1. Introduction

1.1 Motivation

Traditional Artificial Neural Networks[1] can not classify in non-stationary environments accurately. In such systems, although the class to be learnt may be understood, the locations in the input space of these classes are unknown. Non-stationary systems may also change the number of classes contained in the environment – by adding new or removing old classes. Many real world systems are non-stationary, but are bounded such that the number of classes in the problem is restricted. This is because a Neural Network requires examples of all the classes that it will be required to classify *prior to training*. If the system is truly non-stationary, then it is not possible for the Neural Network to learn all the different inputs as they will not be present before training begins.

The work described in this thesis attempts to solve this problem by developing a novel Neural Network that can grow and shrink its topology to accommodate non-stationary environments. The new algorithms presented here are to be tested on several environments for exploration of the networks' controlling parameters, as well as operating on a real Radar Data set (an example of a real world non-stationary environment).

A novel Neural Network is required for the representation of the temporal structure of these non-stationary systems. This enables the temporal structure of a system to be investigated. Once a structured environment has been sampled for a number of iterations, the resulting network can be used to predict the next input.

The aim of this chapter is to familiarise the reader with Neural Networks and existing work in this field so that the algorithms presented later can be considered in context

[1] From this point, Artificial Neural Networks will simply be called Neural Networks. Biological Neural Networks are identified as such.

and to illustrate how typical Neural Network techniques fail to solve the problems given above. It begins by giving an overview of Neural Networks, their features, benefits and short comings. It then specifies two sorts of static Neural Network, the Multilayered Perceptron (MLP) and Self Organising Map (SOM). The description of the SOM is taken further as it is the base of this work. Temporal Neural Networks are then described and are shown to have the same shortcomings as the static Neural Networks. Growing and Pruning networks are extensions to static Neural Networks and go some way to solving some of the problems above. Following this is a brief look into how hybrid Neural Networks may be employed to solve some problems before finally defining a name to a type of network that can solve these problems, the Dynamic Neural Network and an example that fits these criteria, GNG-U.

1.2 Neural networks

Neural networks are computational structures inspired from structures in the brain. The brain is complex, non-linear and massively parallel. The brain's structure is created in two ways. Initially, there is a makeup of neurons from birth, a set of hard-wired processing systems that are present from the outset. Secondly, the brain adapts by altering its structure of *neurons* such that new behaviours can be learnt. This is a learning process through experience. The phenomenon that allows the alteration of the neurons to learn from experience is called *plasticity*.

In the simplest of terms, a Neural Network is a computational model of some property or function of the human brain. The network can be implemented in hardware (Schoenauer et al, 1998) or software. The following definition is offered (Haykin, 1999):

> *A Neural Network is a massively parallel distributed processor made up of simple processing units which has a natural propensity for storing experiential knowledge and making it available for use.*

This definition is general enough to include all variants of Neural Networks while at the same time it is not too vague to be useful. The storing of experiential knowledge in an orderly way, such that it may be of use, is called *learning* (p.51, Haykin, 1999). In most Neural Network architectures, this learning process takes the form of the alteration of synaptic weights, such that future presentations of an input class yield similar results. More elaborate processes also include the removal or addition of neurons to the network structure (see Section 1.6).

Neural networks gain additional applicability through their massively parallel, distributed structure and also their ability to generalise. Generalisation is the ability to produce a reasonable output for inputs that were not present during the training phase. However, Neural Networks do not solve every problem and work best when integrated into a larger system (Lawrence, 1997).

Typically, Neural Network architectures consist of a data structure of *neurons* (nodes) and *weights* (links, edges or arcs). The neurons hold some capacity to process information whereas the weights contain numerical values that are modified during training.

Neural networks offer the user the following capabilities (Haykin, 1999):

- *Nonlinearity*. As neurons can be inherently non-linear, the network as a whole is non-linear. This is a very important property, especially if the system providing the input is essentially non-linear.
- *Robust data acceptance*. Natural data is not always low-order and well defined. Under these conditions, Neural Networks are more effective than traditional algorithms.
- *Input-Output Mapping*. When using a supervised learning algorithm, a Neural Network can be used to provide an output given a certain input. The network is said to have created an input-output map.
- *Adaptivity*. A network can adapt its synaptic weights to any specific environment and can be retrained by initialising the same network architecture.
- *Confidence*. A network can often give its confidence in the result it has produced.
- *Fault Tolerance*. A Neural Network's distributed structure is robust in that its architecture can be damaged and the network's operation will degrade gracefully.
- *VLSI Implementability*. Due to a parallel design, a Neural Network is an excellent candidate for implementation into hardware.

However, Neural Networks also suffer from the following disadvantages.

- *Topology preset.* Before any training has begun, the topology of the network needs to be set. Although there are some heuristics for estimating this topology, given statistical information regarding the input set (p.178, Haykin 1999), they operate in an ad-hoc application specific fashion. Some methods

of altering network topology during training are noted later (See Section 1.6 below).

- *Exemplar patterns required.* As a Neural Network interpolates from given data, good examples of patterns belonging to the different classes are required before training.
- *Stationary Data Set.* The classes available at training time are the only classes that can be accepted by the network. New classes that arrive after the training period will be classified incorrectly, provided the new class is outside the range of the outputs from existing classes. Further, old classes that no longer appear in the input space after training will still be stored by the network. This is a problem as the network will be using weight storage for classes that will never be shown to the network again.
- *Temporality Ignored.* When training a Neural Network, it is common practice to randomise the order in which the patterns are presented to the network, removing any structure that the dataset may have contained. Thus, the order in which the patterns arrive at the input is ignored and potentially useful information is lost.
- *Overfitting.* If a Neural Network is shown many examples of a class, it may over-fit the data. This is where the network has memorised the input data set and thus can no longer generalise.

1.3 Static Neural Networks

1.3.1 Feedforward networks

Feedforward networks are a type of Neural Network that has at least input nodes, hidden nodes and output nodes. Learning is typically supervised (Sutton and Barto 1998), in that the desired output is shown to the output of the network at the same time as the input and then learning alters weights so that a future presentation of this input will give the desired output. An example of a feedforward network is the Multi-Layered Perceptron (MLP – Rosenblatt 1958). An MLP's topology is normally a number of nodes arranged in layers (Figure 1.1). Typically, there is an Input, Output and a Hidden layer. Between each of these layers are a series of synaptic weights. Any input signal propagates through the network giving an output on the last layer.

In a typical MLP, the output of one neuron connects to the inputs of every neuron in the next layer; it is said to be fully connected.

The classic MLP is trained using the *error back-propagation algorithm* (Werbos, 1974 Rumelhart et al 1986) which is a generalisation of the Least Mean Square algorithm. Essentially, error back-propagation passes twice through the network on each learning cycle. Firstly, the forward pass uses an input vector to propagate activations through the network. From this, the error between the actual output and desired output can be calculated. This error is then used in a backward pass where the weights between layers are altered. The result of this is that future presentations of the input vector in question are more likely to give the desired output.

