Evolutionary Learning on Structured Data for Artificial Neural Networks

Filip Radliński

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

July 2002

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work.

Filip Radliński
4 July 2002

This thesis is dedicated to my family, and especially my mother.

Acknowledgements

In writing this thesis, as the completion of a year of hard work, I have many people to thank for their help along the way.

I wish to thank my supervisor, Professor John W. Lloyd, for helping to guide my project throughout the many twists and turns it has taken from random ideas into a concrete research project, and for helping me transform those ideas into reality. I also thank him for his patience in helping proof read the many drafts of this thesis.

I wish to thank Kee Siong Ng for having laid many steps along the way which have allowed me to focus on other aspects of this project, and for some insightful comments while proof reading this thesis.

I wish to thank my mother and older sister for their constant encouragement and support throughout this past year.

I would also like to thank John Uhlmann for many useful discussions and very careful proof-reading of my thesis, David Hellier for picking at my grammar, Richard Walker for pushing me to get more work done, Sarah Melissa Witiak for providing insights into evolution, and the Australian National University for providing me with a free lunch every day of the year through a generous scholarship.

Abstract

In this thesis, I present the GALSINE learning algorithm for automated learning on structured data. This is a genetic algorithm for artificial neural networks, modelled on evolution in nature, to allow effective machine learning while simultaneously limiting the amount of human intervention necessary.

The performance of a machine learner depends on an appropriate knowledge representation for the input presented to the learner, and a flexible design of the learning algorithm. The level to which both of these requirements are satisfied is largely determined by the background knowledge provided to the learner. In the approach discussed herein, it is shown that by including the background knowledge given to the learner explicitly in the knowledge representation, while automating the design of the learner, very good learning results are observed.

This thesis begins by presenting a background in machine learning, neural networks, and genetic algorithms, followed by an analysis of the GALSINE algorithm. I show the benefits of the algorithm, and demonstrate its effectiveness on a number of datasets. Finally, the learning process is studied, and the success of this project is discussed.

Contents

Part I

Introduction to Learning

Chapter 1

Introduction

This chapter briefly introduces machine learning, and some of the motivations behind this work.

1.1 Learning

Machine learning has a goal of developing software that can improve itself with experience, and generalise what it has learnt. Such software becomes truly useful to people not directly researching machine learning when the task in question is harder to perform than the task of creating and training the learning system.

There are two basic tasks a learner working from training data can perform: classification and regression. Whereas regression is essentially interpolating between existing data points, the task of interest in this thesis is classification. Classification is concerned with grouping examples into categories, be these examples of molecules being grouped according to whether or not they cause cancer [HKKS01], or websites being grouped according to type of content [Teo]. In the unsupervised learning approach, the learner is simply given these examples and must itself determine the way they could be grouped. In contrast, in supervised learning the examples together with labels indicating the correct category are provided to the learner. Therefore, the categories are fixed prior to any learning taking place. It is the process of determining which example is in which category that is learnt.

There are many approaches to achieve learning. These range from deterministic methods that process the examples in a predetermined way while tuning predefined parameters, to nondeterministic methods that randomly change the characteristics of the classifier. A particularly interesting approach is based on the evolutionary model seen in nature. Given some initial potential solutions to a problem, they are repeatedly perturbed, at each iteration selectively eliminating those that perform worst. It is on such an evolutionary approach that I will focus in this thesis.

1.2 Motivations

One may ask why use an evolutionary model at all. Classical learning methods are faster, and can be applied to almost any machine learning problem. The major benefit of evolutionary learning is automation. Instead of the learning model parameters being determined experimentally by a person for every task the learner is to perform, they can be determined by the machine. This results in the person just performing the knowledge representation – presenting the training examples in a machine readable form. The aim is for everything else to be taken care of automatically. This means the learner is given complete control over the model it builds.

Classical learning often utilises vectors of interesting properties of objects. These vectors are constructed by a person performing so-called "feature extraction". If done by hand, feature extraction requires knowledge as to which properties are useful, and extensive, time consuming experimentation. To learn from input which has not necessarily had feature vectors extracted, the learner must be able to process structured data, because most real world data is structured.

Typically, "learning", in machines as in humans, refers to the process that each individual goes through during its life. It is generally understood that learning does not allow for fundamental changes to the construction of the individual, although there is no clear delineation as to what constitutes such a change. Taking people as an example, we assume that we learn throughout our lives, but from one generation to the next we can only evolve (albeit in extremely minor ways over one generation). However, the connections in our brains do change as we learn, in effect constantly creating a new individual. Another interpretation of learning consists of changes which do not affect the code from which the individual was built, for people, DNA. For this thesis, I have taken an even broader definition of learning that includes all forms of adaptation, both within a single individual and within an entire population.

Throughout the many decisions in designing this algorithm, the most pervasive theme has been an attempt to observe nature. Evolutionary forces in nature have produced highly specialised life forms that have adapted to extremely varying environments, and have learned both to recognise and cope with endless external stimuli. Although the evolutionary process is slow, it is tremendously versatile. Many aspects in this design are based on nature and whenever a problem was encountered, the first question I asked was "how does nature solve this problem?". Moreover, in modelling machine learning on evolution, we can study the actual learning process, giving us greater insights into how learning occurs.

1.3 Project Aim

The aim of this project was to design an algorithm that can efficiently allow artificial neural networks to learn to classify structured data with minimal human intervention. It was also a goal of this project to observe, and study, the incremental improvements made by the learner, so as to begin to better understand the learning process.

1.4 Thesis Organisation

This thesis is in three parts. Part I provides a background in the relevant concepts from machine learning. The learning algorithm used is presented in Part II. Finally, in Part III, results from learning and observations of the learning process are presented and discussed.

In Part I, I start by providing a background in machine learning, artificial neural networks, genetic algorithms and approaches to structured data. Then I present an overview of work related to my own. Once all the necessary background is in place, Part II presents a quick overview of the GALSINE algorithm, followed by the details of the various components that make up the algorithm. The topics discussed are data structure analysis, individual representation, fitness evaluation and genetic operations. Part III starts with the presentation of learning results, showing how well the learner performs on various structured and unstructured data sets. Next, I present some observations of the learning mechanisms, and study the changes in the complexity of the learner with time. I finish by discussing how the parameters of the algorithm affect learning, and present a summary of the level of automation that this project has achieved.

Chapter 2

Machine Learning

Machine learning aims at re-creating learning processes in a computer. There are many ways learning can occur, and a number of ways the learning can be applied and evaluated. In this chapter, I present a brief background in the subject of machine learning, and in particular discuss a number of important points relating to background knowledge.

2.1 Supervised and Unsupervised Learning

There are two fundamental types of machine learning; supervised and unsupervised learning. To understand the distinction between them, we need to look at the source of the knowledge which allows the software to learn. When learning occurs in a program, it is through the program drawing on experience. This "experience" is provided in the form of *training examples*: instances of data that are provided as input to the learner. These examples are more often than not real world objects or measurements. They are usually described to the learner by a set of *features*, or calculated properties about each example.

In supervised learning, the learner has a "teacher" who gives a label to each training example – the solution the learner should find[1]. As an illustration, a supervised learning system might learn to recognise images of characters [ABE00]. The training data given in this case would be a set of images annotated by the correct character to which each image corresponds. An unsupervised learning system does not have this teacher. For example, you might have a web page clustering algorithm which is asked to learn to group websites without being told which websites belong to which clusters, or even how many clusters exist [Teo]. Reinforcement learning is a subclass of unsupervised learning that relies on rewards given to the learner by the environment to help it identify the method it should use to receive the highest reward. It is also often the case that the reward comes some time after the decisions have been made,

[1]This label is not necessarily correct, for example due to noise or an error in the data.

or with many fewer rewards than decisions by the learner. An example of a reinforcement learner is TD-Gammon, a backgammon playing program that learns to play the game by simply getting an appropriate reward at the end of each game [Tes95].

2.2 Data Classification

One of the main application areas of machine learning is data classification. Given an input, the learner's task is to assign it to one of a set of categories (or classes). As far as the learner is concerned, there need not be any meaning associated with the categories, but there is invariably something in common between the examples within each category. Data classification is a particularly appropriate application of machine learning because the range of the output is small (the output being simply the label of the category chosen), yet the problems still allow machines to learn complex concepts.

Data classification can be applied to a wide variety of domains, ranging from diagnosing cancer [Sel02], to parsing satellite data to map soil types in the Australian outback [KFS95]. Other areas where classification has been recently applied include classification of astronomical observations [PWKJ02], recognising carcinogenic molecules [HKKS01], or even trying to recognise abnormal behaviour such as the traffic patterns which may occur during attacks on computer systems [SFL$^+$00]. Research in data classification methods is thriving, with unlimited other areas to which it can be applied.

2.3 Hypotheses and Concepts

Machine learners work by trying to learn a function which maps the input to the correct output. The function that a learner is considering at a given point in time is referred to as a *hypothesis*. Machine learners can only learn hypotheses they can represent. In building a hypothesis, the learner constructs it from subcomponents, basic facts, or *concepts*. To illustrate this, consider a trivial learner fitting a curve to some data points. The curve fit with particular parameter values is a hypothesis. The concepts this learner knows about are real numbered parameters to the curve fit function. It cannot learn a function of a different form than the one given since it has no concepts describing different functions.

2.4 Background Knowledge

Machine learners always explore a hypothesis space defined by the algorithm and the training examples given to them. The hypothesis space can be defined as the subset of all hypotheses that can be represented by the learner. A number of machine learning techniques can explore an unbounded and complete hypothesis space. In particular artificial neural networks that are sufficiently large can approximate any given function to any required precision [Cyb88]. This means that, in principle, an artificial neural network can learn to perform perfectly any classification task over a finite number of examples. However, the hypothesis space searched must be limited, in the interest of finding solutions that can be generalised to unseen examples, and for the solution to be found quickly.

The hypothesis space searched by a machine learning algorithm is determined by a combination of the algorithm design and the data given to the learner. It is usually intentionally limited, and this occurs through the incorporation of explicit and implicit background knowledge. The implicit background knowledge, often overlooked, results from the features selected to represent the training examples. The explicit background knowledge consists of that applied by an person when features are added or removed, and that applied in deciding in what ways the learning algorithm should be restricted.

Implicit background knowledge is easier to understand if we consider an example. There is a classic tennis playing problem, proposed by Quinlan [Qui86] and discussed in depth by Mitchell [Mit97]. This problem involves determining if a given person will play tennis based on the weather. The observations are those of the weather, and belong to the class of all possible observations of weather, the *example space*. In the problem, the weather is described as:

$$Weather = Outlook \times Temperature \times Humidity \times Wind.$$

These four features are extracted from the examples and give a feature vector in *feature space*. The transformation, from example space to feature space, involves the application of background knowledge, in this case about what is important when describing the weather. But what if the deciding factor in the weather is not a property that was extracted into the feature vector, such as the temperature, or perhaps more unexpectedly, the dust content of the air? In this sense, the learner may be more successful if we could allow it to work directly from the examples.

The process of feature extraction is shown diagrammatically in Figure 2.1, where examples are mapped to feature vectors by the function F. This function applies implicit background knowledge. Noticing this implicit background knowledge shows the importance of selecting an appropriate knowledge representation.

The next step consists of applying explicit background knowledge to the feature vector. This is where new features can be created from existing ones, if an expert deems them particularly useful. Alternatively, existing features can be removed, if the expert considers them irrelevant. In creating new features, the expert is applying domain

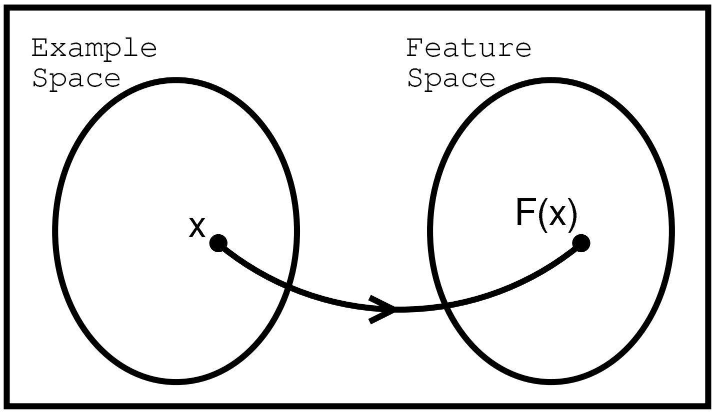

Figure 2.1: Data presented to traditional machine learners must be mapped from examples to feature vectors describing the examples. This mapping implicitly incorporates background knowledge.

knowledge from the problem area so as to simplify the learner's task.

Background knowledge is a double-edged sword. To help the learner most, it must be carefully considered. Put most plainly, background knowledge makes the assumption that the expert is right. In practice, the so-called expert often spends a substantial amount of time blindly exploring possible background knowledge that could be applied to the training examples. In doing so, he or she chooses by observing the results of the learning process. Needless to say, this task is very time consuming and error prone, and requires the input of a highly knowledgeable person to be effective. Additionally, it is very rare that the expert works from the examples directly. Usually, he or she assumes that the original feature vectors describe the examples well. In fact, more often than not, the original examples are no longer even available, so the expert is limited in what he or she can do by the pre-existing features.

There has been some research into automatically selecting features; for example, using genetic algorithms to pick a subset of a large number of possible features [HJ98]. Yet even the task of automatically selecting a subset becomes difficult because the number of possible subsets grows exponentially as the number of features gets larger.

Explicit background knowledge is also included when designing the learner. The hypothesis space searched by the learner is rarely manipulated directly. When dealing with many types of algorithms, the expert only decides on the structure of the learner, and this is especially true for artificial neural networks. As the expert makes modifications to the learning model, the real changes in the hypothesis space searched are often difficult to predict, and occasionally completely unknown to the expert[2] (such as when additional neurons are added to or removed from an artificial neural network).

[2] There has been some progress in this area, for example see [AB99], but this is still very difficult.

For the reasons presented above, the GALSINE algorithm places particular importance on the knowledge representation used for examples. Background knowledge is only provided through the selection of the knowledge representation, where it is most explicit. It needs to be provided somewhere, so that the task of the learner is not overly difficult. In this approach, the learner takes the knowledge representation and processes it directly. Moreover, it attempts to explore a complete hypothesis space, starting with simpler hypotheses, being left to itself as to how far it should explore. The natural exception is that some basic computational limits are placed upon it by the user.

2.5 Overfitting

For a learner to be successful, it must be able to reliably generalise what is has learnt from the training data, and not simply repeat answers it has been already given. When attempting to fit a model to some training data, one of the most common mistakes is overfitting. Overfitting occurs when the learner memorises the data rather than learning the concept that generated it.

As an illustration using an example regression problem, consider Figure 2.2. You can think of the curve fitting process as learning parameters in a general function. When working with real world data, there is always a small error in each data point. If we fit the real world data with a simple function, Equation 2.1, we get $a_1 = 3$ and $a_2 = 1$, resulting in the dotted line. If, however, we give the curve fit a more general function, such as described by Equation 2.2, we get the solid curve.

$$f_1(x) = a_1 + a_2 x \tag{2.1}$$

$$f_2(x) = a_1 + a_2 x + a_3 x^2 + a_4 x^3 + a_5 x^4 \tag{2.2}$$

f_2 passes through each of the data points exactly, whereas f_1 only comes close. There is no error in the fit f_2 but there is a non-zero error in the fit f_1. However, f_2 matches the data *plus errors*, not the underlying concept which resulted in this sample of data. As such, if we extrapolate the fit to cover examples not previously seen, a better fit may generalise worse. In this case, for larger values of x, the predicted result is worse in the case of the fit which better matches the training data. This effect is called overfitting. Avoiding overfitting is an important reason the hypothesis space of learners needs to be limited, although determining exactly how much it should be limited is often difficult to determine, except by trial and error.

However, in limiting overfitting, there is a risk of placing too many constraints on the hypothesis space. If the search is too limited, the target hypothesis that we would like

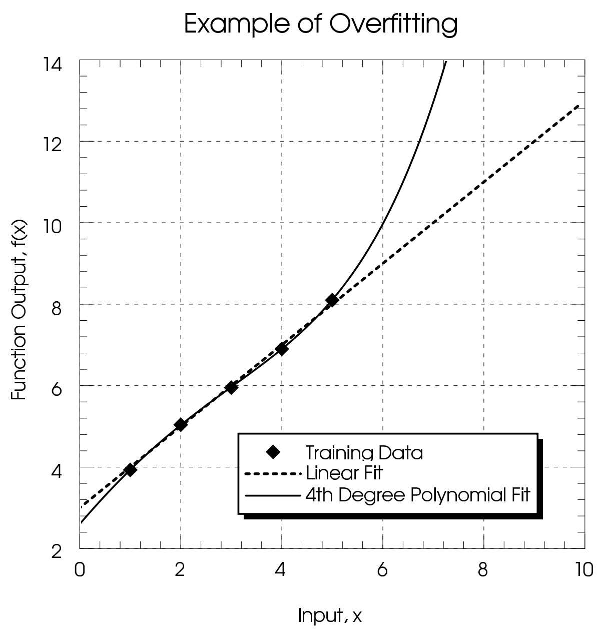

Figure 2.2: Real world data with a small error are fitted with a first and 4^{th} degree polynomial. Although the 4^{th} degree polynomial fits the data slightly better, we can intuitively see that extrapolating to larger x values produces a much larger error.

the machine to learn may simply not be present. For example, if we tried to perform the above curve fit with the function $f_3(x) = a_1$, then the fit would be even worse than f_2. In this sense, the hypothesis space must be chosen, or searched, in such a way so as to limit the complexity of the solutions but not overly so. Overfitting as applied to artificial neural networks is discussed further in the next chapter.

For a more complete overview of machine learning, I refer the reader to Mitchell's outstanding book [Mit97].

Chapter 3

Artificial Neural Networks

Artificial neural networks (ANNs) are built from remarkably simple components which, when taken together, can exhibit remarkably complex behaviour. In this chapter, I present a background to this form of machine learning.

3.1 Neurons

The basic component that ANNs are built from is the neuron[1]. Mathematically, each neuron is very simple, performing a weighted sum and applying a "squashing" function to the result. Figure 3.1 shows a neuron diagrammatically. Generally, neurons have n inputs with values i_j, each weighted with a real number w_j. The first stage of the neuron calculates a sum of the weighted inputs:

$$s = \sum_{j=0}^{n} i_j w_j = \vec{i} \cdot \vec{w}. \tag{3.1}$$

Input 0 is fixed to the value 1, and is referred to as the bias. Through an appropriate weight w_0, it allows s to be zero for certain non-zero inputs i_1 through i_n.

The second stage of the neuron applies a function to ensure the final output is in a small range, usually $[-1, 1]$ or $[0, 1]$. This function is often referred to as a "squashing function," and gives the neuron the non-linear properties which allow sufficiently large ANNs to approximate any given function to any finite precision [Cyb88]. The functions most commonly used as the squashing function are the Heaviside function[2], H, defined in Equation 3.2, the sign function, sgn, defined in Equation 3.3, and the sigmoid function, σ, defined in Equation 3.4. Using a sigmoid squashing function, this gives the final output, o, of a sigmoid neuron as $o = \sigma(s) = \sigma(\vec{i} \cdot \vec{w})$. A number of

[1]Sometimes called a perceptron or a node, although perceptrons are slightly different.

[2]The difference between perceptrons and neurons is that perceptrons always have the Heaviside function as the squashing function.

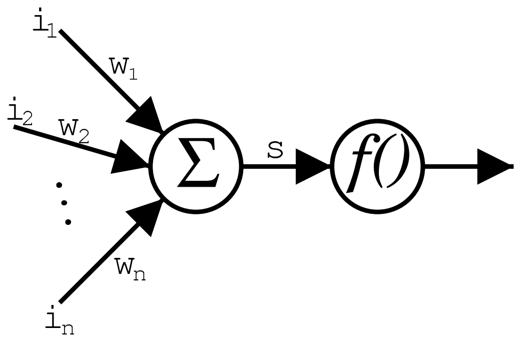

Figure 3.1: Functional view of a neuron.

other squashing functions are explored by Duch and Jankowski in [DJ01].

$$H(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.2}$$

$$sgn(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.3}$$

$$\sigma(x) = \frac{1}{1+e^{-\alpha x}} \qquad \alpha > 0 \tag{3.4}$$

A typical ANN is constructed by combining a number of neurons into multiple layers, creating a Multiple Layers of Perceptrons (MLP) network[3]. Only the first layer takes input from the outside world, while the neurons in the last layer return the "answer". Usually, the output of each neuron in each layer is connected to the input of each neuron in the next layer. We refer to such a network as fully connected, an example of which is shown in Figure 3.2. There is usually an implicit neuron in MLP networks that always outputs 1 and it connected to all other neurons. Any intermediate layers between the input layer and the output layer are called hidden layers, as they are not directly connected to the outside world. The layers do not usually have the same number of neurons. Instead, the number of neurons in each layer is determined experimentally through trial and error, by finding the setup that gives the best performance. This addition and removal of neurons and layers is most often the method by which the hypothesis space searched by ANNs is adjusted.

Up to this point, I have generally discussed ANNs as modelling a function. This implies a single output neuron per value returned from the ANN. Classification is handled slightly differently – with one output for each possible class. We can define the *target output* as the particular values we would like the neurons in the output layer to return when an example is presented to an ANN. In classification, the target output

[3] This name is misleading, as what is actually constructed is a network of multiple layers of neurons.

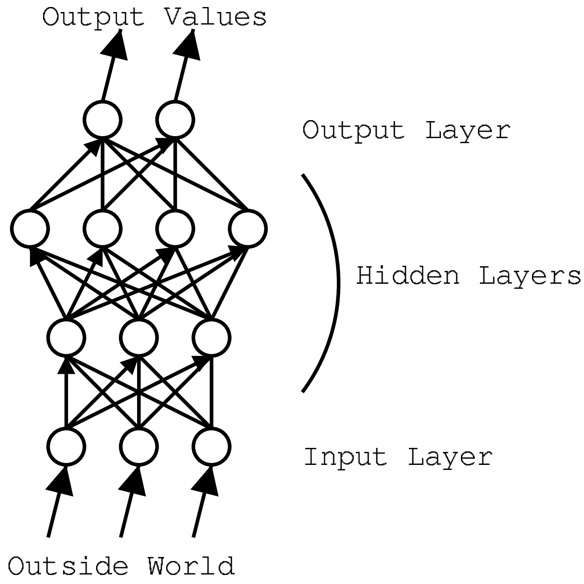

Figure 3.2: A fully connected ANN with two hidden layers.

is for the neuron associated with the correct class to be 1 with all other values 0.

