Networks with Distribution Estimation Algorithms. *GECCO. USA, 2003*, pp. 790-800.

137. Pavlidis, N. G., Tasoulis, D.K., Plagianakos, V.P., Nikiforidis, G., and Vrahatis, M.N., 2004. Spiking Neural Network Training Using Evolutionary Algorithms. *International Joint Conference on Neural Networks (IJCNN 2004)*, Budapest, Hungary.

138. Eldracher, M., 1992. Classification of Non-Linear-Separable Real-World-Problems Using Δ-Rule, Perceptions, and Topologically Distributed Encoding. *In Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing.* vol. I, ACM Press.

139. Luger, G. F., 2005. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* 5th Edition, Addison-Wesley.

140. Collobert, R. and Weston, J., 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. *In: proceedings of the 25th international conference on Machine learning*, Helsinki, Finland.

141. Heidi, H, and Yeung, T., 2004. Online Semantic Extraction by Backpropagation Neural Network with Various Syntactic Structure Representations. *AAAI 2004*, pp. 1042-1043.

142. Garside, R., Leech, G. and Varadi, T., 1987. *Manual of Information to Accompany the Lancaster Parsed Corpus.* Department of English, University of Oslo.

143. Tepper, J., Powell, H. M. and Palmer-Brown, D., 2002. A Corpus-based connectionist Architecture for Large-scale Natural Language Parsing. *Connection Science,* vol. 14, no. 2, pp. 93 – 114.

144. Tepper, J. A., 2000. *Corpus-based Connectionist Parsing.* PhD Thesis, Faculty of Engineering and Computing, The Nottingham Trent University, 2000.

145. Palmer- Brown, D., Tepper, J. and Powell, H., 2002. Connectionist Natural Language Parsing. *Trends in Cognitive Sciences,* vol. 6, no. 10, pp.437 – 442.

146. Pollack, J., 1990. Recursive Distributed Representations. *Artificial Intelligent*, vol. 46, pp. 77-105.

147. Elman, J., 1990. Finding Structure in Time. *Trends in Cognitive Sciences*, vol. 14, pp. 179-211.

148. Hasan, M., Manian, V. and Kemke, C., 2007. HCP with PSMA: A Robust Spoken Language Parser. *NeSy.*

149. Mayer, A. H. and Schwaiger, R., 2001. Evolution of Cubic Spline Activation Functions for Artificial Neural Networks. *EPIA* pp. 63-73.

150. Sunat, K. and Lursinsap, C., 2001. Highly Successful Learning Based on Modified Catmull-Rom Spline Activation Function. *In: proceedings of IJCNN2001,*vol 4, pp. 2783 – 2787.

151. Frey, P. W. and Slate, D. J., 1991. Letter Recognition Using Holland-style Adaptive Classifiers. *Machine Learning*, vol. 6, pp.161-182.

152. Slate, D. J., "Letter Image Recognition Data", viewed 2 September 2010, <http://archive.ics.uci.edu/ml/datasets/Letter+Recognition>

153. Schwenk, H. and Bongo, Y., 1997. Adaptive Boosting of Neural Networks for Character Recognition. *Technical report #1072,* Canada.

154. Asuncion, A, and Newman, D. J., 2007. *UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html],* Irvine, CA: University of California, School of Information and Computer Science.

155. Vamvakas. G., Gatos, B. and Perantonis, S. J., 2009. A Novel Feature Extraction and Classification Methodology for the Recognition of Historical Documents. *10th International Conference on Document Analysis and Recognition,* Barcelona, Spain.

156. Schapire, R. E., Freund, Y., Bartlett, P. and Lee, W. S., 1997. Boosting The Margin: A New Explanation For Effectiveness Of Voting Methods. *Proceedings of the 14th International Conference.*

157. Philip, N. S., Joseph, K. B., 2000. Distorted English Alphabet Identification : An application of Difference Boosting Algorithm. *CoRR.*

158. Partridge, D. and Yates, W. B., 1997. Data-defined problems and multi-version neural-net systems. *Journal of Intelligent Systems*, vol. 7, no. 1-2, pp.19-32.

159. Lee, S. W., Palmer-Brown, D. and Roadknight, C. M., 2004. Performance-guided Neural Network for Rapidly Self-Organising Active Network Management (Invited Paper). *Journal of Neurocomputing*, vol. 61C, pp. 5 – 20.