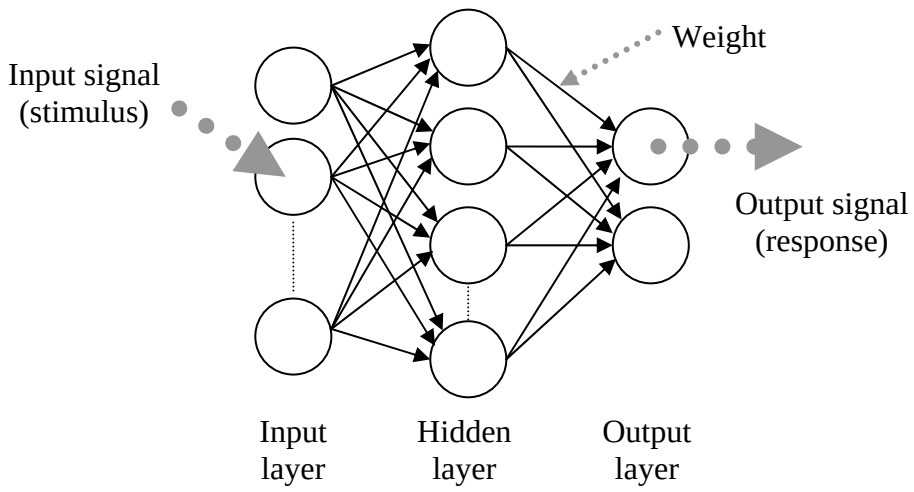

Figure 1.1 Architectural graph of a multi-layered perceptron (MLP) showing three layers.

The power of the MLP is gained from the distributed non-linearity of the neurons, the hidden layer which allows the network to learn complex tasks and a high degree of connectivity. However, these complexities lead to a lack of understanding in the operation and features of the network. Theoretical understanding of the whole network's operation is difficult due to the high connectivity of non-linear units (pp. 157 Haykin 1999).

Applications for the MLP include pattern recognition (Lippmann, 1989) and function approximation (Hecht-Nielsen 1987, Funahashi 1989, Hornik et al 1990 and Poggio

and Girosi 1990). All these applications were successful and outlined some of the limitations of MLPs, especially the requirement for a well selected training input set.

The error back-propagation algorithm is an efficient, polynomial, supervised learning technique. However, the MLP can only interpolate between patterns that it has learnt and this set is stationary. Although generalisation does give the MLP some ability to cope with corrupted patterns, it can not extrapolate to accept new classes outside of the training set. Also, the MLP suffers from the requirement of having its topology chosen and fixed before training begins. For any problem where little is known about the statistical relationships between the classes, it can be very difficult to use heuristics to decide on the size of the hidden layer(s) and an iterative process of trying different sizes of network normally takes place (Fahlman and Lebeire 1990). Back-propagation is a notoriously slow algorithm, taking many iterations to converge, even when applying heuristics to accelerate the process (pp.182, Haykin 1999). Alternatives include Quickprop (Fahlman, 1988) that reduces the number of passes required by the network by using a second order gradient descent algorithm.

The learning rate of the MLP must also be considered carefully. If it is too large, the Neural Network may never converge to a local minimum and if it is too small then training times can take many thousands of iterations. A number of schemes have been introduced to deal with this problem such as momentum (Rummelhart, 1986), Conjugate gradients (Watrous, 1988), Dynamic step sizes (Franzini, 1987) and a second order gradient descent also by Fahlman (1988).

1.3.2 Self Organising Maps

Self Organising Maps (SOMs, Kohonen, 1982) are structures that do not have input and output neurons. The neurons are arranged in a multidimensional lattice of usually low dimension for easy visualisation (Figure 1.2). The SOM utilises competitive learning: each input is shown to the network and a winning node is chosen. Over many training cycles, each neuron is tuned such that it better represents a point in the input space. This tuned neuron becomes ordered, with respect to the neurons close to it in the network, by altering the neurons surrounding it such that they are more similar to the winning node at each iteration. This notion of *neighbourhood* means

that features in the input space become characterised in the topographic map of the neuron space. In this manner, the spatial patterns of neurons represent the statistical space of the input space without external interference. This means that the locations of the neurons on the topographic map can be used to identify properties of the search space. If, for example, two neurons are close to each other then the points in the input space that they represent are also close together. Also, a larger cluster in the input space will be represented by more neurons than a smaller cluster.

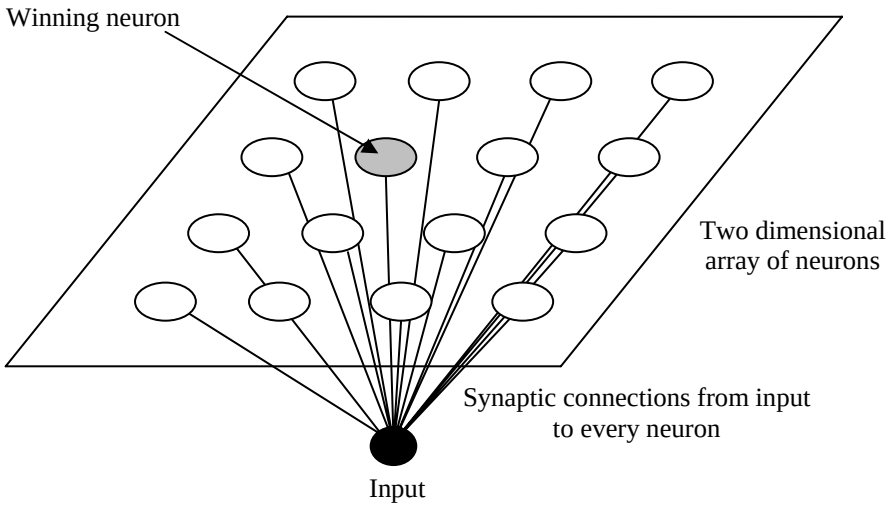

Figure 1.2 An architectural model of Kohonen's Self Organising Map (SOM) showing the winning node (focus), given an input.

As a neural model, the SOM is motivated by the manner in which the brain is organised topographically. Sensory inputs such as tactile (Kaas et all, 1983), visual (Hubel and Weisel, 1977) and acoustic (Suga, 1985) are ordered onto the cerebral cortex in a topological manner. In the cerebral cortex, the use of computational maps offers the following properties (Knudsen et all., 1987):

- At each stage of representation, each incoming piece of information is kept in its proper context.
- Neurons dealing with closely related pieces of information are close together so that they interact via short synaptic connections.