Neurons and perceptrons have been a topic of research since the 1950s. Early research is very nicely described in [MP69]. At that time, MLP networks were simply a novelty. Despite much analysis of perceptrons and of networks made up of one layer of perceptrons, Minsky and Papert wrote:

> The perceptron has shown itself worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: [. . .]. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps [. . .] some profound reason for the failure to produce an interesting "learning theorem" for the multi-layered machine will be found.

This problem of finding a learning algorithm for MLP networks was solved by Werbos [Wer74], Parker [Par85] and completed by Rumelhart [RHW86]. The method discovered, back propagation, allowed research in MLPs to start at a rapid pace, and brought about a renewed interest in ANNs. Back propagation will be discussed further below.

A simple extension upon MLP networks is to allow recurrent networks, that is, with connections allowed from any neuron to any other, in either direction. In this way neurons in later layers can have their output connected to the input of neurons in earlier layers. When evaluating the output of neurons that have inputs from later layers, the output of those later neurons is not yet known. This means the "backwards" links must be interpreted slightly differently – the earlier neurons take the output of the later neurons from evaluating the *previous* example as their input. In this way, recurrent MLPs essentially remember previous results and so can be used where the output must depend on earlier examples and not just the current one.

3.2 Learning

When creating an ANN, the links between neurons must first be created, and then the weight of each link must be determined. I will refer to the first of these steps as *construction*. The second step is usually called learning, although I will call it *training*. Construction is generally regarded as the harder task and is more often than not performed by a person. This encompasses determining the number of neurons, layers and links, as well as deciding which neurons should be linked to which other neurons. These definitions allow me to define *learning* in the broader sense that I referred to in the introduction as the sum of construction and training.

When the ANNs studied become larger and can approximate more complex functions more closely, the number of possible links grows extremely rapidly. As a result, the fraction of possible links that are considered must become smaller to avoid the training problem becoming computationally intractable. Whereas construction can be neatly partitioned off from training if we are willing to restrict ourselves to fully connected layered ANNs, this is not the case if we want to allow the linking to be determined as part of the learning process. Selecting the best level of linking, and which particular links should be present, becomes an integral part of learning.

3.3 Training

When an ANN is constructed, it typically has each weight initialised to a small random value, usually in $[-1,\ 1]$. A training algorithm is then used to train these weights. The most common training algorithm for ANNs is back propagation, or some modification on it. There is an excellent review of back propagation in [Mit97], to which I refer any reader interested in a thorough description, although in the interest of completeness I will now present a brief summary.

We can start by considering a single neuron, where we have an input vector $\vec{i}$, with target output t. We can rephrase the learning problem as minimising the error in the output o by modifying the weight vector $\vec{w}$. We can define the error $\mathcal{E}$ as:

$$\mathcal{E}(\vec{w}) = \frac{1}{2}(t-o)^2. \tag{3.5}$$

To determine the optimal change in $\vec{w}$ at each training step, we perform gradient descent down $\mathcal{E}$. Gradient descent is process of looking for the minimum on surface by always heading in the direction where the slope is steepest. This is done by computing $\nabla\mathcal{E}(\vec{w})$, the gradient of $\mathcal{E}$ with respect to each weight w_j in $\vec{w}$.

$$\vec{w} \leftarrow \vec{w} - \eta\nabla\mathcal{E}(\vec{w}) \tag{3.6}$$

In other words, each component weight w_j of $\vec{w}$ is updated by the following rule. Note that this calculation requires that both $\mathcal{E}$ and the squashing function are differentiable.

$$w_j \leftarrow w_j - \eta\frac{\partial\mathcal{E}}{\partial w_j} \tag{3.7}$$

In Equations 3.6 and 3.7, η is a positive constant known as the learning rate. It determines the step size taken as we perform the gradient descent. It must be tuned for optimal performance: if η is too small, the weights will change extremely slowly and so learning will be slow. If η is too large, we can overshoot minima in the error surface, resulting in oscillating behaviour, and not produce good learning results.

Typically, gradient descent in training an ANN is repeated for each example in a set of training examples. The process is then repeated through a large number of iterations over the set of all examples.

Back propagation is an extension of gradient descent to MLPs, such as the one shown in Figure 3.2. As we are dealing with multiple outputs, we first need to rewrite the error function appropriately:

$$\mathcal{E}(\vec{w}) = \frac{1}{2}\sum_{k\in O}(t_k - o_k)^2, \tag{3.8}$$

where O is the set of all neurons in the output layer. In a similar way to Equation 3.7, when working with a layered ANN with many neurons, we can update the i^{th} weight of the j^{th} neuron using the rule:

$$w_{ij} \leftarrow w_{ij} - \eta\frac{\partial\mathcal{E}}{\partial w_{ij}}. \tag{3.9}$$

Differentiating $\mathcal{E}$ with respect to the weights is relatively straight forward in the case of the squared error function given in Equation 3.8, although I refer the reader to Chapter 4 of Mitchell [Mit97] for a full working.

One of the main problems with gradient descent algorithms is that they can get trapped in local minima. This is best illustrated with a diagram showing one dimensional gra-

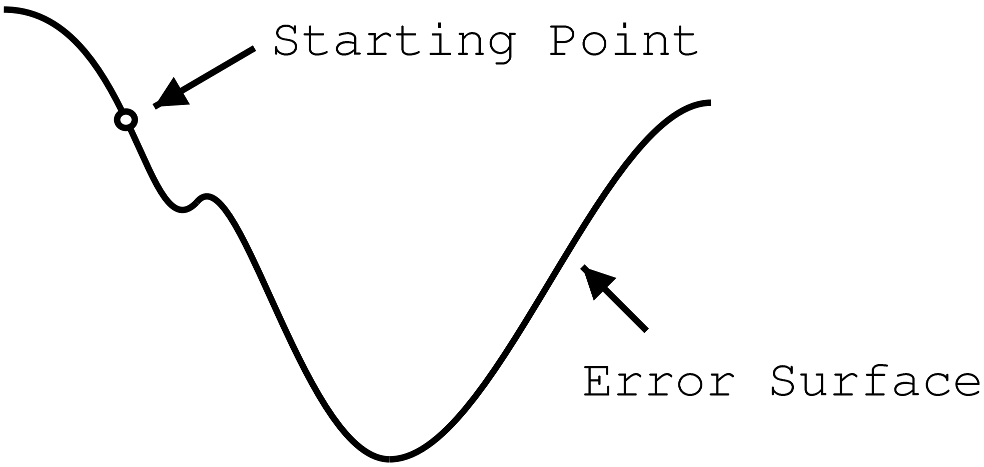

Figure 3.3: An example where a simple gradient descent algorithm would not find the minimum correctly

dient descent. In the case of the error surface shown in Figure 3.3, we can see that performing simple gradient descent can be more or less successful in finding the minimum depending on the starting point. ANNs in particular can have complex error surfaces, which makes finding the global minimum very difficult. In their simplest form, gradient descent algorithms continue in each direction so long as the value of the function being minimised decreases. However, learning is stochastic, and there must be a finite step size. In other words, when we calculate the slope, we only know it at a particular point, but we must update each weight by a finite amount. The magnitude of this update is proportional to the step size η. At each iteration, the current best solution moves by some distance (determined by η) down the slope through an adjustment to each weight. This means that if there is a very small local minimum relative to the step size, we are unlikely to fall into it, which is good. On the other hand if the step size is too large and global minimum is in a very narrow valley, the correct solution may be overshot. In any event, for a given step size, there is also a threshold barrier size past which a gradient descent algorithm cannot pass. For example, assuming the step size is small (say 1mm), gradient descent in Figure 3.3, with the starting point shown, would stop in the small valley. Various methods to minimise the effect of local minima have been proposed in the literature, and they are discussed in the next section.

3.4 Variations on Back Propagation

Many extensions to, and variations upon, back propagation have been developed over the years. I present those that I consider particularly interesting and most relevant below.

The first extension is the application of a solution from nature to simple gradient descent. When a ball is rolling down a surface, it is performing gradient descent in minimising its potential energy. In the example presented in Figure 3.3, we would expect the ball to escape from the small valley and find the global minimum (the large valley) due to its inherent momentum. Similarly, we can add momentum to gradient descent. First we rewrite Equation 3.7 as follows:

$$w_j(t) \leftarrow w_j(t) + \Delta w_j(t), \tag{3.10}$$

with the t indicating this update is taking place at iteration t. $\Delta w_j(t)$ given by:

$$\Delta w_j(t) = -\eta \left(\frac{\partial \mathcal{E}}{\partial w_j} \right)_t \tag{3.11}$$

Adding a momentum term to the update rule, we can rewrite Equation 3.11 as:

$$\Delta w_j(t) = -\eta \left(\frac{\partial \mathcal{E}}{\partial w_j} \right)_t - \alpha \Delta w_j(t-1), \tag{3.12}$$

where α determines how pronounced the momentum is. α typically has a value of about 0.1, and always satisfies $0 \leq \alpha < 1$. More informally, the update at iteration t has a fraction of the update at iteration $t-1$ added to it. This means that if the weights are updated by a large amount in a particular direction at one iteration, they are likely to change in a similar direction at the next iteration. For example, if the weights change in a particular direction at iteration t, but at the next evaluation the slope in that direction is slightly positive, the weights may still move in that direction.

In the rolling ball analogy, after the ball has found the local minimum, we would expect to see the ball continue up the hill and oscillate for some time before coming to rest. In fact, it is conceivable that if the level of friction on the ball is just right, the ball may rise up the other size of the global minimum and come back to get stuck in the local minimum after all. This balance between a high momentum to allow escape from local minima, and a momentum which is not so high so as to overshoot lower minima, makes it difficult to pick an optimal value for α. That said, a small momentum typically increases the performance of back propagation significantly, as the error functions of ANNs tend to have many small local minima.

Another feature visible in Figure 3.3 is that the result of the gradient descent can depend greatly on the starting point. Historically, ANN training has relied on many repeated learning attempts, each with a different random starting point, to find an optimal solutions. A significant improvement to this "repeat everything" method was presented by Fahlman [Fah88] with a further improvement, the Early Restart Algorithm, proposed by Magdon-Ismail and Atiya in 2000 [MIA00]. In the Early Restart

Algorithm, we follow the convergence of the error in the output of the ANN, and restart the training at a new set of random weights if the convergence is too slow. Restarting gives an additional improvement because it avoids waiting for slow descent down small gradients. Magdon-Ismail and Atiya showed that their method allows significantly faster learning than the "repeat everything" approach on small ANNs learning simple functions.

Finally, the error function given in Equation 3.8, the squared error function, was presented without any justification. This is the standard choice for back propagation, but any differentiable function is acceptable. Unnikrishnan, Hopfield and Tank [UHT91] propose a different error function, the information theoretic (log) error measure, given in Equation 3.13. In this error function, the target output t_k, must be either 0 or 1.

$$\mathcal{E}(\vec{w}) = \sum_{k \in O} t_k \, log(\frac{t_k}{o_k}) + (1 - t_k) \, log(\frac{1 - t_k}{1 - o_k}) \tag{3.13}$$

The benefit of such an error function is that it is smoother and contains fewer local minima than the squared error function. Unnikrishnan et al. demonstrate that for a network with no hidden layers and a sigmoid squashing function, the information theoretic error has only a single minimum. This error function also showed superior performance when applied to the Alopex learning algorithm (described below). I will explore its utility in Chapter 14.

Back propagation nevertheless has the limitation that it relies on the computation of the partial derivative of the error function with respect to each weight in the ANN. This becomes much less effective for larger networks, when these partial derivatives become very small and result in miniscule changes in weights at each iteration. Unnikrishnan and Venugopal present Alopex algorithm in [UV94]. This algorithm does not assume anything about the error function used, and can train any ANN, irrespective of the squashing function, error function or network linking. Alopex works by modifying the weights by a constant amount δ, either increasing the weight or decreasing the weight at each iteration. The direction of change at iteration t is determined by the difference in error between iterations $t - 1$ and t, and difference in the each weight at these same iterations. For example, if a weight was decreased at the last iteration and this reduced the error, it is likely that the weight will be decreased again. For details about this algorithm, I refer the reader to [UV94].

3.5 Overfitting

As discussed in the previous chapter, overfitting is a major concern in any form of machine learning. ANNs are particularly prone to overfitting because of the number of parameters present. ANNs are a large construction with many numerical parameters so they very often have more degrees of freedom than training examples. There are a number of general approaches to limit overfitting in ANNs. The simplest approach is to stop training at the appropriate time. To measure when is appropriate, training is performed using two distinct datasets – a training dataset and a validation dataset. The training dataset is usually larger, and is used to train the network in the usual manner. As we would expect, while the network is trained, the error on the training data decreases over time. However, at each training iteration, the ANN is also tested on the validation dataset. The assumption is made that so long as the error on the validation set is improving, the learning is generalisable. However, when the ANN begins to overfit the training data, what it has learnt begins to apply worse to data that the ANN has not been trained for. This results in the error on the validation set increasing. Despite the simplicity of this solution, it is essential to remember that the validation set is usually a small set of data, and might not accurately show how generalisable the learning of the ANN is. In particular, this is the case if the validation data is somehow correlated to the training data.

A second approach to limit overfitting involves essentially the opposite process. The Optical Brain Damage (OBD) approach is to take an ANN that is performing well on some training data and cut id down to make it more generalisable [LDS90]. As LeCun et al. succinctly put it:

> The basic idea of OBD is that it is possible to take a perfectly reasonable network, delete half (or more) of the weights and wind up with a network that works just as well, or better.

The OBD algorithm involves first training an ANN using back propagation, until it reaches some target performance on the training data. Then the link whose deletion would increase the error least is chosen and deleted (or equivalently, its weight is fixed at 0). The network is retrained, and the entire process is repeated numerous times. This limits overfitting by simplifying the network as much as possible, which is equivalent to reducing the number of parameters in the overfitting example in the previous chapter. LeCun et al. show this method to be very effective. A number of modifications on the OBD algorithm have been proposed in the literature (for example see [HS93]).

We can, for example, consider removing entire neurons from an ANN rather than just links. In the same way that links are selected for removal in the OBD approach, we can remove the neurons whose removal has the least impact on the performance of a given ANN. This approach is proposed and discussed by Mozer and Smolensky in [MS89].

3.6 Constructive Learning Methods

So far, I have only discussed ANN learning restricted to what I have defined as training, that is, given an ANN with predetermined links, the weights on these links are determined. Numerous constructive learning methods have been suggested to construct and train ANNs simultaneously in an automated fashion, so as to remove the limitation of ANNs being necessarily designed by a human. In this section I will only discuss non-genetic algorithms of this type, as the next chapter is devoted to genetic algorithms.

The most common constructive algorithm is cascade correlation [FL90]. Cascade correlation starts with a simple network and add new neurons one at a time. Each time a neuron is added, it is connected to the output of all neurons already present. The new neuron then has its weights trained by backpropagation. When this has been done, the weights are locked permanently and the process is repeated until the ANN is performing acceptably. The original purpose of cascade correlation was to provide a faster alternative to back propagation, although the more relevant idea that it demonstrates in this thesis is that all of learning can be automated.

Cascade correlation results in ANNs of a very rigid structure, and later neurons have a tremendous number of inputs when the network size becomes larger. We can essentially consider a network built by cascade correlation as having each neuron in its own hidden layer and connected to all the neurons in all previous layers. A modification to cascade correlation was proposed by Phatak and Koren [PK94], whereby the connectivity is more restricted. They place more than one ANN in each layer, and with the input of each new neuron connecting only to the output of the neurons in the previous layer. This can be thought of as an MLP network with new layers inserted between the last hidden layer and the output layer. A further extension to the idea of restricting the connections in cascade correlation was suggested by Tenorio and Lee [TL90]. Given a set of possible links to a new neuron, they propose a way to select the best ones to actually connect.

Another approach to dynamic ANN construction is adding width to the network rather than depth. For example, the Dynamic Node Creation algorithm [Ash89] starts with a small ANN with a single hidden layer and trains it using back propagation. Whenever the performance of the ANN does not improve fast enough, a new neuron is added to the hidden layer and the process is continued. An interesting twist on such a width increasing algorithm is the Meiosis approach [Han90], where neurons in the hidden layer are split in two when certain criteria are matched. This and other neuron splitting approaches are, however, computationally expensive, since selecting the neuron to split depends on each neuron's properties, which must be calculated and tracked.

There are a number of authors who have surveyed dynamic construction of ANNs, discussing these and other methods (for example [KY97], [Cam97] and [AG97]).

Chapter 4

Genetic Algorithms

Genetic algorithms are inspired by the evolution of life. Also called evolutionary algorithms, genetic algorithms for learning in ANNs have recently been explored by a large number of researchers (for example[Yao99], [MSG99] and [Koz98]). This chapter introduces genetic algorithms and points out the underlying principles. In so far as possible, I try to present genetic algorithms as a general method applicable to any form of machine learner. However, in some places, for reasons of clarity I must resort to describing them as applied directly to ANNs.

4.1 Basic Concepts

The basic process for all genetic algorithms is shown in Figure 4.1. There are three major components to a genetic algorithm – in the words of John Holland, in his ground breaking book [Hol75], these are:

> (1) the environment, E, of the system undergoing adaptation, (2) the adaptive plan, τ, whereby the system's structure is modified to effect improvements, (3) a measure, μ, of performance, i.e., the fitness of the structures for the environment.

In adapting to the environment, we must of course be dealing with individuals, or structures as Holland calls them. Each individual in the current generation is evaluated in E by the fitness function μ. The worse individuals are discarded while the best individuals are then modified using the adaptive plan τ and the process is repeated.

If we think about how we start an evolutionary process, the first step must be the creation of an initial population of individuals. Each individual is a potential solution to the learning problem, representing a point in the hypothesis space being searched. Usually, these individuals are created semi-randomly so as to sample the hypothesis space as uniformly as possible.

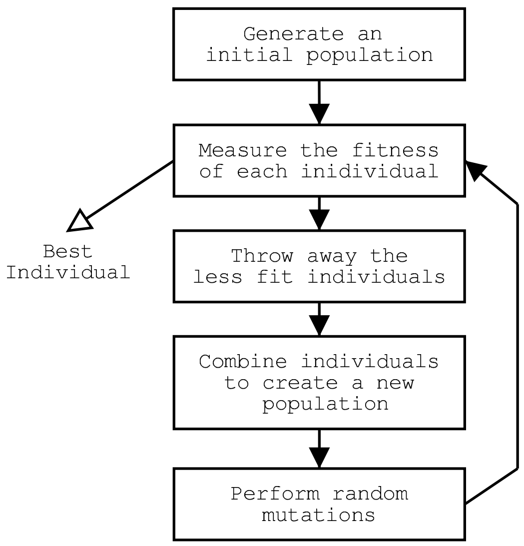

Figure 4.1: The basic structure of any genetic algorithm.

The fitness function, μ, tells us how good a particular individual is as a solution to the problem, with a higher fitness meaning a better individual. A common fitness function simply measures the error on the outputs of the ANN, for example setting $\mu = 1 - average(error)$. If the squashing function ensures the output is in the range $[0,\ 1]$, we know $\mu \in [0,\ 1]$.

Let us for a moment consider how an ANN classifies examples. As discussed in Section 3.1, we have one output for each possible category. Consider an ANN which classifies an input as belonging to one of two categories. If it classifies an example perfectly, it would output 1 on the correct output neuron and 0 on the other. However, if the ANN outputs 0.9 on the correct output and 0.1 on the other, it is still fairly clear that the classification is correct despite there being an error in the output (taking the squared error function, it is 0.02). Taking a set of say 100 examples, an ANN which always outputs 0.9 and 0.1 but gets the classification right every time would have an average error of 0.02 (meaning $\mu = 0.98$). However, every example is classified correctly, so we can say that the ANN has a *correctness*, $\mathcal{C}$, of 100% (i.e. $\mathcal{C} = 1$). On the other hand, an ANN which always outputs 1 and 0 on the correct outputs, except for one example where it outputs 0.5 and 0.5, would have a better fitness $\mu = 0.995$ despite a smaller correctness of $\mathcal{C} = 0.99$. In the end, we want to find the individual which has the best correctness. This shows us that an intuitive fitness function which only takes into account the error, such as that used in back propagation, can be misleading. The solution to this problem is for the fitness function to take both error and correctness into account. We need to include the error to have a fine grained fitness measure which can recognise small improvements in individuals, but in the end correctness is more important. I will discuss the intricacies behind fitness function design in more detail in Chapter 10.

After each generation has been evaluated, genetic algorithms allow the better individuals to survive, whilst removing the worst individuals. The surviving population is then recombined into a new complete population using the genetic operators of the adaptive plan τ. Usually, the best half of the population is allowed to survive. Next, individuals are either slightly modified (mutated) to create a new population of individuals, or combined (crossed over) pair-wise to create new individuals which then may or may not be mutated as well. The precise types of operations depend greatly on the particular algorithm, although they fall into these general categories. The defining characteristic is that mutation maps a single point in hypothesis space to another point in hypothesis space, whereas crossover combines multiple hypotheses into a new one.

Let us first consider the harder of these operations; crossover. Crossover operations aim to *non-destructively* combine useful features, or functional parts, of two or more parent individuals into a new individual. The non-destructiveness requirement of crossover has led to many genetic algorithms which simply omit the crossover operation. Often, it is replaced with a simple duplication of the surviving individuals. The difficulty in crossover is strongly related to the representation used for individuals. In order for a crossover operation to be successful and useful, we need a representation of individuals which helps in the preservation of useful features. Section 4.2 discusses the various representations that can be used.

The easier operation is mutation. Taking a new population, created either through crossover or copying, each individual in the population may be modified in a small way through a mutation. The mutations are often random and provide the other main mechanism for exploring the hypothesis space. Mutations perform small changes to the representation of the individuals, and so long as a small change in the representation usually results in a small change in the ANN, the mutation operations will be effective. It is worth noting that, despite mutation being simpler than crossover, we must also be careful that when mutations are performed they don't result in unwanted ANNs (such as recurrent networks, if recurrent networks are unwanted as if often the case). The algorithm may choose to not perform mutations that result in unwanted networks, but must do so explicitly.