160. Lee, S. W. and Palmer-Brown, D., 2006. Phonetic Feature Discovery in Speech using Snap-Drift. *International Conference on Artificial Neural Networks, ICANN'2006, Athens, Greece, 10th - 14th,* pp. 952 – 962.

161. Lee, S. W. and Palmer-Brown, D., 2006. Modal Learning in A Neural Network. *1st Conference in Advances in Computing and Technology, London, United Kingdom,* pp. 42 – 47.

162. Kohonen, T., 1990. Improved Versions of Learning Vector Quantization. *Proc. IJCNN'90,* pp.545 – 550.

163. Carpenter, G., 1997. Distributed Learning, Recognition, and Prediction by ART and ARTMAP Neural Networks. *Neural Networks*, vol. 10, pp. 1473 - 1494.

164. Alimoglu, F., Alpaydin, E., 1997. Combining Multiple Representations for Pen-based Handwritten Digit Recognition. *ELEKTRIK: Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 9, no. 1, pp.1-12.

165. Alpaydin, E., Kaynak, C., Alimoglu, F., 2000. Cascading Multiple Classifiers and Representations for Optical and Pen-Based Handwritten Digit Recognition. *IWFHR*, Amsterdam, The Netherlands.

166. Garris, M. D. et al, 1991. NIST Form-Based Handprint Recognition System. *NISTIR* vol. 5469.

167. Rosenblattm F., 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, vol. 65, pp. 386 - 408.

168. Zhang, J. and Li, Z., 2005. Adaptive Nonlinear Auto-Associative Modelling through Manifold Learning. *PAKDD*, pp.599-604.

169. Chakraborty, B. and Chakraborty, G., 2002. A New Feature Extraction Technique for On-line Recognition of Handwritten Alphanumeric Characters. *Information Sciences*, vol. 148, issues 1-4, pp.55-70.

170. Liu, Q. and Wang, J., 2008. Two k-winners-take-all Networks with Discontinuous Activation Functions. *Neual Networks, 2008 Special Issue: Advances in Neural Networks Research: IJCNN07*, vol. 21, pp. 406 – 413.

171. Findlay, D. A., 1989. Training Networks with Discontinuous Activation Functions. *Artificial Neural Networks*, vol. 16, issue. 18, pp. 361 – 363.

172. Uncini, A. and Piazza, F., 2001. Blind Signal Processing by Complex Domain Adaptive Spline Neural Networks. *ICASSP '01.*

173. Vigliano, D., Scarpiniti, M., Parisi, R. and Uncini, A., 2006. Flexible ICA in Complex and Nonlinear Environment by Mutual Information Minimization. *In: Proc. of IEEE Machine Learning for Signal Processing, Maynooth, Ireland,* pp. 59-64.

174. Solazzi, M. and Uncini, A., 2004. Regularizing Neural Networks using Flexible Multivariate Activation Function. *Neural Networks,* vol. 17, pp. 247-260.

175. Vitagliano, F., Parisi, R. and Uncini, A., 2003. Generalized Splitting 2D Flexible Activation Function. *Lecture Notes in Computer Science, Sprinter-Verlag Heidelberg,* vol. 2859, pp. 165-170.

176. Solazzi, M. and Uncini, A., 2000. Artificial Neural Networks with Adaptive Multidimensional Spline Activation Functions. *In: Proc. of the IEEE-INNSENNS International Joint Conference on Neural Networks IJCNN2000, Como, Italy,* vol. 3, pp. 471-476.

177. Scarpiniti, M., Vigliano, D., Parisi, R. and Uncini, A., 2008. Generalized Splitting Functions for Blind Separation of Complex Signals. *Neurocomputing.*

178. Cherry, B. "The Thames Gateway: An introduction to the historical landscapes of the northern riverside", viewed 10 September 2010,
<http://www.helm.org.uk/upload/pdf/BCherry2.pdf?1283253165>

179. Diffie, W. and Hellman, M. "New Directions in Cryptography", IEEE Transactions on Information Theory, Vol. 22 Issue 6, pp. 644-654, 1976.

APPENDIX A: TERMINOLOGY GLOSSARY

The following is a summary of some common neural network terms. They have been used throughout this thesis.