One important point that emerges at this point is the *principle of topographic map formation*, which may be stated as (Kohonen, 1990):

The spatial location of an output neuron in a topographic map corresponds to a particular domain or feature of data drawn from the input space.

Kohonen's SOM (1982) was not designed to explain neurobiological details, instead it concentrated on a computationally tractable model that captured the essential features of the brain's maps. The SOM has been effectively used for pre-processing patterns for their recognition and projecting and also visualising high dimensional signal spaces on a two dimensional display (Simula et al, 1999).

1.4 Kohonen's Self Organising Map in Detail

This section details the principles of Kohonen's SOM, while simultaneously introducing the notations used later when describing the novel work and concepts. The basic SOM (Kohonen, 1982) uses adaptation as well as non-parametric regression techniques to achieve its goal. Formally, a SOM may be described as a non-linear, ordered, smooth mapping of high-dimensional signal space onto a regular, low dimensional array. Notations have been changed from Kohonen's original nomenclature to facilitate later comparisons with novel algorithms presented later (Chapters 2 and 4).

Firstly, assume that an input, u, is a vector of real numbers with a size of n, thus:

$$u = \left[\xi_1, \xi_2, ..., \xi_n\right]^T \in \Re^n \tag{1.1}$$

Where T represents transpose.

Associate each element in the regular SOM array with a regular vector x_i called the *reference vector* which is a vector of real numbers.

$$x_i = \left[\mu_{i1}, \mu_{i2}, ..., \mu_{in}\right]^T \in \Re^n \tag{1.2}$$

Assuming a general distance measured between u and x_i, $d(u, x_i)$, the image of the input vector u on the SOM is described as the array element x_c that best matches the input vector. Thus, the index, i, of the best match element can be defined as:

$$c = \arg\min_i \{d(u, x_i)\} \tag{1.3}$$

At this stage, the representation moves away from the classic vector quantisation (VQ) algorithm in that the SOM needs to be topologically ordered (this is not a facet of the typical VQ algorithm). To achieve this goal of topology, a notion of *neighbourhood* is introduced. The neighbourhood, N_i refers to all the nodes within a certain radius around a node x_i.

Before any recursive processing is carried out, the reference vectors of the two-dimensional SOM must be initialised. Normally, random numbers are used to show that from an arbitrary starting state, the nodes will become two-dimensionally ordered values. This is the most simple (and perhaps powerful) effect of self-organisation. Let u be an input vector that needs to be learnt by the network. The first step is to find the element which is the best match for input u. The best matching node is normally found by calculating the Euclidean Distance $\|u\text{-}x_i\|$ and is called the *focus*, z.

$$z = \arg\min_i \{\|u - x_i\|\} \tag{1.4}$$

Once the focus is found, the neighbourhood, N_i, is updated such that the nodes in the neighbourhood learn from the input. This results in a local smoothing effect of the reference vectors within the neighbourhood. Local smoothing means that there is a gradual (or smooth) change in the values of the vectors as one scans across the network. Over a number of training presentations, this local organisation leads to a global ordering. Given that the initial values of the nodes, $u_i(0)$, are random the learning process can be viewed as thus:

$$x_i(t+1) = x_i(t) + h_{zi}(t)\left[u(t) - x_i(t)\right] \tag{1.5}$$

where t = 0,1,2,… is an integer, the discrete time index. In the smoothing process, h_{zi} plays the role of the *neighbourhood function* and for convergence, it is necessary for $h_{zi} \to 0$ when $t \to \infty$. Typically the neighbourhood function is of the form:

$$h_{zi} = h\left(\left\|r_j - r_i\right\|, t\right) \qquad r_j \in \Re, r_i \in \Re \tag{1.6}$$

Where r_j and r_i are the location of the nodes j and i in the array respectively. By increasing $\|r_j - r_i\|$, $h_{zi} \rightarrow 0$. There are two widely applied neighbourhood functions. The first, is a neighbourhood set of nodes around the focus, simply being those n neurons closest to the focus. This is a rectangular shaped function and can be seen in Equation (1.6). The other is a smooth kernel that can be written in terms of a Gaussian function, Equation (1.7).

$$h_{ci} = \alpha(t) \cdot \exp\left(-\frac{\left\|r_j - r_i\right\|^2}{2\sigma^2(t)}\right) \tag{1.7}$$

where $\alpha(t)$ is a scalable learning rate factor and $\sigma(t)$ defines the width of the kernel. Both $\alpha(t)$ and $\sigma(t)$ are monotonically decreasing functions of time. The Gaussian function is often preferred as it is more biologically appropriate than a rectangular neighbourhood function.

Other functions include *sine* based functions such as the *sinc* function (see Chapter 2).

The initial selection of the width of the neighbourhood is very important. If the neighbourhood is too small then the map will not be ordered globally, but instead order into localised pockets with large discontinuities. The neighbourhood radius is often set as being half the width of the network. If the initial values of the network were set arbitrarily then $\alpha(t)$ should be set close to unity, decreasing monotonically. With large networks, it may be necessary to minimise the learning time, making the selection of the parameter crucial. Once the network has been trained, information can be derived by inspecting the visual structure of the network or by finding the focus for other known inputs. No further adaptation is performed after training has finished and thus any changes in the input space after this point will not be reflected in the network.

The training of the network can be seen as two phases, firstly there is the self-organising or ordering phase. This is where the network initially orders the patterns in the network topologically. At this point, the learning rate is high and so is the neighbourhood width. The second phase is the convergence phase that fine tunes the map to the data, providing an accurate statistical quantification of the input space. For this to occur, the neighbourhood width and the learning rate should be kept small. These requirements often lead to non-linear decay functions, which reduce learning rates, further complicating the process of initialisation.

Improved convergence can be achieved by using alternative initialisations. Rather than randomly generating the initial reference vectors, a linear scheme dependent on the known training data can be used. This can also aid a smooth approach to the equilibrium point (where further alteration in the topology does not improve convergence) and will make better use of the neurons in the network. Given that both the width of the neighbourhood function and the learning rate are decreasing during the training phase of the network, it is not clear how the learning rate should be optimised (neighbourhood sizes can be set arbitrarily large and normally need no tuning).

Although the basic SOM has been widely used (Simula et al, 1999), perhaps due to its simplicity of structure and yet highly complex emergent behaviours, there are some issues with the initialisation of the network. Questions such as ‘How many nodes to use?’ and ‘How many training steps?’ are often posed. Although there are heuristics to govern the creation of the network (Kohonen, 2001), no analytical theory is in place to pre-define the structure or parameters of the network before training. Network design heuristics also suggest that:

- Hexagonal lattices are preferred as not to give any preference to vertical or horizontal neighbours.
- When training samples are few then they may be used reiteratively in a cyclic or randomly permuted order.
- Rare cases should be duplicated and enhanced.
- Some exemplar patterns should be forced onto a specific location on the map.