There are a number of other genetic operations which have been proposed and discussed in the literature, although they have been ignored recently. For an analysis of a large number of classes of operators, I refer the reader to the remarkable text about genetic algorithms by Goldberg [Gol89].

Finally, once the genetic operations have been performed, the entire process is repeated. It is worth briefly mentioning that the final goal of genetic algorithms for ANNs is usually the production of one individual which performs well, and so after fitness evaluation the algorithm must identify the "best" individual. Selecting this best individual is by no means trivial, due to the risk of overfitting. Many algorithms also use this best individual to determine whether to stop or continue for another generation.

4.2 Individual Representation

To be able to perform the two basic genetic operations described above, mutations and crossover, we must consider how the individuals are represented by the algorithm. For example, a crossover operation that combines two ANNs directly makes little sense, because we cannot define mathematical functions of links or nodes. The operations that are available to the genetic algorithm rely on the internal representation used for individuals. This representation is usually called the *genotype*, and describes how the individual is put together. The actual individual, built by interpreting the genotype, is referred to as the *phenotype*. Both these terms are taken from biology. The process of converting a genotype to a phenotype is referred to as *decoding* throughout this thesis. There are two basic types of encoding schemes, that is, ways the genotype can described the genotype – direct and indirect. There are a number of studies of the many encoding schemes presented over the years (for example [Yao99] and [Bli98]).

4.2.1 Direct Encoding

A direct encoding of an individual contains all the information about the individual in a simple form. In the case of ANNs, this includes all the information about the links and weights of the individual. The process of decoding the genotype to a functional individual (phenotype) is deterministic, simple and predictable. It is easy to predict how a change to the genotype will modify the phenotype of an individual. If the properties of an individual are measured, they are guaranteed to be preserved if the same individual is recreated at a later stage.

However, a direct encoding scheme makes the genotypes very large for large networks, growing at least as fast as the number of links. If the size of the representation must be limited for practical reasons, the individual is restricted, and hence a smaller hypothesis space can be represented, although this is rarely a problem.

The simplest direct encoding is a connectivity matrix (for example [MTH89]), one row and column for each neuron, and a link between two neurons is represented by a non-zero weight in the appropriate position. This encoding has a major drawback that it does not provide for a simple non-destructive crossover operation. Moreover, the size of the individual encoding grows as $O(neurons^2)$. A much better encoding is presented in Chapter 9.

4.2.2 Indirect Encoding

An indirect encoding differs from a direct encoding in that the construction process is more complex, and often even non-deterministic. The genotype describes various key parameters used when decoding the individuals, and sometimes describes rules for construction of individuals. Examples of key parameters may be the number of neurons, the layout of the neurons and the range the weights are chosen in. For an example of such an approach I refer the reader to Harp et al. [HSG90].

Often, with indirect encoding schemes (such as in the case of [HSG90]), after an individual is decoded, it is trained using some non-genetic training algorithm before having its fitness evaluated. This means that if an individual has a high fitness, it does not guarantee that it will perform equally well in the future after it is decoded and trained again. In this sense, one of the hardest aspects of indirect encoding schemes is selecting the essential features which should be encoded in the genotype.

A particularly interesting approach to indirect encoding is the use of development rules [Kit90]. Development rules are a recursive approach where we start with a single element. This element is replaced by a 2x2 matrix of other elements. These elements are recursively replaced until we have just terminal characters in the matrix. For example, we can define the following rules:

$$S = \begin{pmatrix} A & B \\ C & A \end{pmatrix} \quad A = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Then we can perform recursive expansions giving:

$$(S) = \begin{pmatrix} A & B \\ C & A \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

In this way we have a simple set of rules which can be used to create an ANN and these rules can be represented and manipulated with genetic operations in a relatively straight forward manner.

4.3 Inspirations

Genetic algorithms come from modelling nature, which is also where the entire area of artificial neural networks originates from. In nature, genotypes are sets of genes, each encoded by a sequence of DNA. The genes then encode the instructions for the creation of proteins, which form the basis of life. It is true that a method being inspired by nature, or not, does not change how effective it is, but real evolution has been so successful that it is certainly worth studying it in more detail.

Despite a large amount of research in the area, automated network algorithms are still not useful or effective enough to compete with traditional methods. ANNs constructed and trained automatically are generally less effective learners than manually constructed ANNs. When ANNs are designed by hand, almost all of them have the same basic structure – a given number of layers, with each layer having a specific number of neurons, and all the neurons in each layer connected to all the neurons in the next layer. This uniform connectivity is very unlike anything seen in nature. Moreover, the global learning techniques applied to ANNs, where each neuron observes the performance of the ANN as a whole, is very unlike learning in the brain, which is the original neural network. These properties in traditional learning only come about from the need for a theoretical understanding of the construction and training methods, but I believe they also limit the success of ANNs.

In the remainder of this section, I essentially provide a summary of some basic facts from Rolls and Stringer [RS00], which provides a very interesting background into the construction of the brain. This is relevant to machine learning because I believe that, having taken the ANN model from nature, we can learn a lot more from nature as to how ANNs should be built and trained.

Neurons in the brain are not uniformly connected to other neurons. It is logical that links are made mainly to closer neurons and more rarely to neurons further away. Moreover, certain groups of neurons are highly interconnected within the group and only weakly connected to neurons outside of the group. This is somewhat like a well designed piece of modular software, with high intraconnectivity within a module and low interconnectivity between modules. As we design more and more complex ANNs, we would expect, and hope, to see such modules appear, and to have these modules perform various specific tasks.

The non-uniform connectivity of neurons in the brain is also of interest. In the brain, any given neuron only connects to a very small fraction of the other neurons by typical ANN standards. The human brain has between about 10^{10} to 10^{11} neurons, each with an average of about 10^4 links to and from other neurons, resulting in an average connectivity of about 1 in 5×10^7. This is very different from the 50% connectivity common in ANNs today, although we do see lower connectivity in ANNs as the number of neurons grows.

Moreover, in the human brain there is one network but many different types of neurons. Particular functions are performed by neurons of particular types, and so a group of neurons of one type may link to neurons of different types in a certain predetermined ratio. The different types of neurons look and behave differently. Additionally, the links between neurons are permanently either inhibitory or excitatory, and throughout the learning process, although the strength of the links changes, the type of connection does not. Applying this model in an ANN would mean having different neurons with, for example, different squashing functions in the one network, and also setting the sign on the weights of an ANN as fixed throughout the learning.

The human genome does not use a direct encoding for the neural connections in the brain [RS00]. Despite not knowing how the brain is encoded, we can be sure of this simply because the amount of data stored in the genome is much too small to store enough information for a direct encoding. Therefore, the encoding must be indirect. Determining the exact information encoded in the genotype is still an active area of research [RS00].

Finally the learning process in the brain is very different from that used in ANNs. Back propagation, as most ANN learning algorithms, is concerned with global learning – the result from the output is checked for correctness and the error is propagated back through the entire network. Learning in the brain is local, with each neuron only relying on information available directly to it. There are many learning rules, but in the simplest version links become stronger between neurons that have a correlation between their outputs, and weaker if there is no correlation. This sort of learning is not random, but the random learning due to mutations in weights in evolutionary approaches is at least local. It is interesting that this learning technique was originally proposed for ANNs in 1949 by Hebb [Heb49] but proved to be completely ineffective for machine learning.

4.4 Computational Limitations

Genetic algorithms are intrinsically a very computationally intensive class of algorithms for training machine learners. Whereas most approaches train a single individual (learner) through many iterations, genetic algorithms train an entire population of individuals over many generations. An ideal genetic algorithm would effectively train the individuals in parallel, with useful features evolving independently in each individual that survives, and so making use of all the computational work. However, as half of all individuals are discarded at every generation, much computational work has to be wasted. Additionally, genetic algorithms work with the encoded form of individuals, so every evaluation of an individual must be preceded with the decoding of that individual. This can be especially expensive for indirect encodings.

Despite this computational difficulty, I believe that the primary goal of a neural network is to give correct and generalisable results, with speed of training only a secondary issue.

As a final note, genetic algorithms are intrinsically messy. Most scientific research focuses on finding the simplest patterns, equations and algorithms which describe a given phenomenon. By Occam's razor, the simplest solution is the most probable. Methods like back propagation, and Alopex, are certainly cleaner and simpler, even if we just look at the length of the programs which implement these algorithms. Nevertheless, this does not mean that there does not exist a clean yet effective genetic algorithm for ANN learning, nor does it mean that there exists a simple method which can learn as well as is required to build true artificial intelligence. Perhaps evolutionary processes are complex simply because they must be to be effective.

Chapter 5

Approaches to Structured Data

Machine learners typically restrict the type of data that can be provided to the learner. Recent work has attempted to widen the range of in that can learnt from, in particular to include structured data. This chapter introduces recent research into dealing with structured data.

5.1 Motivations

Almost all ANN learning algorithms assume the input to be of simple form: a vector of real numbers of a pre-determined length. The real numbers in these inputs represent various features of the example given to the ANN and have to be computed by a person. Real world data, on the other hand, especially in areas where machine learning techniques can be most useful, simply cannot be easily represented by a single vector of pre-determined length.

Consider the predictive toxicology challenge [HKKS01], where for a given molecule, the goal is to predict if it is carcinogenic or not. Most current machine learning techniques cannot process a molecule directly, and hence the first stage of the challenge is described as "calculating chemical descriptors and feature construction". However, we could represent the molecule in its natural form, as a graph. This would allow more of the molecule's natural properties to be presented as input to the ANN. It would also allow larger molecules to be represented by more information, unlike feature vector approaches, where the length of the feature vector is fixed irrespective of the size of the training example. Within the graph representation, an expert could still annotate particular parts of the graph with additional features, but this is no longer an essential first step. This approach of dealing directly with structured data allows the expert to concentrate on the representation of the examples, and the meaning and form of the background knowledge provided. The expert's role is no longer a question of turning a molecule into a vector of numbers, but it is rather that of annotating the molecule with other potentially useful information.

We could go a step further and regard a learning system as a black box that is given a molecule and simply returns an answer. In this sense, it could be argued that given the molecule and drawing on his or her experience, the expert is not providing any new information, but is rather using his or her knowledge to pre-process the example in the same way that a machine learner could. We could envisage the entire system of the expert plus machine learner being replaced by a larger machine learning system that learns to perform the role of the expert as well. However, expecting such a system to perform as well as one where the expert can annotate examples would make the learning task tremendously more difficult. Moreover, it would probably require training on not just the problem at hand, but also on other data similar to that seen by the expert during his or her education. It would be like expecting an untrained person to, for example, see molecules labelled as carcinogenic or not, and use them to discover all the necessary principles of chemistry and physics, without access to any textbooks or experimental apparatus. For this reason, the expert is a necessary part of the machine learning process, although a structured approach allows him or her to perform his or her task more easily.

The background knowledge problem presented here is clearer if we consider the problem of overfitting. If the concept we would like the learner to learn is a very subtle function of the training data, it may be easier for it to overfit the data than to learn to classify the data in a generalisable way. We usually make the assumption that overfitting occurs when the learner has been learning too long and has passed the "correct" solution. But if the "correct" solution is more complex than an overfitting ANN, it is very unlikely that the learner will find a generalisable solution at all. In this sense, the role of the expert in providing background knowledge is to make the target function which the learner must find a simpler function of the input.

As another example where structured data is useful, consider n-parity type problems, where given a vector of length n, containing just zeros and ones, the learner is to classify the vector as containing an odd or even number of ones. If $n = 2$, this is the classic XOR calculation task. For larger n, the problem quickly becomes much more difficult – Tesauro and Janssens demonstrate that the time to solve the problem grows exponentially in n [TJ88]. As Duch [Duc97] states:

> There is a simple way to solve the parity problem in $log_2 n$ time using a hierarchical multilayered structure, where at each level the nodes are designed to solve the two-input parity (i.e. XOR) problem.

Using the structured approach, we can define parity problems as a class of problems where sets of binary values are classified into one of two classes. If this set is processed one element at a time, as shown for $\{1, 0, 1\}$ in Figure 5.1, a simple XOR is sufficient to solve the problem.

These two examples demonstrate two features we would like to achieve in our machine learning model:

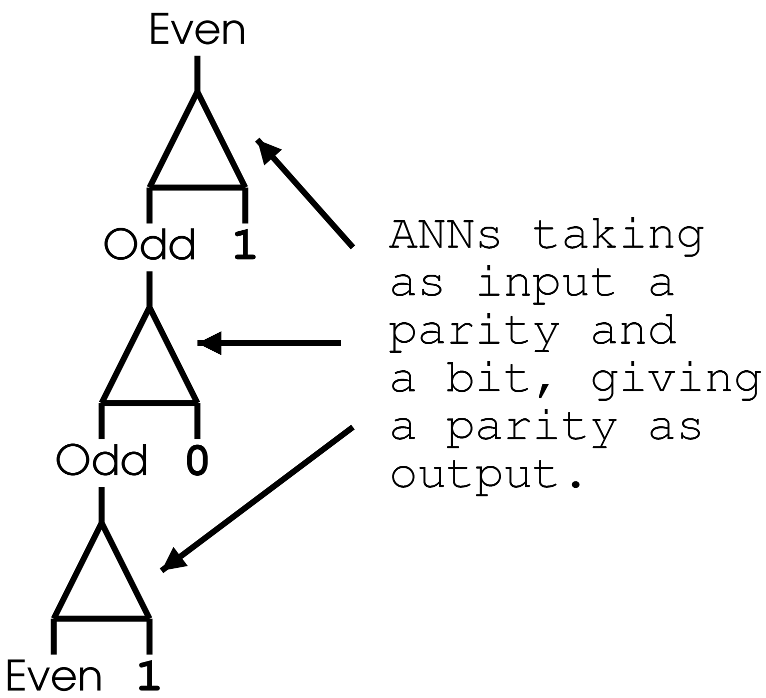

Figure 5.1: n-parity as a structured problem processing $\{1, 0, 1\}$.

1. The ability to work with structured data types.

2. The ability to recurse over similar input.

5.2 Knowledge Representation

In most cases, machine learning data is simply taken as a feature vector. Between the data observation phase and the processing phase there is the notable omission of a design phase. In other areas of computer science, for example when dealing with databases, a conceptual model is always constructed before real world data is plugged into an application. This step forces us to take a step back and carefully observe the data that we are dealing with. In this context, a suitable knowledge representation is an essential part of any machine learning process, and such a representation can be derived by studying structured data.

In the remainder of this chapter, I present approaches to structured data that preceded and have led to the representation method using in the GALSINE algorithm.

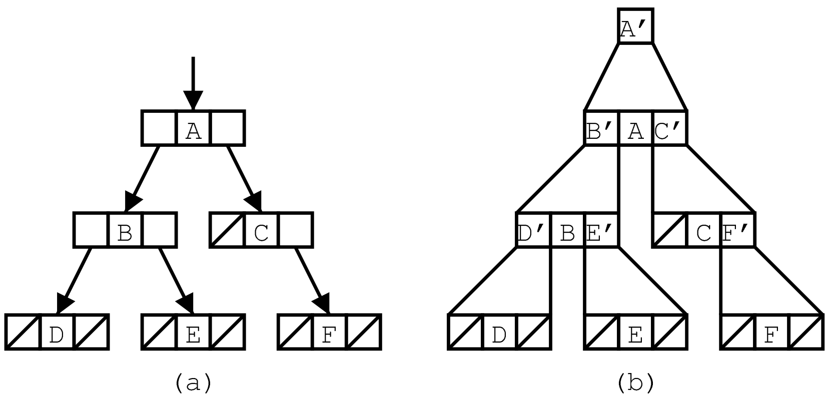

Figure 5.2: (a) A traditional binary tree representation and (b) a possible neural network representation.

5.3 Reduced Representation

Consider trying to present a binary tree as input to an ANN. Typically, a binary tree is represented in memory using a pointer structure such as shown in Figure 5.2 (a). In the computer's memory, the entire binary tree is represented by the single pointer to the value 'A'. Similarly, subtrees are represented by the pointer to their root nodes. In this way, following earlier work dating back to 1984, Hinton proposed in 1990 [Hin90] the idea of a reduced representation. This approach consists of associating a meaningful value, rather like a tag, with each link to allow an individual element to represent an entire tree. This can apply to any recursive structure where we can allow an individual element to represent the information contained under it. If we allow each little box in Figure 5.2 (b) to represent a value or a set of values, and the trapezoidal shaped areas to represent (distinct) ANNs with inputs at the bottom and outputs at the top, we are presenting the binary tree as input to the ANN, which outputs a reduced representation of the tree. If we were to calculate the inverse of one of the ANNs, say the left subtree of A, we could recover $\{D', B, E'\}$ from B' and so B' both summarises the information in the subtree below it and represents that subtree. Note that B' is not a feature vector which only contains some information about the subtree.

Immediately expanding on this work, on the following pages in the same journal, is an implementation of such an architecture by Pollack [Pol90], the Recursive Auto-Associative Memory (RAAM). Pollack provides a convincing argument for building ANNs which can fully represent the information being stored by noting that Minsky and Papert wrote a few years earlier [MP88], that "no machine can learn to recognise X unless it possesses, at least potentially, some scheme for representing X". In order to ensure a structure is being completely represented, Pollack proposes two components for working with symbolic trees: A compressor which can compress a small set of fixed-width patterns into a single such pattern, and a reconstructor which can take a compressed pattern and expand it into the original set of patterns. The compressor

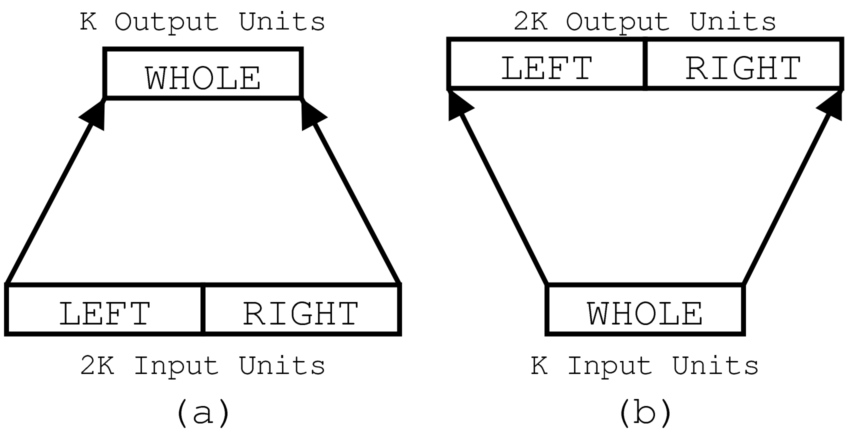

Figure 5.3: (a) A compressor and (b) a reconstructor for encoding and decoding binary trees.

can be applied recursively, bottom up, for example on subtrees in a binary tree to compress an entire tree into a single set of values. The reconstructor can be applied top down to recover the original tree and should recognise when it needs to call itself recursively to expand sub components.

Pollack notes that initially the compressor and reconstructor are simply hypothetical mechanisms. He states that the functioning of such devices is unclear, and it is not obvious how the tree could even be represented by a vector. However, he assumes that both the functioning and the internal representation can be left to the ANN to devise during training, with us simply representing the binary trees with k-tuples of real numbers. In this way, he proposes two basic neural network components, as shown in Figure 5.3, (a) a compressor and (b) a reconstructor. These components convert a representation for the right and left subtrees of a binary tree to and from a compressed version respectively (Pollack considered binary trees where information is only stored at the leaves of the tree).

This gives us a workable model that allows a neural network to be trained to take a tree as input, and step by step reduce it to a simple representation, in a fully reversible manner. In order to train such compressor and reconstructor networks, we can connect them end to end, with data being input into the compressor, passing through the compressed form and then back into the decompressed form. Pollack argued that such training should be possible with simple back propagation because this model is effectively equivalent to the encoder problem which had already been solved. The encoder problem consists of an 8 bit input which is compressed into 3 bits and then expanded again [AHS85]. Pollack subsequently used this method to compress and decompress not only trees but also stacks, sentences and relationships between objects.

As an extension to the RAAM model, Sperduti [Spe93] proposes the Labelling RAAM (LRAAM) algorithm, allowing for labelled directed acyclic graphs to be represented within an ANN. He considers that each node in a graph can be represented by a label and pointers to all the nodes which that node links to. To train the encoder, simply

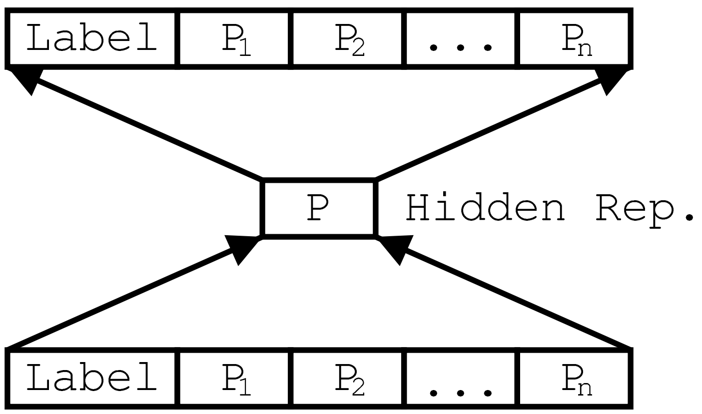

Figure 5.4: The framework used for training an LRAAM for encoding a directed graph.

connect an encoder to a decoder and train it to the identity function as in the case of RAAM. Figure 5.4 shows the ANN setup used. In the LRAAM approach, there can be any number of pointers associated with each label, although the number of pointers is fixed for a given network, and the hidden compressed form, H, is of the same size as a pointer P. This allows the encoding to be used recursively to encode an entire graph. A number of applications of LRAAM have been explored by Sperduti and his group. For example, Lonardi et al. [LSS94] were interested in using LRAAM for image compression. The LRAAM architecture was also shown to be able to represent arbitrary directed graphs.

Inspired by the LRAAM architecture, Goller and Küchler [GK96] proposed the folding architecture by replacing the decoder in LRAAM with a standard classifier ANN, as shown in Figure 5.5. An alternate approach would have been to fully train an LRAAM network, fix its weights and remove the decoder, and then connect the classifier to the hidden representation. This would have guaranteed a complete and unique hidden representation, but the Goller et al. note that "for many inference tasks unique representations are not necessary as long as the information needed for the inference task is represented properly". Instead of training the encoding part of the network using a decoder, the identity function and back propagation, they simply propagate the classification error back from the classifier through the encoder.