Network Layers: A layer of a neural network is an array of nodes. A layer of "input" units is called ***input layer.*** Input layer is connected to a layer of "hidden" units or connected a layer of "output" directly in different network structures. However, the ***hidden layer*** does not connect to the environment. It is connected to a layer of "output" units which is called ***output layer.*** The activity of the input units represents the raw information that is fed into the network. Whereas the activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units. And the behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

Δf: Δf is the modification of the neuron's activation function that takes place during learning.

Δw: Δw is the modification of the neuron's weights that takes place during learning.

Activation: The signal that a unit sends, to either other units or the environment.

Activation Function: Function used to calculate a unit's activation from its input.

Classification: Problem in which patterns are to be assigned to one of several classes.

Delta rule: Learning rule based on minimisation of squared error for each training pattern.

Epoch: A single iteration through all patterns.

Generalisation: A term used to refer to how well a network performs on data on which it has not been trained.

Learning rate: A parameter that is usually set to a constant value before training. This parameter controls the amount by which a weight can change during a single update.

Linearly inseparable: Training patterns belonging to one output class cannot be separated from training patterns belonging to another class by a straight line.

Multilayer perceptron (MLP): A type of feed forward neural network that is an extension of the perceptron in that it has at least one hidden layer of neurons.

Pattern: Refers to a data record that is presented to a network.

Perceptron: The simplest type of feed forward neural network. It has only inputs and outputs, i.e., no hidden layers.

Supervised learning: A type of learning that can be applied when it is known to which class a training instance belongs.

Unsupervised learning: Learning that is used when the training instances do not have a known class.

Weight: A connection between two units.

APPENDIX B: PUBLICATIONS

[1] Palmer-Brown, D., S. W. Lee., Draganova, C. and Kang, M., 2009. Modal Learning Neural Networks. *WSAES Transactions on Computers*, vol. 8, no. 2, pp. 222 – 236.

[2] Kang, M. and Palmer-Brown, D., 2008. A Modal Learning Adaptive Function Neural Network Applied to Handwritten Digit Recognition. *Information Sciences*. 178(2008), pp. 2802-3812.

[3] Palmer-Brown, D., Kang, M. and Lee, S. W., 2008. Meta-Adaptation: Neurons that Change their Mode. *In: proceedings of the 9th WSEAS International Conference on Neural Networks (NN'08),* Sofia, Bulgaria.

[4] Kang, M. and Palmer-Brown, D., 2007. Snap-drift ADaptive FUnction Neural Network (SADFUNN) for Optical and Pen-Based Handwritten Digit Recognition. *In: 10th International Conference on Engineering Applications of Neural Networks (EANN'2007).* pp.247-253, Thessaloniki, Hellas, Greece.

[5] Kang, M. and Palmer-Brown, D., 2007. A Multi-layer ADaptive FUnction Neural Network (MADFUNN) for Letter Image Recognition. *In: the International Joint Conference on Neural Networks (IJCNN'2007).* Orlando, Florida, USA, pp. 2817-2822.

[6] Kang, M. and Palmer-Brown, D., 2006. A Multi-layer ADaptive FUnction Neural Network (MADFUNN) for Analytical Function Recognition. In: the International Joint Conference on Neural Networks (IJCNN'2006). Vancouver, Canada, pp. 1784- 1789.

[7] Kang, M. and Palmer-Brown, D., 2005. An Adaptive Function Neural Network (ADFUNN) for Function Recognition. *In: the 2005 International Conference on Computational Intelligence and Security (CIS'2005).* Xi'an, China.

[8] Kang, M. and Palmer-Brown, D., 2005. An Adaptive Function Neural Network (ADFUNN) Classifier. *In: the second International Conference on Neural Networks & Brain (ICNN&B'2005).* Beijing, China, pp586-590.

[9] Kang, M. and Palmer-Brown, D., 2005. An Adaptive Function Neural Network (ADFUNN) for Phrase Recognition. *In: the International Joint Conference on Neural Networks (IJCNN'2005).*Montréal, Canada, pp593-597.

[10] Palmer-Brown, D. and Kang, M., 2005. ADFUNN: An adaptive function neural network," *the 7th International Conference on Adaptive and Natural Computing Algorithms (ICANNGA'2005).* Coimbra, Portugal, pp.1-4.