- The quality of learning should be monitored by measuring the mean of $||u-x_z||$ when reiterating the input set.
- The network should be trained several times with random initialisations and then the best result chosen.

Like all static networks, the SOM cannot accept any new data once the training cycle has finished. This leads to the SOM only being able to operate on non-stationary data sets where the number of different classes is fixed and finite. One could try retraining by using a large learning rate, but the existing data would be corrupted by the arrival of the new pattern. Another assumption that reduces the effectiveness of the SOM is that the input data can be well represented in a grid structure. Intuitively the representations required need to be of low dimensionality (1,2 or 3) for the user to understand, but a hexagonal grid is not necessarily the best format for visualisation of a dataset.

Although well proven heuristics exist for estimating the number of training inputs required for a certain size of network, there is no heuristic for determining the size of the network, given the input data. This heuristic, should it exist, would require knowledge of the statistical densities and regularities of the input space before training has begun. As the SOM is often used for finding these properties, a catch 22 dilemma is created. For the most part, researchers use arbitrarily large networks for the data and rarely vary the size once the network is working. Only when the efficiency of the network is of particular interest is the network size properly investigated. Size is not the only issue; the shape of the SOM can be varied adding choice as well as complexity to the initialisation procedures.

Another problem affecting the SOM is the requirement of exemplar patterns before training. The input set is often artificially fused with additional versions of an exemplar pattern (by repeating patterns that are known to be good examples of the class it belongs to), such that there will be assuredness that the network properly classifies it. With online systems, one is not afforded such a luxury, the patterns are accepted only once by the network. In real-world systems, it can be difficult to make sure that the training set includes exemplar patterns of each class. As dimensionality increases, this problem is exacerbated.

The SOM also ignores the temporality of patterns. Often, the order or frequency in which the patterns are presented to the network gives some important information. One example of this is the arrival of radar pulses. Each radar pulse has its own radio frequency and bearing, but it may also be pulsing in such a way to make classification by normal systems difficult. A network with a grasp of time and ordering may make better use of this information.

1.5 Temporal Neural Networks

Temporal Neural Networks are those networks that use time series blocks within data streams. Although they could be seen as dynamic as each pattern in the input set is a varying set of numbers, the number of different classes in the input space is stationary and the topology of the network does not change after training. Also, exemplar patterns are required before training. This not the case for Dynamic Neural Networks (see Section 1.8). A temporal Neural Network is often required to do some of the following (Dorffner, 1996):

- Forecasting of future developments of a time series.
- Classification of time series, into one of several classes.
- Description of a time series in terms of the parameters of a model.
- Mapping of one time series onto another.

The first task has the most real world applications. From economics to weather prediction, the ability to predict the future with some certainty is highly desirable. In most cases, the input to a temporal network requires pre-processing to ensure optimal processing. This could be used to remove the existence of linear trends (an undesirable linear feature with a large time constant). To perform this de-trending, one must have *a priori* information regarding the operation of the system. The input may then be transformed to remove the trend. This often occurs in sensor drift, where the physical properties of the sensors slowly change (or degrade) over time giving different results to when they were first installed. In cases such as this, it is normal for the system design to be able to predict the performance of the physical system.

Encoding time into a Neural Network can be achieved in a number of ways (Chappelier and Grumbach, 1994). Firstly, time can be used to pre-process the spatial input set to create a spatio-temporal input set. This can be done by adding a dimension which refers to time or by altering the existing spatial dimensions to include frequency information.

Time can be introduced into a Neural Network on a number of levels. Time can be used as an index to network states. By saving the state of the network at each time index, one can observe the process of the Neural Network, but this does not solve any time related problems. Further to this, time can be introduced by using time delayed weights in the propagation of Feedforward networks but this only allows the network to classify time series more efficiently. The deepest integration of time into a Neural Network model is by adding time at the neuronal level. This allows inputs to a neuron to be asynchronous. The firing of neurons in this case is achieved by a differential thresholding function.

In (Chappelier et al, 1996), a self-organising map was used for the detection of written signatures. This was achieved by transforming the input space in to a spatio-temporal input space. Each signature was encoded into a series of points in time, which then comprised the input vector. Although this network does encode dynamically temporal information, the network itself is not dynamic as such, as it does not change topology or will not accept more patterns after training.

An example of time being introduced at the neuron level is presented by Klaassen (1991) who uses connections with temporal weights. The neuron model derives from biology and consists of modelling the neural propagation with differential equations. This allows neuronal properties such as the time when neurons are unresponsive to inputs (the refractory period) to be modelled in a more biologically plausible manner. Time is used mathematically to model the delays in the network, but this has limited success at solving problems where time is a factor, such as in the classification of time series. The introduction of time at the neuron level often leads to dynamical properties and complex behaviours implying oscillations (Horn and Usher, 1991).

More biologically inclined works are presented by Abbott and Kepler (1990) and Rinzel and Ermentrout (1989). This research focuses on the introduction of time into neuron models based upon detailed biological observations. The underling principle involves a ‘integrate and fire’ model or ‘leaky integrator’. In this model, the neuron sums inputs over time and when this sum is greater than some threshold, the neuron changes state. Each neuron holds a separate threshold. This research tends toward the understanding of biological issues rather than solving engineering problems (Tabak, Rinzel et al, 2001), but can be of use in engineering tasks (Gerstner 1995).

A form of Neural Network which is particularly attuned to dynamical systems is the recurrent Neural Network (RNN). Recurrent networks are Neural Networks that have feedback loops. A feedback loop can either be local or global and normally contains a unit delay. In this manner, they have memory of the last iteration of the network’s operation. One such network is the Hopfield Network (Hopfield, 1982) which takes the time delayed outputs of the previous iteration and feeds them into the inputs of the current iteration. The task of the Hopfield network is one of energy minimisation and is rarely used for the analysis of systems where temporality of the input data is of interest. There are many different forms of RNN, with detailed analysis being beyond the scope of this work as RNN do not form the basis of the work presented. The architectures include tapped delay line inputs (Hopfield, 1982), recurrent multilayer perceptron (Jordan, 1986 and Elman, 1990) and second order networks (Omlin and Giles, 1996). In all of these networks, feedback is applied with tapped delay line memories. In principle, a recurrent Neural Network with global feedback can learn the underlying dynamics of a non-stationary system. Also, the network can track statistical variations within the data set.

However, RNNs are only capable of temporal processing. The recognition or learning of a time series does not necessarily unlock the temporal complexities that are often of interest because it does not focus on the order of classes in the input set. The input class set is assumed to be fixed in size and the networks are topologically fixed in size for the entire operation. There does not exist general methods for constructing RNNs for a given data set, heuristics only exist for specific applications (Frasconi et al. 1992).