To train the folding architecture, Goller and Küchler use back propagation through structure, in a similar way to back propagation through time is used for recursive ANNs [Wer90]. This extension to back propagation takes into account the problem of training a recursive network where the same weights can have a different error gradient in different places, and so a different Δw. The folding approach is illustrated in Figure 5.6 (b), which shows the ANN used to encode the tree shown in Figure 5.6 (a). The network W is repeated five times, so when the tree is processed recursively, there are five error gradients for each neuron in W. The error gradients are added, giving a total adjustment to the weights of each neuron.

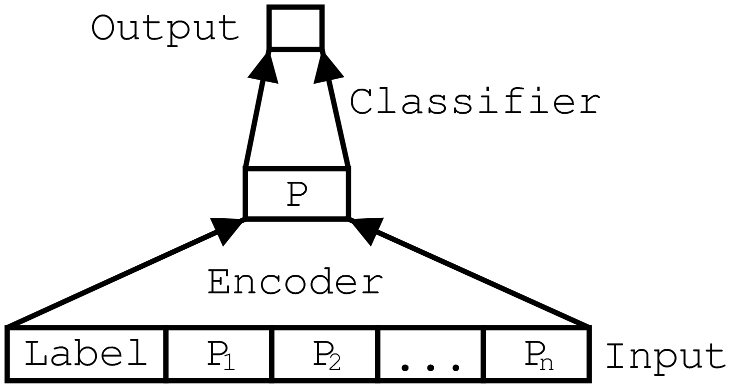

Figure 5.5: The folding architecture replaces the decoder of LRAAM with a classifier.

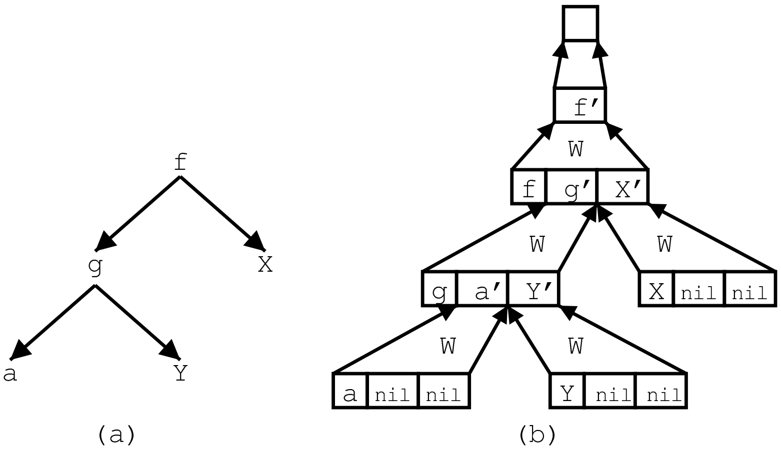

Figure 5.6: To compress the tree (a) and perform the classification, the folding architecture shown in (b).

As a final generalisation, Sperduti and Starita [SS97] mathematically define first order generalised recursive neurons. They expand upon the recurrent neuron (where output of a neuron one time step earlier is used as input to a possibly different neuron) to describe a generalised recursive neuron where the output of a neuron can be used as input into another neuron when determined by the graph topology. This is a formalisation of the approach used in LRAAMs and the folding architecture.

Sperduti and Starita consider how these recursive neurons can be used to encode both cyclic and acyclic graphs, and how a number of classic training algorithms such as back propagation and cascade correlation can be modified. These result in back propagation through structure and cascade correlation for structures respectively. They apply this model to a number of classification problems and show that it can be applied to a large number of areas.

5.4 Term Reduction Networks

The folding architecture presents a practical way to represent directed acyclic graphs using ANNs, but it does not give us the framework to represent general structured data. Taking a step back to observe a mathematical model, we find that examples can be represented using a polymorphically typed higher-order logic [Llo02], growing from Church's theory of types [Chu40] and building on more recent research.

In order to build up structured data, it can be shown that, taking certain terms called basic terms, we have an expressive knowledge representation for structured individuals [Llo02]. We can recursively define the set of basic terms $\mathcal{B}$ as consisting of three classes of terms: basic structures, basic abstractions and basic tuples. I will present these classes through examples and at the end will state a definition for each.

1. Basic structures

   Consider the standard *cons* list constructor which has the signature

   $$cons \;:\; a \to List\; a \to List\; a,$$

   where a is a parameter, that is, a typed variable. We also need a base case, the empty list, with the signature

   $$[\,] \;:\; List\; a.$$

   Then we can represent lists using the cons constructor recursively, for example representing $[1, 2, 3]$ as $(cons\; 1\; (cons\; 2\; (cons\; 3\; [\,])))$. Other structures can be represented in an analogous way – in a similar way to *cons* being defined for lists, we can define the structure of general trees, where there is a node argument and any number of subtree arguments. The simplest case for basic structures are numbers, where there are no arguments. For example the integer 42 has the signature

$$42 \; : \; Int.$$

It helps in understanding basic structures by noticing they are similar to structs as in the C programming language. For example we could represent a car using the constructor *Car* with parameters make, model and year of production. In the general case, the signature data constructors takes the form

$$C \; : \; \sigma_1 \to \ldots \to \sigma_n \to (T \; a_1 \; \ldots \; a_k),$$

where σ_1 through σ_n are types and a_1 through a_k are the parameters and T is a k-ary type constructor.

2. Basic abstractions

   We also want be able to represent sets and multisets. These are built using basic abstractions. Consider the set $\{1, \, 2, \, 3\}$, which can be represented using the λ-calculus expression

$$\lambda x. \text{ if } x = 1 \text{ then } T \text{ else if } x = 2 \text{ then } T \text{ else if } x = 3 \text{ then } T \text{ else } F, \tag{5.1}$$

   where T is True and F is False. In this way, a T means the item is in the set, and an F means the item is not in the set. Notice the 'else F' at the end of Equation 5.1. This is a default value, returned when none of the 'if' conditions are matched. The default value at the end of an expression is called the *default term*.

   By replacing T and F in Equation 5.1 with numbers, we can represent multisets. For example, the multiset containing two 1s and one 3 is represented by:

$$\lambda x. \text{ if } x = 1 \text{ then } 2 \text{ else if } x = 3 \text{ then } 1 \text{ else } 0. \tag{5.2}$$

   In this case, 0 is the default term. It is important to note is that for any one set there are an infinite number of possible λ-calculus representations, because the if conditions can be rearranged and they may also be redundant information. For example, we could add 'if $x = 5$ then F' to Equation 5.1 or 'if $x = 5$ then 0' to Equation 5.2. In order to remove this possible redundancy, and make the representation of a given basic abstraction unique, we need to order the terms and ensure the default term does not appear anywhere in the expression except after the last 'else'.

   In the general case, basic abstractions can be written as:

$$\lambda x.\ \text{if } x = t_1 \text{ then } s_1 \text{ else if } \ldots \text{ else if } x = t_n \text{ then } s_n \text{ else } s_0, \tag{5.3}$$

where s_0 is the default term.

3. Basic tuples

   The third basic type of data is the tuple, consisting of a fixed number of ordered elements. This is exactly the intuitive definition of a tuple.

We can write the inductive definition of the set of basic terms mathematically as shown below. The definition given omits a number of subtle complexities covered in [Llo02]. Given the set of default terms $\mathcal{D}$, the set of basic terms $\mathcal{B}$ is:

1. If C is a data constructor having signature $\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow (T\ a_1\ \ldots\ a_k)$ and $\forall i \in [1, n]$, $t_i \in \mathcal{B}$ with $n \geq 0$, then $C\,t_1\ \ldots\ t_n \in \mathcal{B}$.

2. If $\forall i \in [1, n]$, $t_i, s_i \in \mathcal{B}$, $s_i \notin \mathcal{D}$, $s_0 \in \mathcal{D}$ and the terms t_i are ordered, then $\lambda x.$ if $x = t_1$ then s_1 else if $\ldots$ else if $x = t_n$ then s_n else $s_0 \in \mathcal{B}$.

3. If $\forall i \in [1, n], t_i \in \mathcal{B}$, then $(t_1, t_2, \ldots, t_n) \in \mathcal{B}$.

We can now use these basic terms to construct structured data types. Associating each basic term with a general ANN to produce a hidden representation from the term, we see that this is a generalisation of the folding architecture. These general ANNs are called *Term Reduction Networks* (TRNs), and provide a method of reducing a structured individual to the output of a single top level TRN. In the folding architecture, at this stage the output is considered the hidden encoding of the structured data and is presented to another ANN for data classification. In contrast, the TRN approach previously presented by Ng [Ng01] does not use this classifier. Instead, it implicitly attaches the simplest possible classifier, namely a step function whereby the outputs are rounded to 0 or 1, and this is used as the basis of classification. In this thesis, I evaluate both approaches.

The three types of Term Reduction Networks are shown in Figure 5.7, with network (a) used for basic structures, network (b) used for basic abstractions and network (c) used for basic tuples. Each network reduces the structure to a hidden representation. The TRN for basic abstractions is different from the other two basic terms, encoding abstractions recursively. This leads to each "if $x = t_i$ then s_i else" being encoded by one instantiation of the abstraction network. The hidden representation for the entire abstraction is built up one element at a time. Such recursion can also occur with basic structures, where the structure is recursive, but by no means is it universal as it is with basic abstractions.

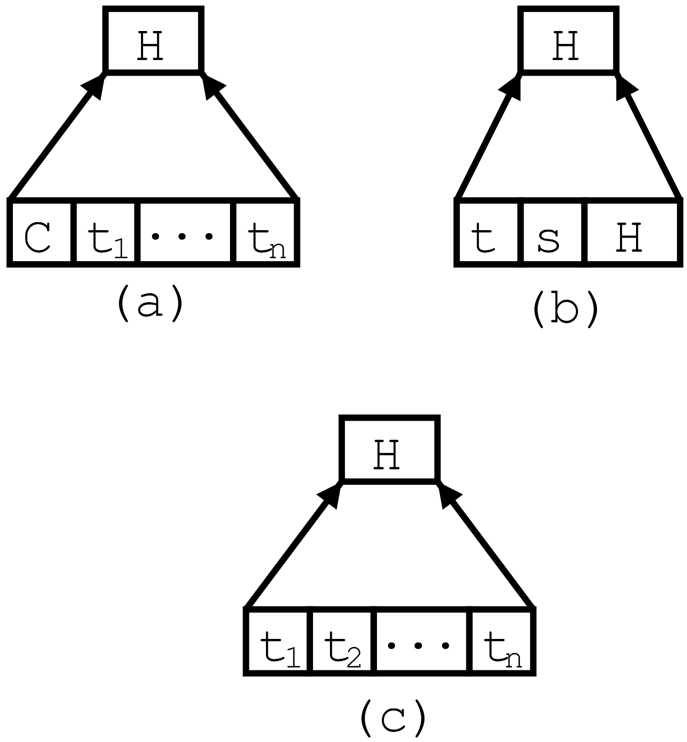

Figure 5.7: The three basic terms are constructed using TRNs of the structure shown for (a) basic structures, (b) basic abstractions and (c) basic tuples. Each network reduces the structure to a hidden representation H.

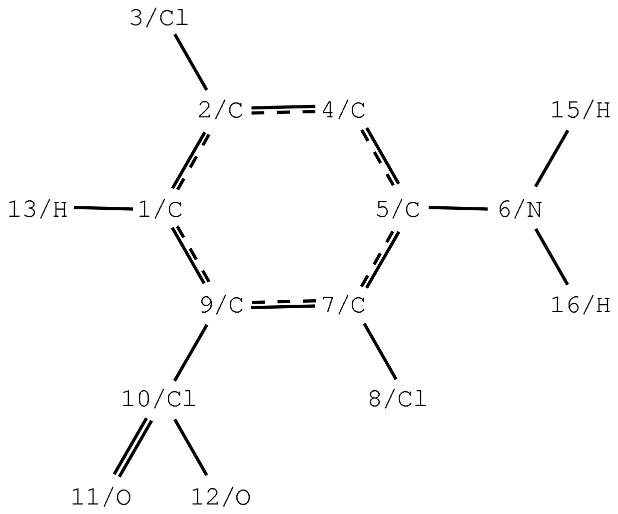

Figure 5.8: Molecule TR025 from the predictive toxicology challenge.

5.5 Representing Graphs

The TRN approach has one limitation which results from complex structures being built using a knowledge representation structured like a tree. Let us consider a very typical graph seen in real world data, namely the graph representing a molecule. Molecules are graphs with atoms at the nodes and links representing bonds between the atoms. An example of such a graph is given in Figure 5.8. The most common mathematical definition for a graph is $G = (V, E)$ where $V = \{v\}$ is the set of vertices and $E = \{e\}$ is the set of edges. The vertices can be labelled with the type of atom, but in order to label the edges we need to describe each edge as connecting two particular vertices, so to differentiate identical atoms at different vertices, we need to additionally label each vertex with a unique number as shown in Figure 5.8.

When we consider how this graph is represented using basic terms, it is clear that if we follow the mathematical notation, the information about the vertices and the information about the edges is separate until it is processed by the graph data constructor. This constructor will see a hidden representation of all edges, and a hidden representation of all vertices, as shown in Figure 5.9. Correlating information about particular vertices to the related information about particular edges linking those vertices seems all but impossible. As an example, the term which computes the hidden representation of the vertices will only have as input the vertex. In particular, it will not have information about the edges from that vertex. Yet, in the case of molecules, is known that the edges determine important properties of the vertices.

There are other possible representations of graphs, for example as an adjacency matrix, but all these seem to be subject to the same problem described above.

In aiming to minimise the importance of additional background knowledge provided

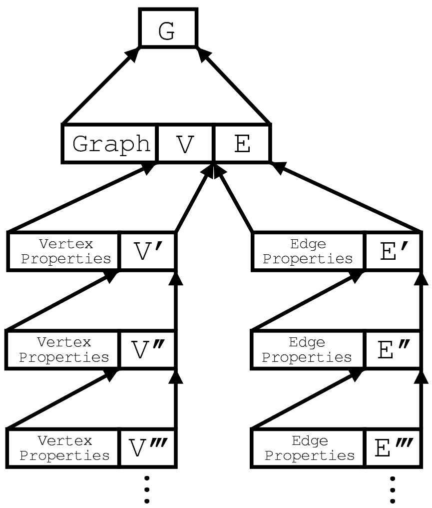

Figure 5.9: TRNs which take as input graphs represented using the standard G = (V, E) format.

by an expert, it would be interesting to design a learner that is able to access databases of knowledge, much like an expert accesses reference books. For example, consider a learner that recognises when it has been given a molecule, and subsequently looks up features about it. This learner would be able to apply general background knowledge to particular problems. Such an approach would allow it to select useful features for itself, similarly to that described by Hallinan and Jackway [HJ98]. However, the difference is that the features could be determined by the learner specifically for each example.

There is a very large body of research in methods which allow background knowledge to be provided in this way. However, this knowledge is usually provided as a library of functions available to a rule learning algorithm, as part of a learning method known as ILP, Inductive Logic Programming (see Chapter 10 of [Mit97] for a background in ILP). It could be possible to link in such background knowledge as part of an automated evolutionary method for ANNs while allowing the background knowledge to have as general a form as possible.

Chapter 6

Other Related Work

There has been much research in genetic algorithms for artificial neural networks. A large number of researchers have explored many different approaches to the problems presented in this thesis. In this chapter I will briefly describe some of that work.

Schmidhuber and Hochreiter [SH96] present a three page technical note about the difficulty in selecting good benchmark problems for ANN learning. While studying long time lag recurrent ANNs, they notice that they can achieve faster learning on a number of often used benchmark problems simply by guessing weights. They propose an algorithm consisting of repeated random initialisation of the weights in an ANN, until the resulting ANN happens to classify all the training examples correctly. Such an algorithm could be regarded as an extremely simple genetic algorithm, although presumably solving problems by random guessing becomes much less effective when the solution requires larger ANNs. However, it does show why benchmark problems should be chosen very carefully.

Belew et al. published a well written overview of many aspects of genetic algorithms, and present an encoding method for ANNs based around a binary bit string crossover operation [BMS90]. They start with a crossover operation where a substring of the representation of two individuals is simply swapped. For example, the crossover of two individuals encoded as 00000000 and 11111111 might be 00011110 and 11100001. For such a crossover operation to be useful on ANNs, they devise an appropriate encoding scheme for ANNs. Of particular interest in their work is the hint of isolating the population into independent groups, where they notice that "whichever solution is discovered first comes to dominate the population and resist alternatives". The next step is the idea of isolating the different possible alternative solutions through speciation, which I will discuss further in Chapter 10. They also discuss many of the difficulties in preserving useful features in ANNs when they are crossed over.

Many other genetic algorithms are explored, of which I here mention but a brief selection. Yao and Liu present an algorithm for constructing ANNs based on mutations, then training based on back propagation [Yao98]. Blanco et al. present a genetic algorithm for training recurrent neural networks and apply it to recognising a regular fuzzy grammar [BDP00]. Evolutionary algorithms for reinforcement learning are studied by Moriarty et al. [MSG99]. Mandischer recently studied in depth the effectiveness of a particular class of evolutionary algorithms on a number of benchmark problems and compared their performance to back propagation [Man02]. On a more applied side, Gallagher and Beer [GB99] explore the potential of using genetic algorithms to train an ANN to control a robot, by experimenting with a simulation.

Despite this continued research, a second paper by Mandischer el al. [MGU99] demonstrate the continued lack of effective genetic algorithms. They apply ANNs to experimental science, being interested in regression rather than classification. The goal is for an ANN to learn to predict the heat of vaporisation of molecules. In order to present these molecules to a traditional ANN, they extract feature vectors from the molecules. In their work, they experiment with ANNs trained by back propagation and by a genetic algorithm. Following a number of trials, they find a good set of feature vectors and a network architecture where they achieve good results using one of the hand built ANNs trained by back propagation. In the ideal scenario, using an automated learner much like GALSINE , we could hope that applying machine learning to such problem is simply a matter of representing the data in a standard way and giving it to the learner.

Part II

The GALSINE Learning Algorithm

Chapter 7

Algorithm Overview

The algorithm I present here starts with a basic genetic algorithm as described in Chapter 4. To that general evolutionary model, three additional complexities are added. In the interest of readability, I will refer to this algorithm as the GALSINE algorithm, Genetic Automated Learning on Structured Individuals through Neuro-Evolution.

- Speciation
  When selecting which individuals survive in a genetic algorithm, the individuals' fitness need not be considered in a single group. We can consider the individuals as belonging to different "species" that are not directly compared to each other. Instead, the best individuals in each species survive at each iteration. This draws directly on work presented by Stanley and Miikkulainen [SM01], and is further discussed in Chapter 10.

- Structured Data
  By applying the Term Reduction Network approach discussed in Chapter 5, the GALSINE algorithm can classify structured data. The exact implementation is described further in Chapter 8.

- Automated Data Analysis
  The initial step of any ANN learning is usually involves setting the initial parameters of the population by hand, such as the length of the input vector, learning rates, ANN size and so forth. I propose a model which can analyse its own input and generate a population of individuals with minimal human intervention. Moreover, GALSINE allows some of the learning parameters to be automatically tuned. The analysis of data is further discussed in Chapter 8 while automatic parameter tuning, being much more experimental, is presented together with results in Chapter 13.

Together, these design features give a slightly modified flow of the algorithm, as shown in Figure 7.1.

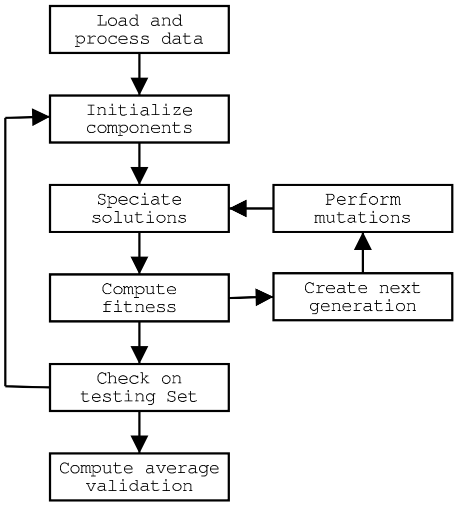

Figure 7.1: The steps involved in the GALSINE algorithm, namely automated analysis of the learning data, processing of structured input, speciation of individuals and 10-fold cross validation.

The GALSINE algorithm first processes the data it is provided with in order to determine the structure of the data. This data is then split into ten parts to allow 10-fold cross validation of the results to demonstrate generalisable learning [Koh95].

In 10-fold cross validation, each fold involves taking eight parts of the data as training data, one part as testing data and one part validation data. During each fold, a different validation set is used. GALSINE starts by creating a population of minimal ANNs. These individuals are separated into species, and the best ones in each species are chosen to reproduce to create the next generation of individuals (using the training data). The individuals in the next generation have mutations performed on them, and the entire process is repeated for a predetermined number of generations, or until a sufficiently fit individual is found. When the process if finished, the best individual found is checked using the validation data, giving the validation correctness.

GALSINE is implemented in C as 28 modules in approximately 6000 lines of code, including blanks lines and comments. Help is provided on the command line summarising all optional parameters. Assuming a data file is structured as described in Chapter 8, the program can learn from that data without any modifications. The variable parameters are passed via the command line. The source code is available on request.

Chapter 8

Structure Analysis

Before any learning can occur, we must build ANNs that can process the input as it is provided. These ANNs can be constructed automatically through observation of the data. In this chapter, I present how the training data is analysed and used to construct individuals.

8.1 Representation of Basic Terms

In the Term Reduction Network approach, data is constructed using three kinds of basic terms, as discussed in Chapter 5. Using a notation similar to standard uncurried mathematical notation, we can write data in a simple form. Each of the basic terms is written slightly differently, as shown below.

Basic structures are written as follows:

$$(StructureName\ \ SubTerm1\ \ SumTerm2\ \dots\ SubTermN\ ),$$

where *StructureName* is a unique name for each type of structure seen. The sub-terms in the structure can be of any type, although the number and types of the sub-terms must be consistent for each specific structure.

Basic abstractions are written as follows:

$$\{SubTerm1\ \ SubTerm2\ \dots\ SubTermN\ \},$$

where, as discussed in Chapter 5, the sub-terms can be of an arbitrary type, but within one abstraction the sub-terms must all be of the same type. Internally to GALSINE , basic abstractions are given a name based on the type of the sub-terms. For example, an abstraction of integers would be given the name *SET_OF_(I)*.