Another modified MLP was the Time Delay Neural Network (TDNN) by Lang[2] et al (1990). This architecture implemented a sliding window across the input time series. The sliding window is shifted along the time series at each step, using the contents of the window at each step as the input. The original context was single word speech recognition at which it performed adequately. Improvements in performance could be made if the frequency spectrum of the input sample were included, rather than the raw time series data.

The first temporal SOM is the Temporal Kohonen Map (TKM, Chappell and Taylor, 1993) which is a modification to Kohonen's SOM algorithm. In the TKM, each unit within the network acts as a difference equation that defines current activity as a function of previous activations and the current input. This is often called a 'leaky integrator' or exponential decay storage. This modification not only allows the network to separate similar patterns in the input space but it is also capable of giving context to patterns appearing in sequences. The modelling of the units in this way has some biological standing, as neurons retain an electrical potential on their membranes that eventually decays.

The TKM was then further modified by Koskela et al (1998), terming it the Recurrent Self Organising Map (RSOM). Where the TKM had a decay function on the output of each unit, the RSOM used a leaky integrator onto the input. By feeding back a vector (rather than scalar), a direction of error could be exploited during the weight update. The algorithms appear identical except in the manner in which the optimisation occurs. Further analysis (Varsta et al., 2000) shows that the RSOM is superior in the simple derivation of consistent update rule. Both TKM and RSOM algorithms are used for temporal sequence classification such as detecting the difference between normal and epileptic EEG signals. Little information is given about the manner in which the topology of the network is generated. Also, the data sets that the network is trained on are stationary, new clusters can not be formed after the training phase.

[2] Lang K. L. is not related to Lang R.I.W., the author of this thesis.

Another temporal SOM technique involves using a second map layered with the first so that the second map tries to capture the spatial dynamics of the primary (Kangas, 1990). By doing this, the second map becomes a temporal clustering of the first.

Although all the networks capture a degree of temporal information, they are not temporal themselves. The training is done on time series information, but the number of clusters in the training set is fixed. The level of dynamic processing ends at the time series level. Topologies are static and defined before training. Time is broken into chunks and then learnt or classified. This has many applications (as mentioned earlier) but does not solve the problem of dynamic class sets as new clusters can not be formed by the Neural Network.

1.6 Growing and Pruning of Neural Networks

This section deals with Neural Networks which can determine (to a degree) their own topology. This is done in two ways:-

- *Growing* - Incremental networks are those networks that start with a small amount of neurons and weights and build into larger networks as new information is presented to the network. The main advantages for any type of growing Neural Networks are:
    - No need to estimate the size, depth or connectivity pattern of the network in advance.
    - Operation of the network after training is quicker as the size of the network is tuned to the input data.

The additional implementation complexity of these networks does not impact the computation complexity, as the algorithms given below have been shown to outperform their static counterparts (Fahlman and Lebiere 1990; Fritzke 1994).

- *Pruning* - Pruning networks begin with a large number of neurons and then remove those whose activation levels are statistically low.

Both paradigms tend to modify or extend the existing Neural Network algorithms. Often, this extension leads to a network architecture that is far removed from the

original. This section will deal with growing and pruning techniques separately, detailing with the more important algorithms in the field.

This area of research is not to be confused with growing schemes that use genetic operators that function on the genotype and on the phenotype to create the network (see Section 1.7). Although the topology of the network is set by a genotype, the network does not change shape after creation given a varying environment. These genetically created networks, are often said to be growing networks but they do not demonstrate the behaviour as given above (Nolfi and Parisi, 1995). The application of genetic operations with Neural Networks is discussed in more detail later.

1.6.1 Growing Networks

This section covers both growing feedforward networks and growing self organising maps.

1.6.1.1 Growing Feedforward Networks

The first algorithm investigated is the popular Cascade-Correlation algorithm of Fahlman and Lebiere (1990). Cascade-Correlation (CC) is a supervised learning algorithm inspired by the error-back propagation algorithm. CC begins with a minimal network topology, then automatically trains and adds new hidden neurons one by one, creating a multilayered structure. Once a hidden unit has been added to the network, its input weights are frozen, thus becoming a permanent feature detector within the network. This feature detector can either produce outputs or go on to create more complex feature detectors.

CC outperforms a typical MLP in training (Fahlman and Lebiere 1990). With an MLP, the backprop learning algorithm requires a learning rate to be set. CC outperforms the typical modifications to the error back-propagation algorithm (see Section 1.3.1), because backprop requires a reverse pass through the network, while CC uses a forward pass only learning rule much like that of gradient descent (pp.163, Haykin 1999). The second problem with a typical MLP is one of *moving targets.* In this problem, each unit of the hidden layer is trying to evolve into a useful feature detector that will play a role in the network's overall operation. This task is

overcomplicated as there are other units also changing at the same time. The hidden units do not communicate directly and thus jostle for position within the network which takes much iteration to become stable. CC removes this problem by fixing the input weights when the unit is placed into the network, thus each neuron becomes a feature detector and remains as such.

In addition to the global advantages, CC can partition a high dimensional input space without a dramatic slowdown as occurs with the error back-propagation algorithm. CC can also be used for incremental learning, where more information can be added to the network at a later date. Once added, a feature detector remains unchanged in the network. Also, at any one time, only one layer of the network is trained, so the results from the rest of the network can be cached, speeding up the process. CC is not fully dynamic, as although it can deal with more classes after the initial training has finished, it can not remove parts of its representation pertaining to those classes that are no longer part of the input space. Doing so would require the entire network to be retrained.

The Recurrent Cascade Correlation algorithm (RCC, Fahlman 1991) was an addition to the original algorithm. The modification included local time-delayed self-loops and the necessary alterations to the learning algorithm. This gave RCC state memory. This method was used to recognise simple temporal grammars (such as the detection of Morse code characters). These modifications are very similar to the recurrent loops as suggested in Elman's work (1988). However, the RCC is restricted in the sense that the local recurrent loops are insufficient to represent some finite state automata. This problem was noted and addressed (Giles et al, 1995) by adding fully recurrent links into the network. This addition leads to longer training times, but enabled the network to generally represent any finite state automata. Although RCC can classify time series and is a constructive method, RCC is not purely dynamic as it is not able to forget information that is no longer part of the input set. This would lead to very large networks, as old classes, which are no longer part of the input space any longer, would still be stored.

Another algorithm inspired by the CC family is TACOMA (TAsk decomposition, COrrelation Measures and local Attention neurons) by Lange et al (1994,1997).

TACOMA differs from CC by using a localised routing algorithm and an activation function that combines the characteristics of radial basis functions (RBF, a feedforward network, similar to the MLP, which uses hyper-ellipsoid functions in three non-homogeneous layers (Broomhead and Lowe 1988)) with sigmoid units. The outcome of this network is that different areas of the network reflect different areas of expertise (similar to a Kohonen SOM). Although comparisons with CC are favourable, most tests are done with task decomposition problems, not general tasks. Also, the added computational complexity of the algorithm is not explicitly stated. It is, however, another example of a CC-inspired growing algorithm.

1.6.1.2 Growing Self Organising Maps

The majority of work on growing self organising maps is by Bernd Fritzke, who has created a family of networks based all on the SOM (Kohonen, 1982). As this work is most relevant to the research presented here, the family of algorithms will be dealt with in some detail. For an overview of his work see Fritzke (1996). All of Fritzke's algorithms share the following properties:

- The network is a structure consisting of nodes and edges connecting the nodes.
- Each node represents a point in the input space.
- Adaptation is done by generating an input signal, finding the node whose reference vector is most similar to the input signal and then moving the winning node and those in the topographic neighbourhood nearer to the input signal.
- A local error signal is produced for every adaptation step.
- This error determines where new nodes are to be placed in the network.

From this, the first algorithm under scrutiny is the Growing Cell Structures (GCS) algorithm (Fritzke, 1994). The GCS model has a structure of *hypertetrahedrons* with a predetermined dimensionality. A k-dimensional hypertetrahedron is special amongst polyhedrons because it has only $k+1$vertices. The model is initialised with one hypertetrahedron. The adaptation steps then continue as above. After a number of these adaptation steps, the node with the highest accumulated error is found and then a decision is made as to whether to add a new node. New nodes have a

dimensionality of *k*. Since the GCS model has a fixed dimensionality, it realises a dimensionality reduced mapping from a potentially high dimensional input space.

Secondly, the Growing Grid (GG) model (Fritzke, 1995a) uses a hyper-rectangular structure. To keep the structure rectangular, any addition of nodes requires the insertion of a complete row or column. The adaptation is done in exactly the same manner as above.

Lastly, the most interesting of the algorithms is the Growing Neural Gas algorithm (Fritzke, 1995b). In this model, there are no explicit constraints on the graph. The graph is continuously updated using competitive Hebbian learning (Martinetz, 1995). GNG creates a link between the winning and second winning node at each adaptation step. After a number of adaptation steps, the largest error (of winner and second winner) is calculated and then a node added. The topology of GNG reflects the topology of the input and often has high dimensionality in the neuron space, making it difficult to visualise.

The GNG algorithm is as follows, for clarity the parameter names have been omitted:

1. Start with two nodes at random positions in the input space.
2. Generate an input signal
3. Find the nearest node and the second nearest node.
4. Increment the ‘age’ on all edges emanating from nearest node.
5. Add the distance between the input and the nearest node to sum an accumulative error variable within the node.
6. Move the winning node and all its topographical neighbours nearer to the input by some fraction.
7. If the edge between the nearest and second nearest node exists, set the age of this link to 0.
8. Remove all edges older than some threshold.
9. If the number of input signals is an integer multiple of some parameter, insert a new node using the accumulated error within each unit.
10. Decrease all error variables in the nodes by multiplying by a parameter.
11. If the stopping criterion has not been met, repeat from step 2.

The GNG algorithm has been widely used for mapping topographical features of an unknown input space. This is only really useful in 2 or 3 dimensions, after which visualisation becomes very difficult to achieve. For clustering and visualisation of unknown distributions, GNG works very well (Cheng and Zell 2000). GNG was extended to create GNG-U, presented at the end of the Chapter.

Another growing extension to the SOM is the Growing SOM (Bauer and Villmann, 1995). Nodes are added either to continue an existing dimension or by starting a new dimension. They are always added near to the centre of the map, creating larger hyper-cubical maps. For larger networks, derived from complex problems, the representation in output space can become hugely dimensional which is not desirable. Unlike GG, the GSOM uses time varying parameters that regress during training. Also, the dimensionality of GG is decided upon before training whereas the GSOM algorithm grows dimensionality during training. However, the GSOM cannot reduce its structure in any manner.

A method similar to GCS is dynamic cell structures (Bruske and Sommer, 1995). Developed independently from Fritzke's work, DCS combines the principle of insertion based on accumulated error with Hebbian learning. Bruske and Sommer have applied it to supervised learning and it is quite similar to GNG. A numerical comparison of DCS and GNG has not as yet been performed by the creators.

Another approach is the Growing Hierarchical Self Organising Map, GHSOM (Dittenbach et al, 2000) that evolves a hierarchical structure according to the requirements of the input space. The GHSOM consists of a number of layers. In each layer is a number of SOMs. For every unit in the SOM, a new SOM in the next layer may be added. An accumulative error decides where the new SOM is added to, much like the GG method. New units are added between the focus (winning unit) and the most dissimilar neighbour. Although the authors of the GHSOM claim that the network size is much restricted for large inputs sets compared to the GG method, the addition of many layers in a hierarchy does not reduce the overall size of the network. This method is particularly useful where hierarchical clustering of the input data is required.

1.6.2 Pruning Networks

These algorithms begin from the opposite premise to growing Neural Networks. In this case, networks are initially arbitrarily large and then neurons are removed to make the network either more computationally efficient or to increase the network's ability to generalise. Most of the work in this area is applied to feedforward networks (Thimm and Fiesler, 1997).

The main features of any pruning algorithm involve three decisions:

- *Which units to prune?*
- *When to prune?*
- *When to stop training?*

The choice of method to select the units to be removed is the most important as the other two questions are often answered within the algorithm. When designing an algorithm, a goal for the removal of connections needs to be chosen. This goal will control whether or not a connection is removed. Either the error induced from the removal of the unit is minimised or an estimate of the sensitivity of the Neural Network to the removal of a certain unit is calculated. Presently, the main methods for pruning are:

1. The simplest heuristic is to select and remove the smallest weights. The values of the weights are then added to the bias of the neuron.
2. Another possibility is to remove the connections with the smallest contribution variance (Sietsma and Dow 1988). The mean output of the removed connection is added to the corresponding bias.
3. The sensitivity of a connection can be calculated using the current weight value and the next change during training (Karnin, 1990). This sensitivity can then be used to remove neurons from the network.
4. A technique called Skeletonization estimates the error induced by the removal of a unit by multiplying its output with an additional strength (Mozer

and Smolensky, 1989). If setting the additional strength setting to 0 is equal to removing the links, then the units with a small derivative can be removed.