Finally, basic tuples are written as:

$$[SubTerm1\ \ SubTerm2\ \ldots\ SubTermN\,].$$

From the original definition of basic tuples, we know that the sub-terms can be of any types, although with any particular tuple type the sub-terms at a given position must always be of the same type. Once more no name needs to be given, as GALSINE determines one from the types of the sub terms. For example, a tuple containing two integers followed by an enumerated type would be given the name *CONS_OF_(I_I_E)*, where "cons" stands for constructed type.

8.2 Representation of Simple Types

In addition to basic terms, inputs can be integer, real or of enumerated type. Integer and real inputs are treated identically: After all the training data has been observed, and limits of the values have been determined, the values at each position of every data type are independently normalised to the range [0, 1]. Input of numerical type is given to ANNs as a single value. Although normalisation is used in ANN learning to give all numerical input types uniform importance, if future unseen input examples have values outside these ranges then the learner will process the input as expected. There is no requirement for numerical input to ANNs to be in the range $[0, 1]$.

Enumerated types are treated differently, in that each possible value of the enumerated type is treated as distinct. Each enumerated position in every basic structure or tuple has its own enumerated type created. To illustrate how an enumerated type would work, assume the training data has an enumerated type with possible values "Dog", "Cat" and "Rabbit". These are not implicitly ordered by GALSINE . When presenting an enumerated input to ANNs, there is one input neuron for every possible value, so in the example above there would be three input neurons. If a particular training example has a value of "Cat", the first and third inputs would be set to 0 and the second input would be set to 1.

8.3 Dynamic Construction

Having simple types and basic terms, we can see how structured type inputs would be given to an ANN which reduces that particular type. However, we must also decide how to specify the hidden representations used in the structured approach. At its most basic, the hidden representation must be a vector of real numbers so that it can be treated as output from an ANN, and given as input to another ANN. Also, as the basic terms have sub-terms that can be of any type, the structured types must

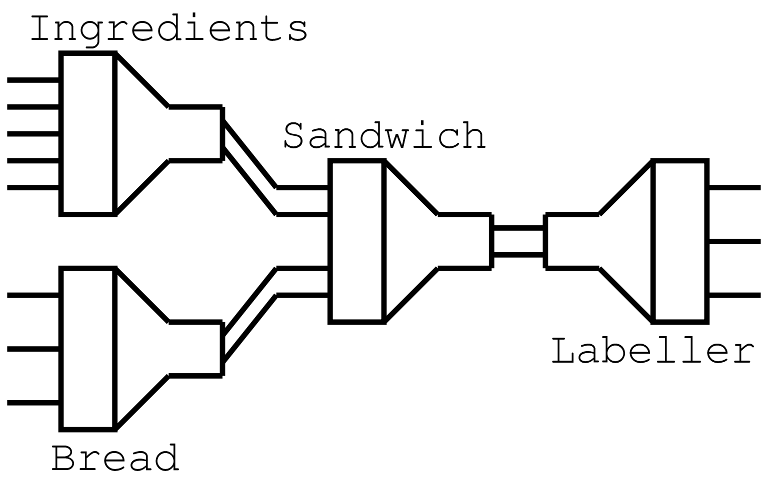

Figure 8.1: An example of an individual made up of three components and a labeller, with the hidden representation length $P = 2$ and the labeller giving the output as one of three possible classes.

"fit together", whereby the hidden representation vector of each data type present must be of a consistent length (although different data types could be allowed to have hidden representations of different lengths).

The simplest approach to solve this, which is the approach GALSINE uses, is to make the hidden representations of all structured types be of an identical fixed length. By default, this length is set to $P = 2$ values (P stands for output pins), hence P neurons are used to input a structured type, and there are always P output neurons on every term reduction network.

The final stage seen in folding architectures is the classification of the hidden representations. Hence, the classifier always has P inputs. As this model allows for a generalised classifier, the output is always of an enumerated type. This enumerated type can have an arbitrary number of values, and so the classifier has the same number of outputs as there are different classes of training examples. Internally, the classifier is called the *Labeller* because it labels each hidden representation as belonging to one of the possible classes of inputs.

Consider an individual that is to be able to process a structured training example. It must contain the definition of a Term Reduction Network for each of the data constructors that may be in the example, as well as a definition for the labeller. Throughout the rest of this thesis, I refer to each of these ANNs as *components* of each individual. The components work together to create a single hidden representation, an example of which is shown in Figure 8.1. A particular set of components is never broken apart and single components are not exchanged between individuals.

8.4 Load and Process

Before any learning can take place, loading the training, validation and testing data needs to be done in two passes. The first pass involves recognising the data constructors and types present, and the second involves loading the data into memory, in such a way that the structure is preserved. The structure of the data is analysed automatically so as to allow it to be provided with minimal modifications. An alternate approach could have required the user to specify the grammar defining the data.

In the data file, each training example must simply be provided in the form:

$$< \mathit{Example\ Name} > < \mathit{Input} > < \mathit{Correct\ Label} >$$

or

$$< \mathit{Example\ Name} > < \mathit{Correct\ Label} > < \mathit{Input} >$$

Both orders are allowed as there is no point in forcing the format of the data files to be more restrictive. The example names need only be unique strings and are usually just integers or integers appended to a few characters. Similarly, the labels can be any string of characters, with each label representing one class of input.

The first pass is called the parsing phase, where each input is decomposed according to the notation described in Chapter 8. Given a training example, the data parsing routine recursively determines the basic terms used to construct it, as well as the type of any sub-terms present. Note that one feature of data types that the parsing routine recognises is where there may be two type names for the same sub-term (despite the requirement that the sub-terms at each position are always the same). This is best understood when looking at how recursive types are often defined. For example, a binary tree that stores integer values is usually defined as:

$$\mathrm{BinaryTree} = (\mathbf{Null}) \mid (\mathbf{BinTree}\ \mathrm{BinaryTree} \times \mathrm{Integer} \times \mathrm{BinaryTree})$$

In other words, a binary tree is either null or it has two subtrees and an integer value. This must be detected automatically when the data parsing algorithm sees an example such as:

$$(\mathbf{BinTree}\ \underbrace{(\mathbf{BinTree}\ (\mathbf{Null})\ 5\ (\mathbf{BinTree}\ (\mathbf{Null})\ 2\ (\mathbf{Null})))}\ \underbrace{10}\ \underbrace{(\mathbf{Null})})$$

In this case, GALSINE can see from the top level example that a BinTree has a BinTree as the first sub-term, an integer as the second sub-term and a Null as the third sub-term. However, take the first sub-term of the BinTree given above. When GALSINE looks at this sub-term, it sees that the first sub-term of a BinTree can also be Null, and that the third sub-term can also be a BinTree. In effect, Null and BinTree are two different data constructors for the same data type. The data parser allows for at most two differently named types to appear at each position.

In this way, the data parser steps through every level in every example given to it, constructing a data definition automatically. When this is complete, the parser performs one final step: the types must be linked, as they depend on each other. Assuming type A has sub-type B, type A must have links to type B to be able to find it quickly. These links cannot be created at the same time as type A since type B may not yet exist. The necessary links are made after all types that exist in the data have been analysed. Within the code, this is referred to as crosslinking.

After the parsing phase is complete, the data can be loaded. It is all loaded into structures in memory that both reflect, and are linked to, the data definition. While this method requires that all training data must fit in memory at one time, it is subsequently available quickly. By linking the data to the data definition, individuals can be rapidly constructed.

Chapter 9

Individual Representation

In this chapter I describe how GALSINE encodes individuals. As discussed in Chapter 4, the internal representation used for individuals lies at the core of any genetic algorithm. This representation determines the possible genetic operations.

9.1 Genotype Encoding

As mentioned in Chapter 8, each individual is made up of a small set of structural components. Each component has a list of input and output neurons, as well as a genotype. It is these genotypes that are directly manipulated by GALSINE .

The genotype of each component is described using a direct encoding scheme. The genotype is made up of a list of genes, an example of which is shown in Figure 9.1. This is a slightly simplified version of the encoding presented by Stanley and Miikkulainen [SM01]. Each neuron in each component has a unique index. Each gene represents one link in the network and contains:

- A source neuron index
- A destination neuron index
- The link weight
- An innovation number
- An enabled flag

The meaning of the source, destination and weight values is obvious. The enabled flag is present to allow links to be disabled without them being completely removed from the genotype. This serves the purpose of preserving the knowledge about this link learned by GALSINE in case a link is removed. If the same link is added again

Source = 1	Source = 2	Source = 1	Source = 2	Source = 5
Dest = 4	Dest = 4	Dest = 5	Dest = 5	Dest = 3
Weight = 0.2	Weight = 0.5	Weight = -1.2	Weight = 0.1	Weight = -0.4
Innov. = 1	Innov. = 2	Innov. = 3	Innov. = 5	Innov. = 6
Enabled = 1	Enabled = 0	Enabled = 1	Enabled = 1	Enabled = 1

Figure 9.1: An example of a genotype coding for a simple ANN.

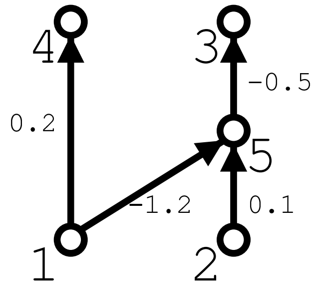

Figure 9.2: The example genotype in Figure 9.1 would decode to this phenotype.

at a later point in time, the old weight is used. Of the features present, the innovation number is the least intuitive.

Innovation numbers were proposed by Stanley and Miikkulainen to allow links which serve the same purpose in different individuals to be identifiable. This was done to help with the crossover operation. At the most basic level innovation numbers are simply integers. They give an order to the creation of new links across an entire population of individuals. If two links joining the same neurons in two different individuals are created in the same generation, they will fill the same role in both individuals. Hence, both links will be given the same innovation number. Similarly if one individual is replicated, then modified, at any later time the links dating back to the original individual will still share the same innovation numbers. Noticing this allows us to know which links serve the same purpose in two distinct and potentially different individuals.

Innovation numbers are used for speciation as described in Section 10.1, and the method of using the innovation numbers when performing crossover is described when in Section 11.2.

9.2 Individual Decoding

Each time an individual has its fitness evaluated, the genotypes of the different components must be decoded into the phenotypes, that is, the ANNs responsible for processing the structured data. Figure 9.2 shows the ANN which the genotype in Figure 9.1 would decode to. In a similar way, all the components of each individual having its fitness evaluated are decoded prior to that individual's fitness being evaluated. While relatively straightforward, this is a computationally time consuming process which is unique to genetic algorithms; traditional learners do not need to maintain a distinct genotype and phenotype. At every generation, all individuals must be processed as genotypes, decoded into a phenotype and evaluated. Then the phenotypes are removed from memory, before the genotypes are worked with again. In order to improve performance, each individual is only decoded once in each generation.

9.3 Population Initialisation

The first step in a genetic algorithm proper is the creation of an initial population. This population is the starting point of the genetic search and so must be very carefully selected. In his work, Stanley and Miikkulainen argue that we should start with individuals which are as simple as possible. They write:

> ...it is desirable to evolve minimal solutions; that way the number of parameters that have to be searched is reduced. Starting out with random topologies does not lead to minimal solutions, since the population starts out with many unnecessary nodes and connections already present.

I completely agree with this, as it is well known that simpler solutions are more generalisable and less prone to overfitting. Moreover, it is computationally faster to evaluate and evolve simpler individuals. For these reasons, based on Stanley and Miikkulainen's work, the components are initialised to be minimal. As mentioned in Chapter 8, each individual has a component for every data constructor necessary. These components have the number of input neurons determined by the sub-terms and a known number of output neurons. They are initialised with no hidden neurons, and all inputs are connected to all outputs. The weights are randomly selected, this being the only genetic diversity among the population.

As evolution progresses, the complexity of individuals naturally grows, as will be demonstrated in Chapter 13. However, this growth in complexity is guided by evolutionary pressure, increasing or decreasing at a rate that is most beneficial in solving the problem at hand.

Chapter 10

Fitness Evaluation

In order to pick the best individuals in a particular generation, the fitness must be measured using a fitness function. This fitness function must work for an arbitrary dataset, and must both promote correctness of the answers and measure the fitness precisely enough to allow the algorithm to recognise small improvements in individuals which are not reflected in the correctness.

10.1 Speciation

When a new neuron is added to an ANN through a mutation, initially the weights on the inputs and outputs are entirely random. This means that even if the extra neuron will improve the ANN substantially in the long run, initially it disrupts the previous behaviour. Usually this means the fitness of the ANN decreases, hence making it less likely for this individual to survive. For this reason, Stanley and Miikkulainen [SM01] propose to compare only the fitness of individuals which are similar to each other, that is, which belong to the same species. Therefore, individuals that are disrupted through mutation are not disadvantaged. A similar report of faster overall learning due to speciation is given by Andre and Koza [AK96], although they originally applied it as an aid in parallelising a genetic algorithm. When selecting the surviving individuals in a population, we can select the best half of individuals in each species.

Speciation also plays another role when considering combining different individuals. When two individuals are very different, the crossover operation is likely to be more destructive. If there is little in common to build off, it is difficult for the neurons and links from the parent individuals to be meaningfully combined. In limiting the number of crossover operations that occur between individuals in different species, we can reduce the number of hard cases. This both promotes the improvement of the unique features in each species and improves the general performance of the crossover operation.

To allow individuals to be speciated, a similarity measure between individuals must be defined. Comparing ANNs directly is not a simple task. For example, if ANNs

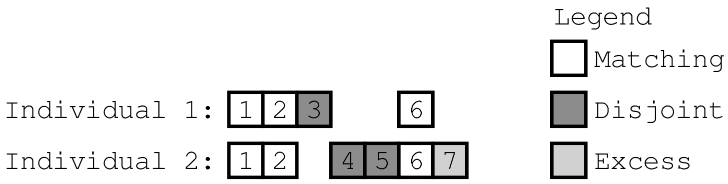

Figure 10.1: Comparing two individuals with genes with the innovation numbers shown, each gene can be classified as matching, disjoint or excess.

with similar structure but different neuron numberings are compared. To avoid this problem, we can return to the representation used for individuals. The genotype used includes innovation numbers which, as mentioned previously, effectively give a historical background into each of the links making up the ANN. When comparing two individuals we can look at the innovation numbers of links in order to determine how the links in the different ANNs are related to each other.

When two individuals are compared, the genes with the same innovation numbers are lined up. The genes in both are called matching genes. If a gene has an innovation number that is present in one individual but not in the other, and also the missing innovation number is within the range present in the other individual, it is called a disjoint gene. Otherwise it is called an excess gene. This is illustrated in Figure 10.1.

We can then define the similarity measure as follows:

$$\delta(I_1, I_2) = C_1 \frac{E_{I_1,I_2}}{N_{I_1,I_2}} + C_2 \frac{D_{I_1,I_2}}{N_{I_1,I_2}} + C_3 \overline{W_{I_1,I_2}}, \tag{10.1}$$

where I_1 and I_2 are two individuals, E is the number of excess genes, D is the number of disjoint genes, N is the number of genes in the larger individual and $\overline{W}$ is the average weight difference of the matching genes. There are also three constants, C_1, C_2 and C_3 which weight the relative importance of each of the terms. Whereas Stanley proposes weights of $C_1 = 1.0$, $C_2 = 1.0$ and $C_3 = 0.4$, I would argue that speciation should be based on the interactions between the different inputs to the ANN, and that structure is a much more important indication of this than the weights. In all my experiments $C_3 = 0$.

Species are stored as an ordered list, with each species represented by an individual selected randomly from the population in that species. Each new individual I_{new} is compared sequentially to the representative I_s of each species s. If $\delta(I_{new}, I_s) < \delta_t$, where δ_t is the speciation threshold, the individual I_{new} is judged to belong to species s.

To illustrate this process I will work through a short example. Consider individuals I_1 through I_8. Initially there are no species. When I_1 is considered, it is placed in the first species and is the representative of that species. Next, I_2 is compared to I_1, as the representative of the first species. Say that $\delta(I_1, I_2) < \delta_t$. In that case, I_2 is placed in the same species as I_1. Then say I_3 is considered, and $\delta(I_3, I_1) > \delta_t$. A second species is created and is represented by I_3. Note that I_3 may or may not be more similar to I_2 than to I_1, but this is never tested. By keeping the list of species ordered we ensure each individual has its species determined deterministically.

In this way, say the individuals are placed into three species $\{I_1, I_2, I_5\}, \{I_3, I_4, I_7\}$ and $\{I_6, I_8\}$. In the next generation, the individuals in these species are combined and mutated into new individuals. These new individuals are then compared one by one to the representative individuals from the *previous* generation to determine their species. After the new individuals have been speciated, a new representative is selected randomly for each species. This results in a continuity in the existence of each species, and allows the progress of a species to be watched independently of the particular individuals in the species.

In the speciation algorithm we could either fix δ_t, as Stanley and Miikkulainen **??**, or fix the *environment* by establishing an optimal average species size. For reasons to be described, GALSINE does the latter. In order for speciation to be effective, we need the species to be large enough to have some diversity of individuals within the species while at the same time having multiple species so that different approaches can develop independently. GALSINE therefore uses a feedback method whereby δ_t is modified dynamically to yield optimal division of the individuals into species. Thus the number of species depends on the population size. The optimal species size is highly debatable, and is a parameter to GALSINE , with a default of 15 to 35. An average species size outside this range triggers a change in δ_t.

We can also attach a total fitness to each species, which allows us to determine how well the species is performing, and hence control how much we allow this species to reproduce. This is important because if, for example, there are 4 species with one performing significantly better, while allowing the other species to continue to exist, we would like to allow the best species to grow larger relative to the others. Hence we define the fitness $\mathcal{F}_s$ of a species s as follows:

$$\mathcal{F}_s = \frac{1}{|s|} \sum_{I \in s} \mathcal{F}_I, \tag{10.2}$$

where $\mathcal{F}_I$ is the fitness of individual I. The fitness of a species is therefore the average fitness of all the individuals in that species. We assign a number of offspring to the individuals of each species proportionally to the fitness of each species. This results in the same numbers of offspring as by the method proposed by Stanley and Miikkulainen [SM01], although this is not obvious from their definition of the fitness.

Magdon-Ismail and Atiya [MIA00] proposed the Early Restart Algorithm, whereby if improvements are not happening fast enough, the learning algorithm is restarted. In the work previously discussed, Stanley and Miikkulainen suggested that if the entire population does not improve in 20 generations, we should allow only two species to reproduce. We can, however, perform similar tracking on a finer scale. The best performance of any individual in each species is tracked. If this best performance does not increase for a large number of generations, the individuals in the species are not allowed to reproduce. In this way innovations which over time do not lead to better results are not protected, but rather eliminated. This may for example occur if during crossover or mutation the best features of all the individuals in the species are accidentally destroyed and these features do not resurface with time.

Speciation is not performed for the first 50 generations in any population. This is so that the very poorly performing initial individuals are not protected in their own species. I explore the effectiveness of speciation in GALSINE in the chapter 13.

10.2 Fitness Function Design

In an evolutionary approach, the design of the fitness function is very important. The fitness function must give a numerical value evaluating the performance of each individual. In GALSINE , the fitness function calculates the fitness based on the error in the output using one of a number of error functions presented below, although the fitness is also modified by the following features:

- The error on training examples is weighted inversely proportional to the size of the class the example is in, so that the evolving individuals do not simply find the predominant class, even if the examples presented are heavily weighted towards that one class.
- If an ANN always gives the same output, irrespective of the input, the ANN is given a very small fitness. This is called a *no effort penalty* and ensures that individuals where the output does not depend on the input (due to an unusual mutations) are eliminated.
- The error on examples correctly classified is multiplied by a constant less that one to balance the importance of a high correctness with that of having a smaller average error, as discussed in Section 4.1.

In order to evaluate the output of the ANN representing an individual, we always take the output neuron which has the highest output value as "on", and all the others as "off".

The functions that can be used for fitness evaluation are the following:

- The mean squared error

  As is usual for ANNs, the target output t is 1 on the output representing the correct class and 0 on the outputs representing all other classes. This allows the error to be defined as:

$$\mathcal{E}_{MS} = \frac{1}{|O|} \sum_{i \in O} (o_i - t_i)^2, \tag{10.3}$$

where O is the set of output neurons from the ANN representing the individual, o_i are the actual values output, and t_i are the target values for the outputs.

- The information theoretic error

  As described in Section 3.4, this error function has been shown to be smoother than the mean squared error function. Although it is not the default option, this error function is available in GALSINE to use in evaluating the fitness of individuals.

$$\mathcal{E}_{IT} = \min(1, \sum_{i \in O} t_i \log \frac{1}{o_i} + (1 - t_i) \log \frac{1}{1 - o_i}), \tag{10.4}$$

  where the minimum is taken so as to bound the error at 1.

- The certainty based error

  When reading the output of ANNs, it may be useful to allow the ANN to answer "I don't know", or equivalently, have a measure of how sure the ANN is of a given answer. Moreover, by basing an error function on certainty, we can remove the need for arbitrary target outputs, when it is only the final classification that is relevant.

  If the label given to a given training example is correct, in other words the output of neuron t representing the correct class is highest, we can define a level of certainty the individual has in this answer as the difference in the value of this output, C, from the next highest output:

$$C_{correct} = o_t - \max_{i \in O, i \neq t} o_i. \tag{10.5}$$

  On the other hand, if the label given is incorrect, we can measure how wrong the individual was using the certainty measure:

$$C_{incorrect} = \max_{i \in O} o_i - o_t. \tag{10.6}$$

  This value indicates how much lower the output of the neuron representing the correct class is relative to the highest output.

  These definitions can be merged, together with a bound to ensure the certainty cannot be too large, to give a definition of certainty as:

$$C = \max(1, |o_t - \max_{i \in O, i \neq t} o_i|). \tag{10.7}$$

Then we can define a certainty based error function as

$$\mathcal{E}_{CB} = \begin{cases} 0.25\,(1-C) & \text{if the label is correct} \\ 0.25 + 0.75\,C & \text{otherwise} \end{cases} \tag{10.8}$$

In this way, if the label is correct, the error is in the range $[0, 0.25]$ and if the label is incorrect the error is in the range $[0.25, 1]$. There is a need for two equations because when the answer is correct a high certainty is good, whereas if the answer is incorrect a low certainty is better.

In addition to these error functions, the other problems the fitness function tries to address are explained below.