5. A statistical method (Finnoff et al, 1993) defines a statistic based on the probability that a weight becomes zero. If this probability is high, then the weight it removed. This sensitivity measure is incorporated into a pruning method called autoprune (Prechelt, 1995).
6. Optimal Brain Damage (OBD, Le Cun et al, 1990) works on the second derivatives of the error function to perform learning. The basic premise is to take a reasonably well working network and remove half or more of the weights to finish with a network that works just as well or better. This method essentially deletes the connections that have the least increase in error. A large Hessian matrix is used for this task which often makes the algorithm computationally expensive.
7. Optimal Brain Surgeon (OBS, Hassibi and Stork, 1993) attempts to solve the computational expense in Optimal Brain Damage by removing any assumption about the Hessian matrix. OBS has been shown to be better than OBD and other simple methods (nos. 1,2,3) on some problems (Kavzoglu and Vieira, 1998) but the criterion for success does not include computational issues and is thus incomplete.
8. Weight decay has proven to be not effective by Hanson et al (1989). Their experiments showed that using penalty terms meant that the weight decay algorithm would not converge where a normal back-propagation algorithm would.

Although pruning is a respected way to remove neurons, it is wasteful compared to the constructive techniques. Although no studies are available comparing the two paradigms, intuitively one can suggest that to construct a network up to the optimum level, is computationally more efficient than training an arbitrarily large network and then using an algorithm to reduce it to a more optimum size. If efficiency without the loss of error minimisation and generalisation is the goal, then a method that starts small is likely to be more efficient. Unfortunately, these sorts of comparisons are in the realm of further work.

All the methods for pruning networks given above operate on high-dimensional multi-layered Feedforward networks. Very little literature exists for self-organising maps, perhaps due to the complexity of removing neurons from a regular grid or perhaps because the problem of over-fitting depends more on the stopping criterion rather than the size and shape of the network.

1.7 Hybrid Neural Networks

All the algorithms investigated thus far have belonged to the realm of Neural Networks. However, there may be acceptable solutions to be found by creating hybrids with algorithms in other computational fields. Integrating two different fields together, is normally done to solve a problem or improve the operation of the network. This section investigates those techniques that aim to alter the Neural Network such that they overcome some of the problems detailed previously in the chapter.

The first field under scrutiny is Evolutionary Algorithms (EA). This field includes a number of algorithms and techniques based on the biological theories of genetics. The most popular algorithm is the genetic algorithm (GA) which operates a fitness function on a population of possible solutions (Holland, 1975). Those solutions with the highest fitness are used as a blueprint for the next population. The field has a staggering weight of literature, only the work that appears to solve the problems presented here is considered.

The main thrust of combining GA with NN research is in either the creation of topologies or the augmentation of learning rules (Branke, 1995). GAs are well suited for these tasks as they can search complex, large, multi-modal, multivariate spaces in parallel. However, there has been no known research into using a GA for finding learning rules that can operate through time. The reasons for this lie in the complexity of the fitness function that must be employed to quantify the usefulness of each possible solution. Also, these techniques are often extremely slow to compute as each generation of possible solutions requires a population number in the 100s. To train, validate and then test 100 possible neural network solutions is either a long sequential process or a complex parallel one. Also, a GA requires many

generations for the population to converge to a desirable solution, this value increasing with the number of parameters to be tuned.

Genetic algorithms also suffer when the network is to be represented by the algorithm. The *genotype* or genetic encoding of the network must be decided in the design. This code includes any information that will need to be optimised, such as parameters, topology or weight values. There are many different ways to set the genotype, some methods may be more conducive to network learning than others. Due to the parallel, abstracted process, it can be very difficult to be certain before the test that an encoding will optimise correctly (Hutt 2003).

The augmentation of learning rules (especially in feed-forward networks) is used to avoid the network being caught in poor local minima (a known problem for gradient descent based algorithms). Evolutionary algorithms can avoid local minima by searching several regions simultaneously; furthermore they do not place any restraints on network topology as they are not constrained by backpropagation learning rules. To this end, evolutionary algorithms appear appropriate for solving problems where gradient information is not easy to obtain (such as in recurrent networks). The genotype of such a system would include a number of weights of the network for a fixed topology. The values of the weights would be optimised by the genetic algorithm to produce an optimal set of weights.

One drawback of genetic algorithms is their inability to fine tune parameters (Kitano, 1990). Kitano's work suggests that there is no benefit of GA algorithms over algorithms such as Quickprop (Fahlman, 1988). An algorithm that can switch between evolutionary algorithms and Quickprop has been suggested (Heistermann, 1994) but this work has not been pursued.

Improved results have been realised using GAs to optimise the topology of the network (Dodd, 1990 is one example). This is because a Neural Network's structure greatly influences its performance. If the topology is too small, then the network may not be able to represent the desired mapping and if the structure is too large, then the network may not be able to generalise. The representation of the topology for this problem is not so straightforward. It must be decided whether the genotype should be

allowed to represent networks that are flawed or meaningless and whether the representation should be allowed to scale up to larger networks. There is a trade off here between expressive power and the exclusion of meaningless representations. Two solutions to this problem exist, either a low level encoding that specifies each connection individually (Miller et al, 1989) or a high level encoding that represents densities of neurons (Harp et al, 1989).

Some evolutionary algorithms attempt to simultaneously determine the weights and the structure of the network (Branke 1995). This takes a mix of the two techniques given above with long encoded genotypes. Unfortunately, it is often the case that the search space is huge and the number of generations required before any useful network structures are produced is also large. Also, the networks produced are specific to the problem given.

Although evolutionary algorithms evolve networks that are efficient for the purpose (through either topographical or weight optimisation), once the evolution has stopped and a solution has been found that is fit for purpose the network is static.

All the above genetic algorithms operate in a fitness space that is stationary. However, there is a branch of work that operates in spaces that are non-stationary. A standard GA has trouble operating on a non-stationary fitness landscape as after much iteration the population tends to cluster on a single peak, thus losing its diversity in search. As the landscape changes, perhaps a lesser peak becomes more prominent. The typical GA would have trouble switching to the other through recombination and mutation.

Simple, application specific solutions, making use of a bi-stable system were suggested by Goldberg and Smith (1987). A more general extension to the GA (Cobb, 1990) measured the average fitness of the population members with the highest fitness, altering the mutation rate depending on this value. As the average level of fitness decreased, the amount of mutation increased. This *hypermutation* allowed the GA to adjust to both tracking problems and to Markovian state transitions by retaining an element of diversity. A Markovian state transition is a probabilistic change from one state to another. This process falls down if the

population cluster does not decline locally, but instead reduces in fitness compared to the whole fitness space. Further work by Grefensette (1992) and Cobb and Grefenstette (1993) solves this problem by persistently perturbing 30% of the population. Although this leads to a slower convergence, it does allow the diversity to be maintained.