10.2.1 Fitness Function Features

Class Dominance

Consider training data where 95% of the examples are in class A and 5% are in class B. Moreover, consider an individual I_A which classifies all examples from class A correctly, and an individual I_B which classifies all examples from class B correctly. If both individuals make the same total number of errors, most approaches would give I_A and I_B a similar fitness. However, I_A would misclassify a larger fraction of the examples from class B than I_B would misclassify from class A, since class A is larger. In effect, misclassifying a given percentage of examples from the dominant class would be less expensive than misclassifying the same percentage of examples from a non-dominant class.

Often the size of the classes in the training examples does not reflect this unbalanced cost of misclassification, and ideally we want the algorithm to misclassify the same percentage of examples from each category. In order to achieve this, the error calculated for each training example is weighted by a value inversely proportional to the size of the class to which the training example belongs.

Activity Requirement

Complex individuals tend to rely on a large number of links in order to perform well. A bad choice of weight on a single link, or the removal of one critical link, can sometimes destroy the classification abilities of an individual. This is a particular concern at the start of a training run when the individuals are relatively simple. It is much simpler to break an individual than the fix the individual after having broken it.

Due to this effect, the fitness algorithm in GALSINE implements an activity requirement, or "no effort penalty". If, for all training examples, an individual always returns the same class, it can be considered broken – the output does not depend on the input across the domain of training examples. Broken individuals are given a very small random fitness. The small fitness ensures that broken individuals are not allowed to survive to the next generation if it is feasible. However, on occasion (especially at the start of runs) most of the individuals in a generation are broken, meaning that if they were all eliminated, the diversity of the population would suffer dramatically. The random fitness is used so as to ensure there is an ordering to the individuals by fitness, so that a cutoff fitness can be selected as to which individuals should survive and which should not. As a result, a random selection of the broken individuals is used (due to the random small fitnesses) in creating the next generation by still requiring the best 50% of individuals to survive. In the subsequent generation there are likely to be fewer broken individuals as they made up a smaller percentage of the parents than in the original population.

Correctness Bonus

If we consider two individuals with the same absolute error, but with one classifying more examples correctly, that individual should have a higher fitness. For this reason, the error on training examples which are correctly classified is multiplied by a constant $(1 - CB)$, with $0 \leq CB < 1$, which effectively gives a "correctness bonus" by reducing the error for correctly classified examples. The effect changing the value of CB, including the effect of its removal by setting CB to 0, is discussed in Section 14.3.

10.2.2 Best Candidate Selection

The most important feature looked for in ANNs trained for classification, after fitness, is generalisability. For an ANN to be useful, it must be able to classify examples it has not seen before with a similar ability as the trained examples.

For this reason, when picking the final best candidate, we do not simply select the candidate with the highest fitness. Each individual I has its fitness $\mathcal{F}_I(Data)$ evaluated on the training data $\mathcal{T}$, $\mathcal{F}_I(\mathcal{T})$ as well as on a set of validation data $\mathcal{V}$, $\mathcal{F}_I(\mathcal{V})$. The validation data is not otherwise used during the training or fitness evaluation. The best candidate is select to have the highest weighted fitness $\mathcal{F}_I'$:

$$\mathcal{F}_I' = \frac{1}{3}\left[\max(\mathcal{F}_I(\mathcal{T}), \mathcal{F}_I(\mathcal{V})) + 2 \times \min(\mathcal{F}_I(\mathcal{T}), \mathcal{F}_I(\mathcal{V}))\right]. \tag{10.9}$$

In this way we favour an individual which has a slightly lower but more consistent fitness over an individual which has a high fitness on the training data but a low fitness on the validation data. The fitness on the training data must be considered in this fitness function because the validation data is usually much smaller than the training data.

Chapter 11

Genetic Operations

This chapter presents the specific genetic operations that GALSINE uses in evolving individuals. These genetic operations are based on methods postulated by Stanley and Miikkulainen [SM01] as part of the NEAT algorithm. I also suggest a number of improvements upon them.

11.1 Mutation

There are two fundamental types of ANN mutations. The first type involves modification of the weights in the ANN. The second involves changes to the structure of the ANN through the insertion or removal of links and neurons.

11.1.1 Weight Modifications

Weight modification is a very simple random mutation of the weights. At every generation, each weight in each individual has a probability $P(\textit{weight change})$ of being modified. If a weight is modified, there is a probability $P(\textit{large change})$ that the weight change is large and $1 - P(\textit{large change})$ that the weight change is small. If the weight change is small, a small random value is added to the weight. If the weight change is large, the weight is set to a new random value from a larger range.

In almost all of my experiments $P(\textit{weight change}) = 0.8$ and $P(\textit{large change}) = 0.03$. This means that small changes occur in most weights at every generation. Such a weight modification scheme works well because genetic algorithms thrive on the randomness and quickly select useful changes. Other values for the weight change probabilities are discussed in Chapter 14. In all my experiments, large changes are in the range $[-5.0, 5.0]$, and small changes are in the range $[-0.1, 0.1]$.

11.1.2 Structural Modifications

Structural modification involves one of three possible operations: (1) adding a link between two existing neurons as shown in Figure 11.1 (a), (2) replacing an existing link with a link to a new neuron that links to the original destination, as shown in Figure 11.1 (b), and (3) disabling (i.e. removing) an existing link. The first two of these are common to almost every genetic algorithm for ANN construction, and the third is also occasionally seen.

Adding Links

At each generation, each individual has a probability $P(c)$ of a new link being added. If a new link is to be added, the genotype is checked for all neurons that already link to any other neuron. These are the allowed sources for the new link (since all neurons which are not in the output layer have at least one output link). Then the genotype is checked for any neurons the selected source neuron does not already link to. A random one of these is picked as the destination of the new link. If the selected source has no allowed destinations (i.e. it already links to every other neuron) then a new source is randomly selected. If there are no allowed sources, GALSINE gives up in its attempt to add a link. The new link has a random weight in the small weight range, unless a disabled link with the same source and destination neurons is already present in the genotype. In such a case, the disabled link is re-enabled, and the original weight is kept. The small weight range is used for each new link so that it does not overly disrupt how the ANN was working prior to addition of the link.

Note that due to every neuron being allowed as a destination for new links, this growth technique can create loops in the ANN, making it recursive. Whether or not loops should be allowed depends on the type of problem being solved. If the output from one input should depend on the output to any previous inputs, loops must be allowed for feedback to occur. On the other hand, if the training examples are independent, then we want the ANN to output the same answer regardless of the previous inputs. In this case loops are likely to reduce performance.

In some cases, recurrent ANNs can give the right output solely because of the order of the training examples. Ignoring this effect can result in learners that appear to have learnt a concept, but which in fact have not.

To avoid this happening, the data can be randomly reordered every time the fitness of each individual is evaluated. This is the approach I first used, which relies on recurrent ANNs not performing well *consistently* for them to be removed from the population. Nevertheless, it is very easy to make a loop through the addition of a single link, and recurrent ANNs may not be eliminated quickly. Hence, this approach allows the population to end up with many under-performing recurrent ANNs, which results in relatively poor learning on problems which do not require recursion.

An alternate approach is to avoid the effect of loops by not evaluating recursive links.

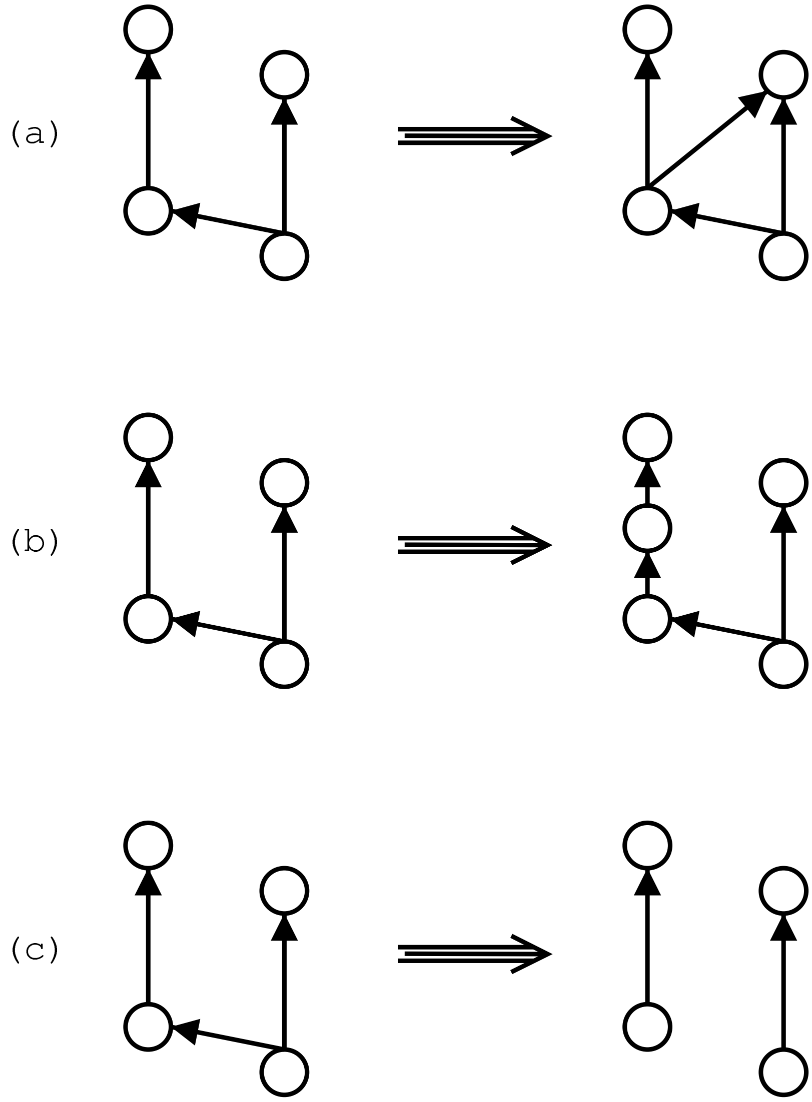

Figure 11.1: Modifying structure by (a) adding a link, (b) adding a neuron or (c) removing a link.

This, as my second approach, improved the consistency in the performance of individuals, making their selection by the fitness function more reliable. However, this meant that the individuals grew more complex. There was no way for the fitness function to differentiate between two similar individuals where only one was recurrent. The individuals with loops often survived, making individuals in the next generation even more complex. This led to extremely complex ANNs which did not perform any better than other much simpler ANNs.

The final approach used to solve this problem worked on the premise that, if recursive links have no effect on individuals' performance, they should not be allowed to form in the first place. GALSINE therefore checks each individual for loops after every mutation or crossover operation. If a loop is present, one of the links in the loop is removed.

Finally, each link that is created has an innovation number assigned to it. The innovation number is incremented by one with every new, different, link. If in the same generation two individuals create the same link through a random mutation, both new links will receive the same innovation number.

Adding Neurons

Adding a neuron is the most important way that ANNs grow. This is because adding links is restricted by the availability of possible neurons to link. In fact, there should be many more potential pairs of neurons to link than actual links present, if we are to observe the brain model discussed in Chapter 4.

At each generation, there is a probability $P(n)$ that a particular individual will be selected to have a new neuron added. In order to add a new neuron, a link is randomly selected and disabled. Then, a new neuron is created and two links are added – one from the source of the disabled link to the new neuron, and another from the new neuron to the destination of the disabled link. In this way, we can think of the new neuron being inserted in the middle of the old link. The weight of the first link is set to unity and the weight of the second link is set to that of the disabled link. This is so that the disruption to the ANN through the addition of the new neuron is minimised. Unavoidably, the new neuron will cause a disruption of the old link. The new neuron is unlikely to be immediately beneficial to the ANN because it has only a single link to and from it. There is an option in GALSINE to additionally provide all new neurons with a link from the bias neuron (which always has an output of 1).

Removing Links

It is beneficial to allow ANNs to become smaller and simpler because of the observed improved generalisability of smaller ANNs. Often, this can even increase the fitness on the training data. Therefore, GALSINE randomly disables links, with a probability $P(d)$ that any given individual will have a link removed at each iteration. If an individual is to have a link removed, a random enabled gene is selected and marked as disabled. Occasionally, removing a link will result in a neuron being effectively disconnected from an ANN, either due to it having no inputs or no outputs. When this occurs, all the other links to the neuron are also disabled so as to produce an ANN which performs identically but is simpler.

11.1.3 Genetic Complexity

Starting with the simplest possible individuals, randomly adding and removing links and neurons, and selecting the best performing individuals, it is clear that the complexity of the individuals must grow, at least initially. It must tend to grow because in general more complex individuals are able to represent more complex hypotheses, and so perform better classification.

If we have probability $P(c)$ of adding one link, probability $P(n)$ of adding a neuron and hence adding two new links, and probability $P(d)$ or removing a link, we can expect the number of links to be added in each component of each individual at each generation to be approximately given by:

$$\Delta\theta = P(c) + 2\,P(n) - P(d). \tag{11.1}$$

Firstly, such a formula ignores the addition of links which results in the removal of a link due to looping in the ANN, and the removal of additional links when the removal of one link results in the removal of a neuron with multiple links. We also have to remember crossover between individuals, but this can be seen as an averaging effect, whereby if some more complex mutations such as those presented above happen, they may be reversed due to crossover with a move complex individual.

The second effect which we are ignoring is an effect which is more important than complex mutations: *evolutionary pressure*. Evolutionary pressure is the pressure on the individuals which results from fitter individuals surviving. If the links being added have no effect on the fitness of individuals, we would expect the number of links per individual to grow as $P(c) + 2\,P(n) - P(d)$. However, the interesting aspect of evolutionary algorithms is that the individuals do not grow randomly, but are guided by selection. If we have an entire population of simple individuals, and it is truly beneficial for the individuals to be more complex, the individuals which randomly get new links are more likely to survive and reproduce to produce the next generation, meaning the next generation will, on average, have more links than expected. Given that we expect more complex individuals to learn more complex hypotheses, the effectiveness of a genetic algorithm can be measured in how fast the complexity grows relative to random complexity growth. This reasoning breaks down when we reach a sufficient complexity to solve the problem at hand, where the ideal evolutionary algorithm would stop complexity growth altogether by noticing the loss in generalisability that results from more complexity in the ANNs.

This leads us to some expectations in the relationship between complexity growth and fitness growth. When the fitness of the population is improving, we would expect to see a strong evolutionary pressure selecting the best mutations and a high complexity growth. During periods when a simple change in individuals cannot improve their performance significantly and the fitness cannot grow significantly until a number of mutations have been discovered, there is no significant evolutionary pressure to guide evolution, meaning we would expect the complexity to increase according to what we expect if mutations were entirely random. Finally, a slowly growing fitness can also be accompanied by a stable or slowly descending complexity if the fitness is very high and any changes to the individuals result in worse performance across the board.

In this sense the growth in complexity of the individuals is an interesting measure usually ignored in genetic algorithms, and shows how well the genetic algorithm works. An ideal evolutionary algorithm will exhibit strong evolutionary pressure irrespective of the problem or fitness, whereas as-fast-as-random complexity growth throughout the learning process indicates a low level of evolutionary pressure. I will explore the complexity growth in GALSINE in Chapter 13.

11.2 Crossover

Crossover is significantly more complex than mutation, and is the harder part to design well in a genetic algorithm. As I mentioned before, the hardest part of crossover is ensuring it is non-destructive with respect to useful features in the individuals being crossed over.

In GALSINE , crossover needs to be performed on a single component at a time, whereby a new individual with a number of components needs to have each component created one by one. All the components in an individual could potentially have different parents, although I have chosen the model from nature where components cannot be separated from the individual, meaning there are only ever two parents of each individual.

The first step in crossover is the selection of parents, and the second step is the merging of the genotypes of these parents to create a new individual. The final, and third step, is to ensure the genotype of the child is functioning and optimal in ways which I will describe below.

11.2.1 Parent Selection

Parent selection starts with determining which individuals and which species are allowed to reproduce. Firstly, species which have not improved for a given number of generations are not allowed to reproduce. In the other species, the median fitness is calculated and all individuals above it are allowed to reproduce: the median ensures half of the individuals survive even if the fitness of individuals is very varied across the species. Using the mean of the fitness as the threshold results in only very few surviving individuals if there are a few very well performing individuals and many that perform much worse.

Having chosen the surviving individuals, each species gets a certain allocation of children for the next generation, as discussed in Section 10.1. Each species has some individuals copied from the previous generation and the rest of the individuals created through crossover. The number of individuals copied is a constant fraction of the total allocation of the species, set to 25% in most of my experiments.

In order to create the new individuals by crossover for each species, GALSINE repeatedly selects an individual from that species. It then selects a second parent with probability $P(\textit{same species})$ that the parent is in the same species. In all my experiments $P(\textit{same species}) = 0.999$. Once the parents have been selected, they are merged.

11.2.2 Parent Merging

The actual crossover procedure is a modified version of that proposed by Stanley. Given two individuals, the matching genes are picked randomly from the parents with the probability that each gene is picked proportional to the fitness of the parent. Excess and disjoint genes are also copied randomly with likelihood of each excess or disjoint gene surviving proportional to the fitness of the parent with the gene relative to the fitness of the other parent. In this way unique structural innovations of both parents can survive or even be combined.

By allowing for the possibility that only some of the excess and disjoint genes from each parent survive, it is possible to have the resulting ANN have neurons which no links to their input or no links from their output. It is even possible to get a neuron with a loop due to the combination of two mutations on links to the one neuron. For this reason, after the crossover operation has been completed, the offspring must have the genotype cleaned, removing genes which do not affect its functioning and removing and loops.

11.2.3 Child Validation

After a child individual has been created, it is automatically checked that it does not contain any loops or stray neurons as described above. An efficient way to check for neurons which are not connected to the rest of each of the components is to evaluate the output on any input, remembering which links have been followed. Any link which has not been followed is irrelevant to the functioning of the individual and can be safely disabled.

Part III

Results

Chapter 12

Learning Results

The GALSINE algorithm was benchmarked on a number of common learning problems. It was tested on the n-parity problem, a binary tree classification dataset, the 2000-2001 Predictive Toxicology Challenge carcinogenicity problem, the Musk problem datasets and on the two spirals problem. Below I present the background behind each of the datasets, and the results of the classification learned. All these datasets can be found at http://csl.anu.edu.au/~filip/honours/index.html.

Three of the problem datasets, the carcinogenicity dataset, the binary tree dataset and the Musk datasets, were provided by Kee Siong Ng at RSISE at the Australian National University. I only made slight modifications to the format of these datasets.

In all the runs discussed in this chapter, the correctness bonus parameter was set to 1, new neurons are linked to the bias node, and all other parameters not mentioned below were left as default. The details about the various parameters available in GALSINE are given in Chapter 14.

12.1 n-Parity

One of the classic problems in ANN learning is the XOR problem, where an ANN must learn to calculate the exclusive-or of two bits, namely 1 if the bits are different, and 0 if they are the same. This is the same as calculating the parity of a set with two bits. We assign a value of 1 to the parity if there is an odd number of 1 bits and a value of 0 otherwise. This problem was originally considered particularly important as it has been shown to require an ANN with a hidden layer to solve.

The parity problem can be considered for an arbitrary number of bits. Given n bits, the ANN must recognise if there are an even or odd number of 1s. Traditional ANNs have a fixed number of inputs, and so n must be fixed. As the number of inputs grows, the ANN must be larger, and requires a longer training time. In fact, the training time increases exponentially in n [TJ88]. Looking at this same problem from the perspective of structured data, the input is a multiset of bit values, in other words, a basic abstraction. Basic abstractions are processed recursively, at any one time taking one bit and the hidden representation of the input already processed as input. If the hidden representation were to become the parity of the processed input, then a simple XOR network would suffice to solve the problem [Duc97].

Although we can see this solution to the XOR problem, it is unclear if the learner will find it. As an experimental dataset, 600 random examples were constructed. They contain between 2 and 12 random bits, resulting in randomly odd and even parities. Of the 600 examples, 302 have even parity and 298 have odd parity.

The GALSINE algorithm learned the classify the unseen validation part of the dataset with a correctness of $95.7\% \pm 11.9\%$ with 10-fold cross validation. The algorithm was run with 800 individuals over 1200 generations. The result did not depend greatly on the mutation parameters, which were set to $P(c) = 0.08$, $P(d) = 0.02$ and $P(n) = 0.01$. Other runs resulted in very similar results.

This result comes from the correctness usually being 100% for each of the ten folds. In the run that gave the results above, eight of the ten folds had a validation correctness of 100%, with one of the remaining folds having a validation correctness of 96.7%. The best individuals selected varied in their complexity, with the simplest solution consisting of an individual where the term reduction network had just two hidden neurons (in addition to three input neurons and two output neurons) and nine links (ignoring links to the bias node). The classifier had just one hidden neuron (in addition to two input neurons and two output neurons) and six links. A number of the other best individuals were of a very similar size. On the other extreme, one of the individual performing with 100% correctness had 38 hidden neurons and 87 links in the term reduction network and about half as many in the classifier. The worst performing fold resulted in a similarly sized best individual.

The results presented above show that, indeed, the structured approach in most cases finds the recursive solution, or a very similar solution, as hoped. However, the solution found is sometimes much more complex than necessary.

12.2 Binary Search Trees

The binary search tree dataset was proposed by Ng [Ng01] to test the term reduction network approach. It consists of 500 examples of binary trees, classified according to whether or not they are binary search trees. A binary tree with root x is a binary search tree if and only if the largest element in the left subtree is smaller than or equal to x, the smallest element in the right subtree is greater than x, and both subtrees are either binary search trees or null. The dataset has 250 examples of both binary search trees and binary non search trees.

The binary search tree classification function can easily be described symbolically. Intuitively, an ANN might be able to perform this classification perfectly if the hidden representation just encodes the minimum and maximum of each subtree whenever the subtree is a binary search tree. However, it is unclear how hard finding the ideal solution may be for a non-symbolic learner such as an ANN. It is worth noting that despite this problem having a possible perfect solution, not all learners will include this solution in the hypothesis space searched. If a learner were to be designed given just the training data and no explanation, there is no guarantee its hypothesis space would include the perfect solution. In the case of GALSINE , it is fairly clear that the perfect solution is in the hypothesis space searched.

Ng reported an average 10-fold cross validation correctness of 87.2% on the validation data set, and a correctness of 87.9% on the training data set. This was achieved using a binary search tree reduction network with 12 hidden neurons. In contrast, GALSINE consistently finds an average validation correctness of about 89.0% $\pm$ 5.0% with an occasional average validation correctness similar to 91.4% $\pm$ 2.4%. The lower result occurs when one or two runs result in an individual that only performs with correctness around 80%, which seems to occur with a likelihood of about 20% at each fold. The higher results can occur for almost all values of the mutation parameters tested ($0 \leq P(c) \leq 0.2$, $0 \leq P(d) \leq 0.1$, $0.01 \leq P(n) \leq 0.1$) when run with 300 individuals over 200 generations.

The individuals found are more complex than those trained by Ng, having between about 10 hidden neurons and 20 links in the binary tree reduction network, and about 40 hidden neurons and 60 links in the classifier, with individuals of all sizes performing similarly in the 10-fold cross validation.

12.3 Predictive Toxicology

The 2000-2001 Predictive Toxicology Challenge (PTC) problem involves classifying molecules as carcinogenic or not [HKKS01]. The original dataset is relatively complex, providing experimental result for 417 molecules tested numerous times, over a number of years, on female and male rats and mice. Carcinogenicity is difficult to measure experimentally with certainty, so the original dataset contains seven levels of carcinogenicity, ranging from negative to positive, including unclear entries such as ‘no evidence’ and ‘some evidence’. Due to the complexity of this dataset, the authors of the challenge used ROC analysis [PF97], which attempts to solve some of the limitations of simply using prediction accuracy in measuring classifier performance. However, for consistency across all my results, and in the interest of simplicity, I will report classifier performance by measuring the correctness.

For this problem to be learnable, the data needs to be more or less consistent. To group the molecules into clear classes, and remove most of the inconsistencies (for example, molecules being noted as carcinogenic in rats but not in mice), only the evidence on female rates was considered. Additionally, molecules that show evidence of carcinogenicity (labelled positive, clear evidence or some evidence) were marked as positive, and molecules that show evidence of a lack of carcinogenicity (labelled negative and no evidence) were marked as negative. Molecules with other labels were discarded. This gave a smaller dataset of 351 examples, 121 of which are labelled as positive. This processed data is the same as that used by Ng [Ng01] and was kindly provided by him for my experiments.

To address this problem, we start with a graph representation as discussed in Section 5.5. The molecules in this problem are represented as follows:

$$
\begin{aligned}
&As, Au, \ldots, Te, Zn \ : \ Element \\
&-, =, \#, + \ : \ Bond \\
\\
&Label = \mathbb{N} \\
&Vertex = Label \times Element \\
&Molecule = \{Vertex\} \times \{Vertex \times Vertex \times Bond\}
\end{aligned}
$$

The $Element$ labels simply represent the chemical elements, and the bond labels represent single, double, triple and resonant bonds in that order.

From this definition for the graph representation of the molecules, we see that all the vertex information is duplicated in the bond information. In particular, both the labels and the elements are part of the description of each bond. This was done by Ng to allow the learner to learn from the relationship between the bond types and vertices that are joined by the bond, working around the problem described in Section 5.5.

Additionally, vertices defining each bond have an artificial ordering imposed on them, such that the label of the first vertex is strictly smaller than the label on the second vertex. This is done to make the representation of each bond unique.

Molecules are reduced to the hidden representation using five basic terms: two terms that generate the hidden representation of vertices and of bonds, two terms that generate the hidden representation of sets of vertices and sets of bonds, and a term that generates the final hidden representation of the entire molecule. GALSINE learned to correctly classify the validation data with a correctness of $67 \pm 6\%$, compared to a validation result of 61.9% for the TRNs considered by Ng.

The achievement of such a high correctness on unseen data, without relying on any additional background knowledge, is remarkable. Other approaches have resulted in a correctness of about 65% (see [HKKS01] for details), although those best performing approaches usually rely on additional carefully selected background knowledge. The standard deviation of 6% on the final result places this result very much on the border of the best results thus far obtained. The parameters used to obtain these results are

$P(c) = 0.1$, $P(d) = 0.033$ and $P(n) = 0.02$, with as few as 125 individuals over 80 generations. The result is not always reproducible, often giving a validation correctness of around $65 \pm 8\%$, although the improved result of $67 \pm 6\%$ is not very uncommon. In any event, these fluctuations are not statistically meaningful, taking into account the standard deviation on each result.

12.4 Musk

The Musk dataset presents molecules that either smell, or do not smell, of musk. The problem was proposed by Dietterich et al. in [DLLP97] as a multiple instance problem, meaning each training example is a set of elements. The example should belong to a particular class if and only if it contains one or more of a group of certain elements. Another way to think of such a problem is for the result to be a logical OR of the classification of each of the elements. When the TRN approach is applied to this problem, the fact that it is a multiple instance problem is lost in the representation, however this problem can still be attempted in the more general sense.

It is known that the most important factor in a molecule smelling in a certain way is its shape. Moreover, molecules have internal bonds around which parts of the molecule can rotate. Each possible set of orientations of the various parts of the molecule is called a conformation. Each conformation can be described by mapping the surface of the molecules after orienting them in some unique manner. The mapping is done by determining the distance from the centre of the molecule to the surface in all directions. By defining 162 directions, the Musk dataset describes each conformation with 162 floating point numbers (distances in Angstroms), as well as 4 floating point numbers to provide the position of a particular oxygen atom in the molecule. Given this original data, Bowers et al. [BGCL00] chose to normalise and then discretise the distances. This allowed the distances to be measured as the number of standard deviations from the mean distance for each molecule.

This complex processing of the structure of molecules means there is a loss of the information that could have been kept in the knowledge representation, but it also shows that GALSINE can learn from feature vectors. The Musk data is also structured with each molecule having a different number of conformations, so this dataset demonstrates that GALSINE can be directly applied to a wide range of problems. A succinct description of the data is:

$$-6,\ -5,\ \ldots,\ 5,\ 6\ :\ \mathit{Distance}$$

$$\begin{aligned} &Conformation = Distance \times\ \ldots\ \times Distance \\ &Molecule = \{Conformation\} \end{aligned}$$

where $Conformation$ is a product of 166 *Distance* components. The training data has each molecule labelled as *Positive* or *Negative*. Additionally, the training data is split in two, with the examples with a small number of conformations in dataset Musk-1 (on average six conformations per molecule), and with the molecules in dataset Musk-2 having a much larger number of conformations (on average 57). This data was learnt with 500 individuals over 350 generations. The mutation rates were set to $P(c) = 0.1$, $P(d) = 0.033$ and $P(n) = 0.017$. GALSINE performed with an average correctness of $79.2 \pm 7.9\%$ on the validation data for Musk-1. As Musk-2 is much more computationally expensive, it was run with 250 individuals over 200 generations and resulted in an average correctness of $77.4 \pm 10.9\%$ on the validation data. This compares to an average validation correctness of 79.55% and 85.30% on the two data sets respectively, found by Ng in the previous TRN approach. The performance of GALSINE on Musk-1 is therefore comparable, but the performance on Musk-2 is worse, although this may be a factor of the smaller population size and generation count.

Unfortunately, compared with other approaches, the previous TRN approach performed well by world standards only on Musk-2. Therefore, by performing worse on Musk-2, GALSINE is disappointing. The world record for other approaches learning on these data sets are correctnesses of 92.4% and 89.2% on Musk-1 and Musk-2 respectively (see [RdR00] for details). That said, many other generic learners have difficulty with this problem. The best performing learners on this problem are specifically designed for it. Even the TRN approach used by Ng [Ng01] required a large number of experiments with different numbers of hidden nodes, and different techniques to decide when to stop learning. For a general algorithm such as GALSINE to outperform many other general algorithms is still a respectable result.

12.5 Two Spirals

The two spirals problem is another classical problem from machine learning. It consists of 194 data points lying on two intertwined spirals, with the classification task that of determining on which spiral a given point lies. Each spiral has 98 data points and completes three rotations around the origin, as shown in Figure 12.1. Fahlman and Lebiere [FL90] discuss some of the background behind the problem and demonstrated successful learning in ANNs to solve the problem quite quickly. This problem does not rely on structured data.

As far as GALSINE is concerned, this is somewhat of a black sheep problem, with which the algorithm has a great deal of difficulty. This data can be learnt successfully by a large variety of machine learners, including ANNs trained by both traditional methods [FL90] and evolutionary methods [Yao98]. Despite a number of attempts at tweaking parameters, learning has not produced results that differ significantly from straight guessing. The classification correctness is usually around $61 \pm 2\%$ when run on a population 2000 individuals for 1000 generations.

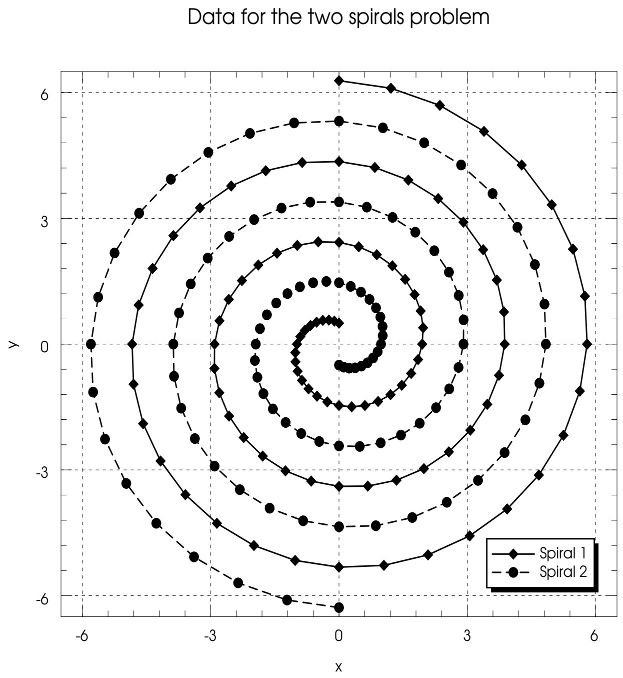

Figure 12.1: The two spirals data consists of the 194 points shown, with the spirals upon which these points lie also indicated.

One of the key features of this dataset is the lack of generalisation requirement. This dataset is given to a machine learner as a whole, so in effect the learner is allowed to overfit the data as much as it likes. Because of this, this is not so much a learning problem as a memorisation problem, although it is possible that the learner be asked to validate its results at some intermediate data points. GALSINE was run on this data with the training, validation and testing datasets identical, so this is not the reason it cannot memorise the dataset. I am unsure where this poor result is coming from.

12.6 Summary

The GALSINE algorithm performs very well on a number of datasets, learning better than hand designed TRNs trained with backpropagation in a number of cases. In particular, learning on the carcinogenicity dataset is comparable to the current world record. However, the learner also performs relatively poorly on the Musk dataset and very poorly on the two spirals problem for unknown reasons.

Chapter 13

Analysis of Learning

In this chapter, I discuss the various observations of the learning as it occurs. Throughout this chapter, when I refer to mutation, I am only referring to structural mutations that result in the addition or removal of nodes and links, and not to mutations that only result in changes to the link weights.

13.1 Evolutionary Pressure

When dealing with genetic algorithms, we may wonder if the evolution is actually working – that is, does the selection of the fittest individuals exert a measurable evolutionary pressure on the population. One measure of evolutionary pressure is to observe the rate at which mutations are adopted in the population, as was discussed in Section 11.1.3. If evolution is selecting better individuals based on their structure, it will at any given point in time prefer individuals which are either more complex, of the same complexity or less complex than the current best individual, depending if added complexity increases the performance of individuals.

Figure 13.1 shows the change in the complexity of the best individual for a particular run of learning on the binary trees dataset. Complexity is measured as the number of links in the term reduction networks making up the best individual. If there was no evolutionary pressure, and the best individual was just a random individual, we would expect the number of links in that individual to be given by:

$$Links = 18 + 2 \times (P(c) + 2P(n) - P(d)) \times Generation. \tag{13.1}$$

The slope of the *Links* function given in Equation 13.1 is the net expected rate of new links being added, taking note that each new node results in two new links. The slope is universal across all problems, apart from a constant factor. The constant factor in this case is two due to the that individuals which solve the binary tree problem being made up of two term reduction networks – one to create the hidden representation of binary trees, and a second as the labeller. The constant value of 18 in Equation 13.1 is the number of links in the initial population for this dataset. In this particular run, $P(c) = 0.05$, $P(n) = 0.02$, and $P(d) = 0.05$, giving an expected growth rate of 0.08 links per generation. This slope is indicated by the vector drawn in the upper left corner of Figure 13.1.

In the diagrams, we see that the rate of growth is not uniform, and moreover it is correlated to the correctness of the best individual on the training data, shown in Figure 13.2.

The first thing we see on Figure 13.2 is that the correctness does not increase monotonically, as we might expect. Most machine learning algorithms see a monotonic improvement in the learner on the training data. The reason the correctness jumps about so much is that the best individual in each generation is not necessarily related to the best individual in the previous generation. Unlike traditional learners, where we see incremental improvement between iterations, here each subsequent point in Figures 13.1 and 13.2 is a completely new individual, which is usually better than the previous best individual, but does not have to be.

Initially, in the first 50 generations, we see some growth in the complexity, and an increase in the correctness of the best individual. Over these first few generations, we also see the correctness become more consistent between generations – the population is becoming somewhat more uniform. Over approximately the next 90 generations, we then see the correctness improve significantly, but the complexity stays roughly fixed: the evolutionary algorithm is selecting the best individuals, and these are the individuals without additional complexity. This is the effect of evolutionary pressure. At about generation 140, we see a sudden drop off in the correctness of the best individual and a simultaneous increase in the complexity. What has happened is due to the best performance of the individuals in the best species not improving over many generations, as discussed in Section 10.1. At about generation 115, a particularly good best individual was found. This individual belonged to a species which appears to have been the only species performing very well. After 25 generations, the record fitness of this species had not improved: a better solution had not been found. This resulted in GALSINE removing all individuals in that species from the population. The next best species had individuals which were more complex, and so the complexity of the best individual increased. Over time, this other species also grows to have much improved individuals.

We see that the growth of the individuals in the best species is very much kept at bay by the evolutionary pressure. The mutations that occur are apparently not beneficial, so when the best individuals are selected, the ones which have increased in size tend to be eliminated. However, this is only possible due to the relatively small rate of mutations. Consider Figures 13.4 and 13.3. In this run, the mutation rates are significantly higher, $P(c) = 0.15$, $P(n) = 0.1$, and $P(d) = 0.02$, giving an expected growth rate of 0.66 links per generation. We can see slight inflections in the complexity of the

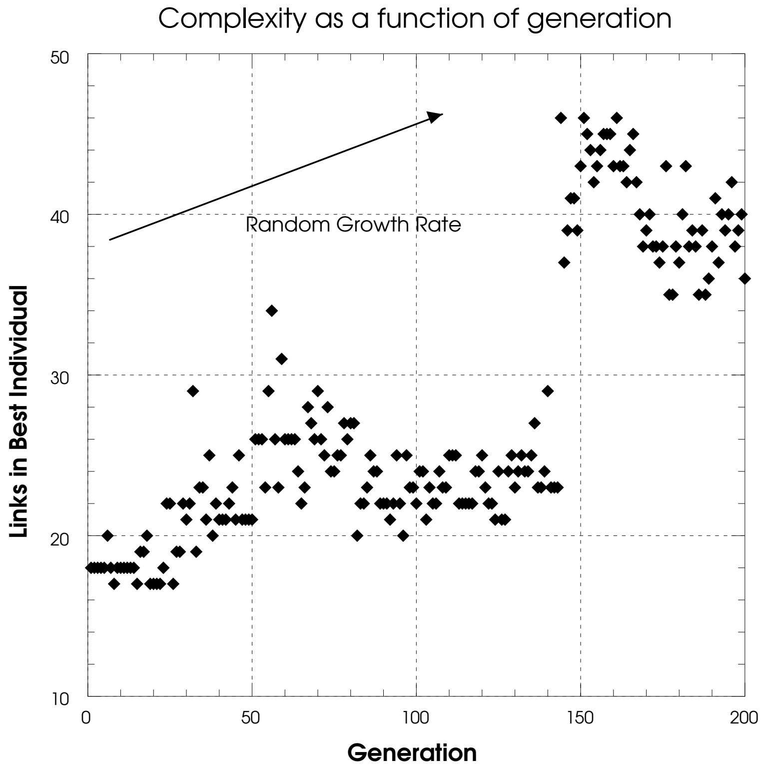

Figure 13.1: The complexity of the best individual in this binary trees run does not change at the rate expected from random mutations, shown by the vector in the upper left, but rather is directed by evolutionary pressure.

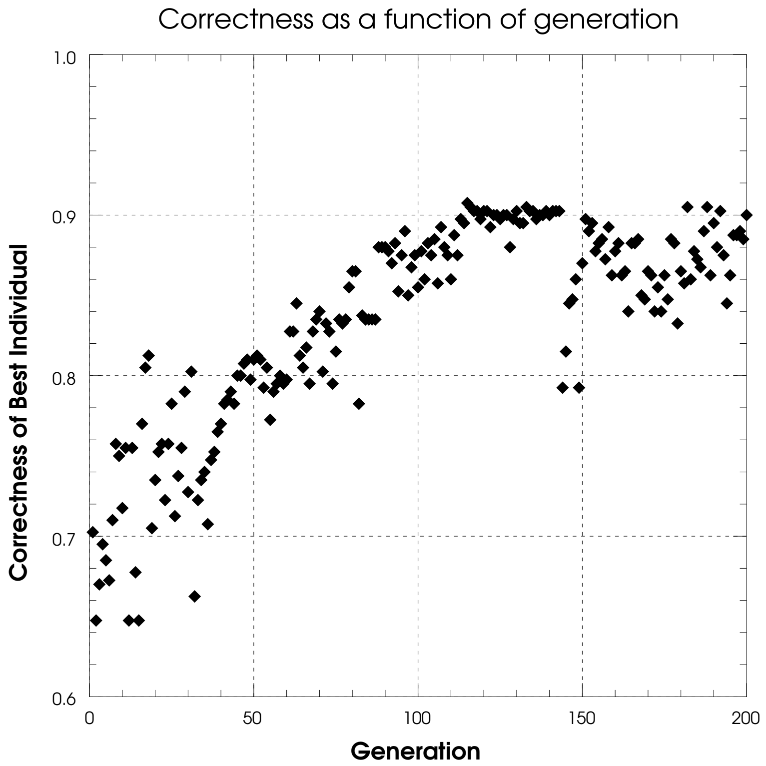

Figure 13.2: The correctness of the best individual guides the rate at which the complexity of the best individual changes. We can see the elimination of the best species at about generation 140 results in the complexity surge seen in Figure 13.1.

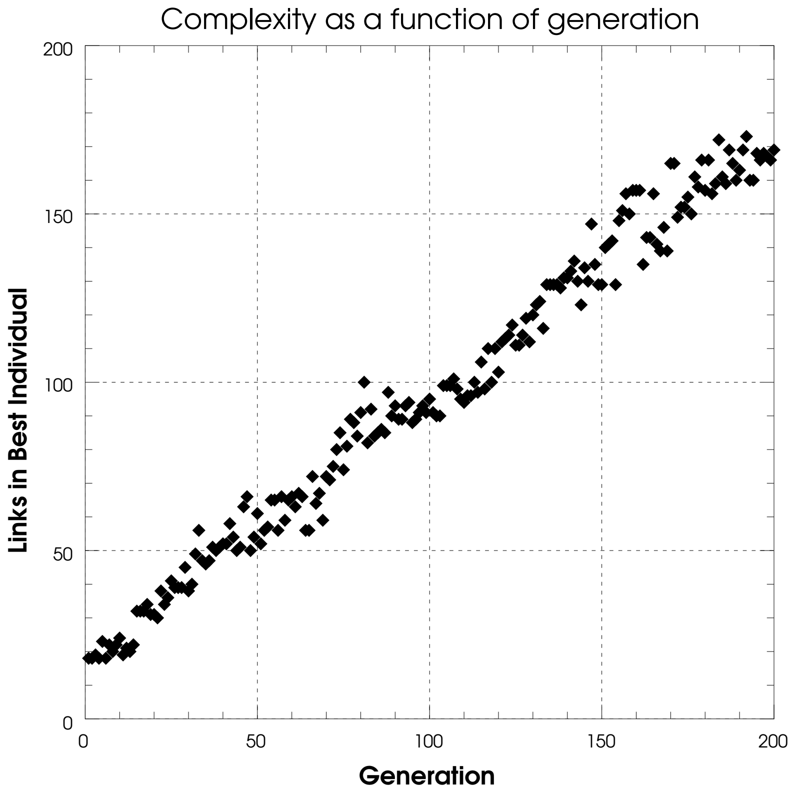

Figure 13.3: The complexity of the best individual in a population trying to solve the binary trees problem with high mutation rates. In this case, the evolutionary pressure cannot control the rate of complexity growth.

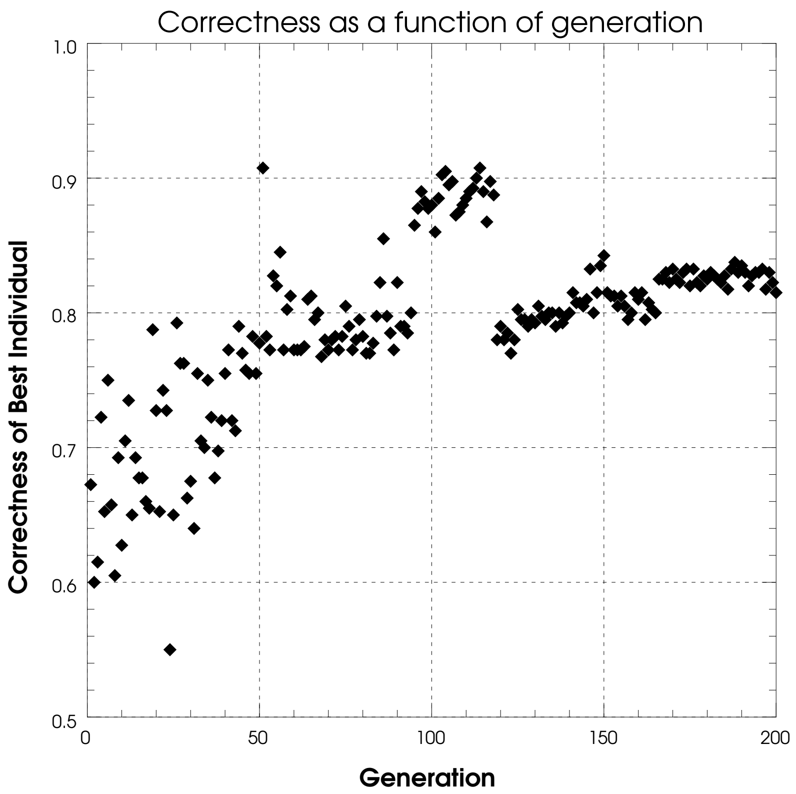

Figure 13.4: The complexity of the best individual in a population trying to solve the binary trees problem with high mutations rates.

best individual, for example around the 70th and 100th generations, but the complexity is always growing. We can also see in this experiment that when the best species was removed, other individuals did not improve as quickly to take the place of the best individual. Note that when there is no improvement within a population, the individuals cannot be differentiated, and so evolution cannot see mutations as either beneficial or not. This means that when the correctness curve is flat, we expect the rate of complexity growth to be very close to what we would expect if random individuals were chosen to survive.