More recently the multinational GA (Ursem, 2000) uses a *niching* technique for tracking optima and sub-optima. Diversity is maintained by stopping all population members reaching the same peak. In this case, the population is encouraged to split into groups, each holding different sub-optima. This works very well for non-stationary systems as the sub-optima are the areas of the fitness landscape most likely to become future optima. This method is an improvement of its predecessors as all of the population is involved in the tracking of the non-stationary parameter, rather than using a percentage over the population to track the change. Also, unlike Cobb's initial work, there is no trigger to begin the tracking; the niching technique works constantly as part of the algorithm.

The work in genetic algorithms on non-stationary fitness landscapes is of particular interest because if the learning rule for a Neural Network was a multinational GA, then the learning process could change as the statistical makeup of the input space changed. This would allow learning in non-stationary environments. However, like other GAs used for optimising Neural Networks, this could be extremely inefficient and very complicated.

1.8 Novel Definition of Dynamic Neural Networks

A term is required that differentiates the networks illustrated above with those that solve the problems as outlined in the motivation. Presently the term *Dynamic Neural Network* could represent any of the following:

1. A Neural Network that is dynamic through learning (such as a Hopfield Network).
2. A Neural Network that constructs its topology through training (such as GNG and CC).
3. An algorithm that prunes it structure after training (such as OBD).

4. A Neural Network that operates on temporal sequences (such as TKM).

For this work, a new definition of a Dynamic Neural Network is required:

> *A dynamic Neural Network (DNN) is a Neural Network that can alter its own topology to accept novelty within a non-stationary signal space.*

In this definition, novelty is defined as a pattern belonging to a class that has not been shown to the network before. A stationary system is one whose class means (first order statistics) and variances (second order statistics) do not change over time. A non-stationary system is one where either the means or the variances (or both) change over time. The specific form of non-stationary systems considered here are those where classes in the input space do not change (there is no drift of classes) but new classes may appear and existing classes may disappear. This is the novelty in the non-stationary system. Drift can be taken into account in the adaptation of existing neurons to the input patterns but that form of non-stationary behaviour is not the aim of this work. In the real world problem given, radar data pulses are shown to the network. Although a radar source may be moving, and thus there would be drift in the 'bearing' parameter, the time constant of the drift means that many millions of pulses are seen before the emitter perceivably changes location. More interestingly is the appearance of a new radar pulse emitter and the disappearance of an existing one and the detection of this novelty in a timely manner. The traditional method of detecting novelty is using a rule based system, including many thousands of rules that are administered manually by a human expert.

Existing Neural Networks that could claim to be dynamic do not currently fit into this framework for one of the following reasons:

1. Topology is fixed after training.
2. Signals or patterns in the data set are stationary.

Although Cascade Correlation (CC) has the ability to accept new patterns after the initial training session (by simply adding new feature detectors), classes that no longer belong to the data set are still held by the network. Growing Neural Gas

(GNG) does have the ability to follow slow moving non-stationary distributions but rapid moving distributions often lead to neurons representing old distributions being held by the network.

As far as the author is aware, the sole piece of existing work regarding this area of research is a not widely reported piece of work, Growing Neural Gas with Utility (GNG-U) algorithm (Fritzke, 1997). This extension to the earlier work (Fritzke, 1995b) measures both the cumulative error and the utility of a neuron and then uses this information to decide whether a neuron should be removed. GNG tracks slow moving distributions well, however link removal is only performed locally around the focus neuron. As GNG-U is the only algorithm that attempts to solve the problems detailed above and bears some resemblance to the work here, it will be considered in some detail. However, it is not known to have been expanded upon since 1997.

The GNG-U structure is similar to GNG. It consists of a directed graph of nodes and arcs. Each node holds a reference vector that represents a position within the input space. Also, each node contains an accumulated error and a utility number. The algorithm is mostly the same as the GNG algorithm (as stated above) with the following modifications.

1.8.1 Utility Measure

The utility measure allows the network to remove those units which are no longer required by the network, due to a change in the input distribution. Utility is a measure of how much the error would increase for a given input signal, u, if the focus, z, did not exist. In this case, the input would be mapped to the second winning unit, z_2. Therefore, the increase in error (the utility) is simply the difference between $\|u-z\|$ and $\|u-z_2\|$. A local node variable, U_x, is introduced to keep the sum of the utility for a unit x. For each input signal, the utility variable is updated according to:

$$U_x = U_x + \|u - z_2\|^2 - \|u - z\|^2 \tag{1.8}$$

As with the local error values, the utility variables are subject to exponential decay after each adaptation step.

$$U_x = U_x - \beta U_x \tag{1.9}$$

The utility becomes small when the unit is very rarely the winner. This can be used to remove the unit when the utility falls below a fraction of the local accumulated error, E_x. Let i be the unit with minimum utility

$$i = \arg\min_c U_c \tag{1.10}$$

Then, unit i should be removed if

$$\frac{E_x}{U_i} > k \tag{1.11}$$

for some value of k. Large values of k will require a large ratio and therefore there will be fewer nodes removed. A small value of k will cause more nodes to be deleted. Fritzke notes (1997) that this paradigm opens up a whole area of applications (explored in Chapter 7, Further Work).

1.9 Introduction Summary

Classic Neural Networks (such as MLP, RBF and SOM) are static in their operation. Once they are trained, the input set is fixed and to accept further classes into the knowledge base of the network requires retraining with all of the classes. It is not always possible to have good examples of all of the classes before training and thus they will not be learnt properly.

The paradigm is to train the network and then validate it. The topology, learning parameters and network functions (such as activation functions) are set prior to the training process. The problem with this approach is that the choice of topology can affect the network operation. The most widely used technique is to iteratively try different topologies until the desired learning objective is achieved. One solution to this problem is to grow the Neural Network by adding neurons until the desired task

has been solved and good results have been obtained. Pruning techniques allow a network that is too large to shrink, but the possible loss of valuable information makes this technique unfavourable. These solutions solve the problem of fixing the topology *a priori*, but the resultant networks are still static as the input data set is static.

A new class of Neural Network termed Dynamic Neural Networks has been identified. Such a network does not have a fixed topology but neurons can be added or removed as required. This allows the network to change to meet the needs of the input data set. Unlike static networks, a dynamic network can learn the characteristics of a non-stationary data set. This ability can be exploited to detect novelty – that is to detect when an input belongs to a class that has not been seen before.

1.10 Thesis Organisation

- Chapter 2. Plastic Self Organising Map (PSOM). This chapter describes a solution to the problems given above and gives details of its extension, the Extended Plastic Self Organising Map (EPSOM).
- Chapter 3. Technique Evaluation: PSOM.
- Chapter 4. Temporal Plastic Self Organising Map (TPSOM). The TPSOM is an extension to the PSOM which solves an entirely different range of problems.
- Chapter 5. Technique Evaluation: TPSOM.
- Chapter 6. Discussion. The PSOM, EPSOM, TPSOM and GNG-U are compared.
- Chapter 7. Conclusions and further work.