To summarise, evolutionary pressure can restrict how quickly the best individual grows, so long as the random mutation rate is not too high. This stands to reason – if almost all the individuals available for crossover have had a structural mutation performed on them, then there may simply be no good individuals which have not been mutated.

In observing the changes in complexity of the best individual, we may be able to use this information to automatically tune the mutation rates. If the complexity of the best individual is not increasing, or is even decreasing, this means the better individuals are simpler, and so the mutation rates could be adjusted appropriately towards slower growth. On the other hand, if an increase in complexity is beneficial, and the complexity of the best individual is increasing rapidly, the mutation rates should be changed such that the net rate of growth is increased. It is unclear what should happen when there is no improvement in the population and the complexity is increasing at the random rate.

13.2 Speciation

In Section 10.1, I discussed speciation and how we establish a target average species size. For every generation (after the fiftieth) the average species size is computed. If the species are too large, the speciation threshold, δ_t, is reduced by a random value of between 0.001 and 0.003. Conversely, if the species are too small, the threshold is increased by a value in that range. This range of values was chosen so that the change is relatively slow but not too slow. The value of the change is randomised so as to avoid oscillation between two speciation threshold values when the larger value is too large and the smaller value is too small for a given population. In addition to this dynamic speciation threshold tuning, I remind the reader that species which do not improve for a given number of generations, i.e., they do not result in an individual better than any previously seen in that species, are eliminated from the population.

We have already seen some of the effects of speciation, but they are much more visible in other runs. Taking a look at Figures 13.5 and 13.6, we see a run again on the binary trees dataset. In this case, we see that at about generation 170 the best species is eliminated. This allows another species to grow more. The other species is initially of a similar complexity to the eliminated species so there is no significant change in the complexity of the best individual. When this new species starts to improve, it quickly outperforms the previously eliminated species. Then, after this newly dominant species has reached its limit and stops improving, it too is eliminated. At this time, the next best species in the population does consist of significantly smaller and worse performing individuals, so both the complexity and correctness of the best individual drop. This process can be repeated numerous times for runs with many generations. In the next chapter I discuss my experiments to determine the best number of generations to wait before a species is removed.

Speciation is performed by setting a target for the number of species. As the generations progress, the complexity of individuals increases, and the diversity in the population relies more on smaller fractional differences between individuals, remembering that the similarity measure given by Equation 10.1 is normalised for individual size. This means the speciation threshold must decrease in order to retain a similar number of species (so that smaller fractional differences result in individuals belonging to different species). The adaptation of the speciation parameter is shown in Figure 13.7 for a run on the predictive toxicology challenge dataset. Figure 13.8 shows how this threshold corresponds to the number of species present.

The target range for the number of species comes from a target species size of between 15 and 35 individuals, with a population of 250 individuals. We can see that normally the number of species decreases, with a sharp increase when the speciation threshold is reduced, although there are also natural fluctuations in the number of species.

Related to speciation, Figure 13.9 shows the complexity of the best individual for a run on the Musk-1 dataset. Usually, we have seen the complexity of the best individual to be relatively stable, most of the time changing little between subsequent generations. The result here is interesting in that the complexity of the best individual diverges, rather than being stable. This may be due to a number of similarly performing species with individuals of different sizes. In this case, if the mutation parameters were to be set automatically, we would want the algorithm to reduce the expected complexity growth because there is no need for complexity to increase fast if there are simpler individuals performing as well as the more complex individuals.

13.3 Network Properties

Chapter 4 mentions the non-uniform connectivity of the brain. The mutation scheme in GALSINE does actively not promote such clustering of neurons, although it certainly allows it. Should such clustering prove useful in achieving a higher fitness, it can occur. The problem sizes studied in this thesis do not result in large enough ANNs for such an effect to be measurable, but we can observe how the relationship between the number of nodes and the number of links changes. Figure 13.10 shows the num-

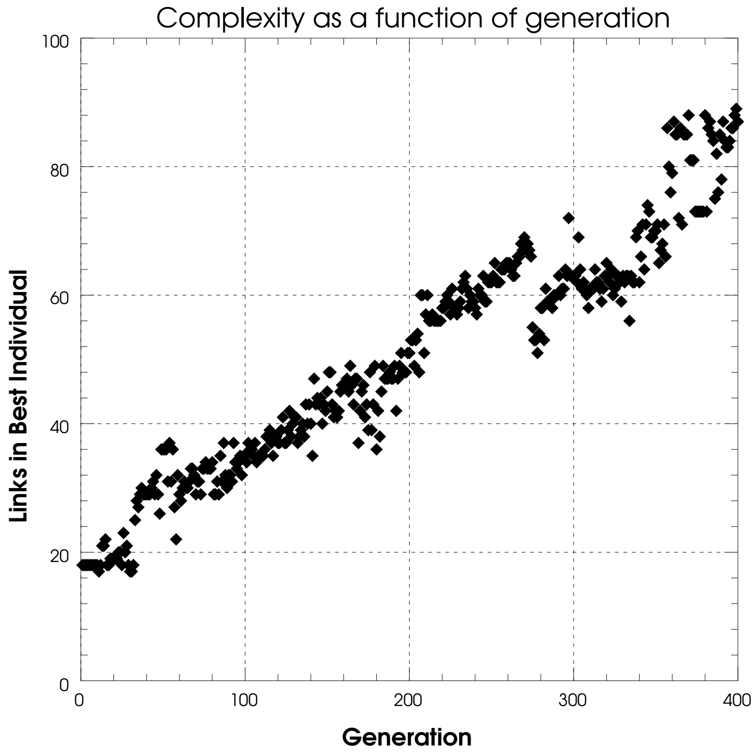

Figure 13.5: The complexity of the best individual in a population trying to solve the binary trees problem. Here we see that when the best species is eliminated at about generation 170, the complexity does not change very much.

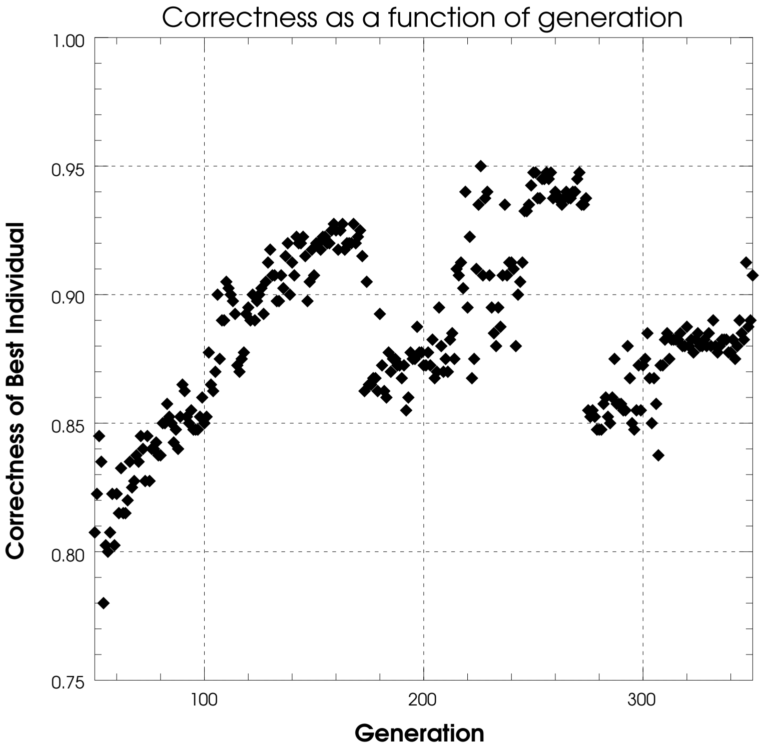

Figure 13.6: The correctness of the best individual in a population trying to solve the binary trees problem. When the best species fails to improve for some time, at generation 170 it is eliminated and replaced by an individual from a different species.

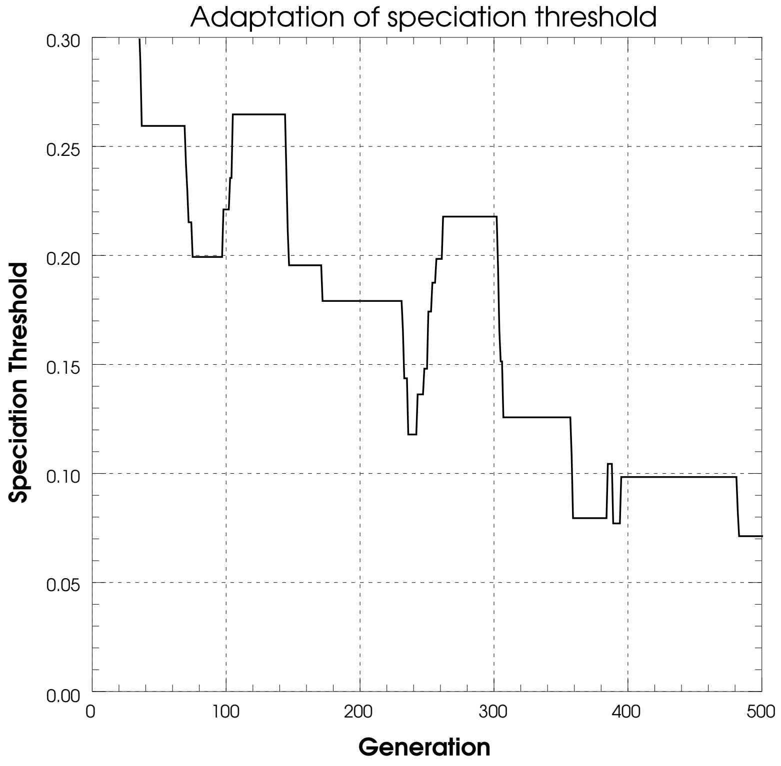

Figure 13.7: Overall, the speciation threshold drops over time, as individuals become more complex and have a smaller fractional difference.

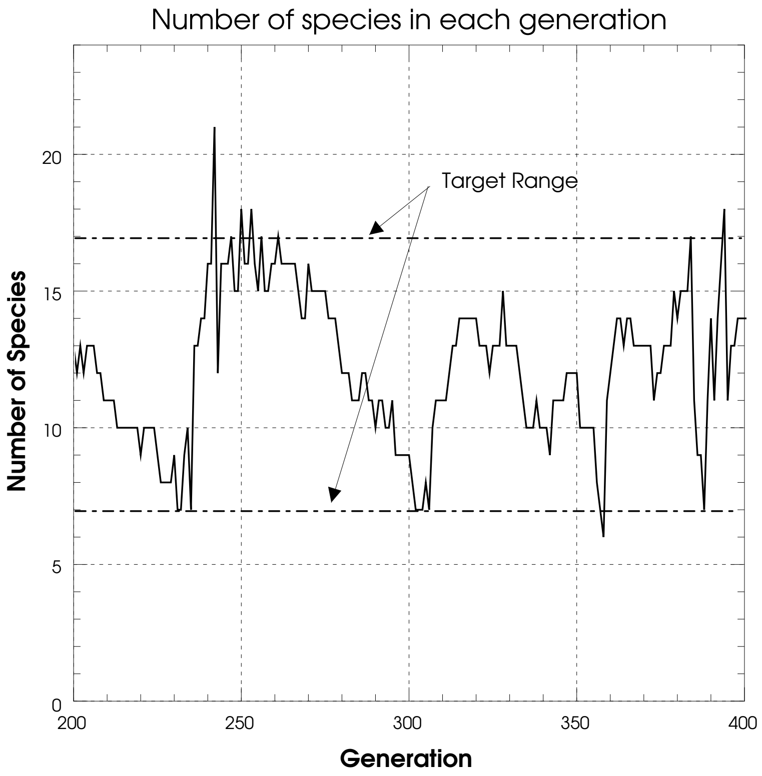

Figure 13.8: The number of species is correlated to the speciation threshold, shown in Figure 13.7. We see a slow decline in the number of species followed by a rapid rise when the threshold is decreased.

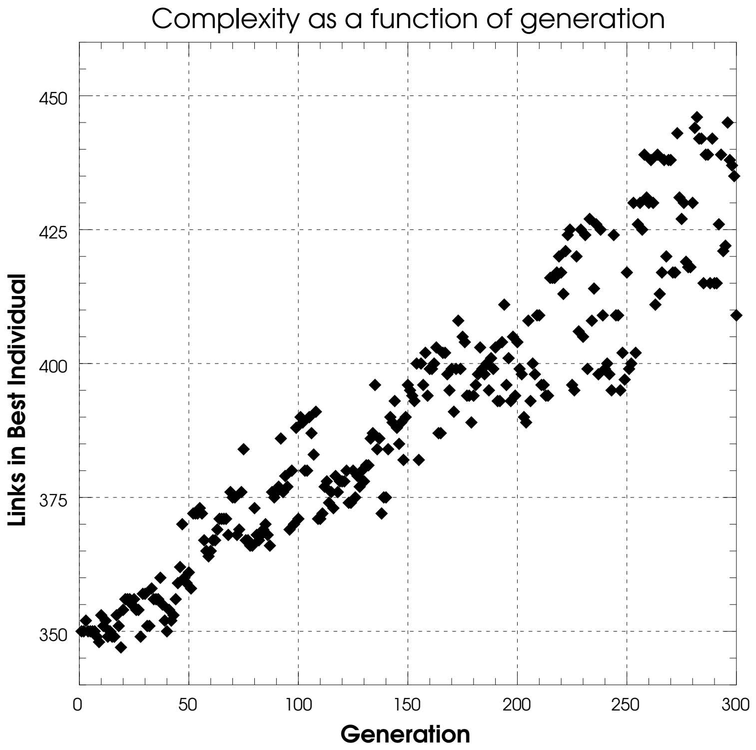

Figure 13.9: The complexity of the best individual varies greatly between generations in this Musk-1 run. This is probably due to a number of similarly performing yet differently sized species competing.

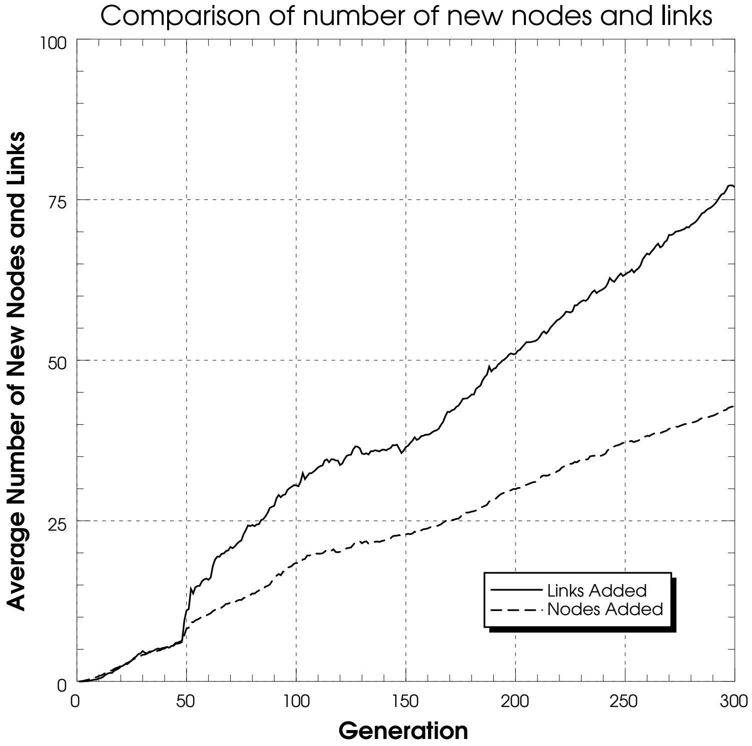

Figure 13.10: This graph shows the average number of new links and nodes added to individuals during a Musk-1 run.

ber of nodes added and the number of links added, on average, to individuals in the entire population during a Musk-1 run. We see that overall the increase in both is linear. In other words, a constant number of links are added for each new node. This is what we expect from random growth, so it is unclear if evolution is playing any part in this. However, growth of this sort will guarantee the fraction of other nodes that each node links to becomes smaller as the ANNs become larger.

It is also interesting to note how during the first fifty generations the number of links added is very close to the number of nodes added. This is because each new node results in one new link being added (in this run, new nodes are not automatically linked to the bias node). Initially, the new nodes are surviving, but new links are being eliminated, or in some cases link addition is not even possible. Notice that while node addition operations will always succeed, adding a link relies on there being a pair of nodes which do not already link to each other. The sudden surge in the number of links at generation 50 coincides with speciation being started, which allows a diversification of the population.

Chapter 14

Learning Parameters

As can be seen in reading this thesis, the GALSINE algorithm relies on a large number of parameters, much as natural evolution relies on an infinite number of parameters. In nature, the parameters range from radiation levels to weather patterns. In this chapter I explore the important parameters of GALSINE , and discuss how the selection of parameter values can be largely automated.

14.1 Population Parameters

The two most fundamental parameters of any evolutionary algorithm are the population size, *PopSize*, and the number of generations, *Gens*. The amount of computation that must be done by the algorithm is proportional to the product of these two parameters. Hence there is a natural tradeoff between *PopSize* and *Gens*. The question often asked is:

> I would like to solve a problem as well as possible within a given amount of time. This allows for a certain number of evaluations of individuals. What should the population size be relative to the number of generations?

The short answer is that, understandably, it depends on the problem. This must be the case because the number of generations determines the maximum possible complexity of individuals at the end of the learning process. How this compares to the complexity necessary for a good solution depends on the problem. For example, when attempting to solve the n-parity problem, the solutions we expect are relatively simple: we are looking to find networks that perform an XOR operation. This means a larger population size, exploring more possible weights for simple ANNs, is likely to give better results, meaning the ratio *PopSize* : *Gens* should be larger. Conversely, when solving a complex problem such as the carcinogenicity problem, we expect the solutions to be complex and so should select a smaller ratio for *PopSize* : *Gens*, so as to allow the complex individuals time to evolve.

However, we can only decide on this ratio having some idea of the expected solution to the problem. Given some new data we wish to learn from, a population size must be chosen. As it currently stands, GALSINE does not help the user pick the population size. There is a default value of 250 individuals, although this is an arbitrary choice. The default number of generations is 200, also an arbitrary choice. In general, best results tend to be obtained when the number of individuals is comparable to the number of generations.

With respect to the total work done, $PopSize \times Gens$, it is interesting to note that on the predictive toxicology data, the final result did not change significantly across a wide range of values. $PopSize \times Gens$ was tested with many values between $10,000$ and $800,000$, across a range of ratios for $PopSize : Gens$. The result showed that, so long as *Gens* was above 100, the difference in final performance was smaller than the fluctuations due to the standard deviation of the validation correctness.

It is also important to notice that more work done does not always guarantee an equal or better solution will be found. By doing more computational work and learning for longer, we can end up with a worse solution – if we look for a solution long enough, we may end up overfitting the training data. Even if the best individual appears to be generalisable, it may simply overfit both the training and testing data. Despite learning only using the training data, the best individual is selected by looking at the testing data as well. The algorithm could stumble upon an individual which looks generalisable because it happens to overfit the testing data as well as the training data.

14.2 Mutation Parameters

The next most important set of parameters are those that determine the mutation rates. The three parameters which we consider as variable, discussed in Section 11.1, are:

1. The probability of a new link being added, $P(c)$,
2. The probability of a new node being added, $P(n)$,
3. The probability of a link being removed, $P(d)$.

Experiments were conducted using the binary trees dataset over a range of values for the mutation parameters, with all other parameters constant. The values tested were $P(c) \in \{0,\ 0.01,\ 0.05,\ 0.1,\ 0.15,\ 0.2\}$, $P(n) \in \{0,\ 0.01,\ 0.02,\ 0.05,\ 0.1\}$, and $P(d) \in \{0,\ 0.02,\ 0.05,\ 0.08\}$.

The results showed that so long as $P(n) > 0$, for sufficiently large values of *PopSize* and *Gens*, the choice of the mutation parameters made very little difference. Additionally, poorer performance was observed when $P(d) = 0$. Interestingly, the performance was not degraded when $P(c) = 0$, in which case we get individuals with many nodes, with each node having very few links.

It was also observed that as the expected rate of complexity growth, $P(c) + 2\,P(n) - P(d)$, goes up, the performance increases slightly, and the standard deviation in the error decreases slightly. However, these changes are very slight, and are dwarfed by the standard deviation in the performance. If the mutation rates are very high, with $P(c) = P(n) = P(d) = 0.9$, we do see a decrease in the performance on the binary trees dataset, yielding a validation correctness of about $75 \pm 20\%$ instead of about $90 \pm 5\%$. The performance with respect to the number of links changed per generation, $P(c) + 2\,P(n) + P(d)$ varied very little. A higher total mutation rate, in general, resulted in a slightly smaller standard deviation in the validation correctness, although again this is not really statistically valid.

14.3 Fitness Evaluation Parameters

As discussed in Section 10.2, there are a number of possible error function that can be used in evaluating the fitness. In particular, GALSINE implements the standard mean squared error function, the information theoretic (log) error function, and the certainty based error function discussed previously.

These three functions were tested on the n-parity and binary trees datasets for various values of the mutation and population parameters. From the tests performed for this thesis, it is visible that the certainty based error function results in the worst learning. Strangely, the difference between the performance of the certainty based error function and the other error functions increases as the mutation rate parameters increase. In any event, this means that the certainty based error function, while allowing for the removal of specific target output values, does not work well. The difference between the other two error functions is relatively small, with the information theoretic error function perhaps performing slightly worse. Hence the best error function to use is the default one, the mean squared error function.

It is interesting that the error functions play such a large role. The main purpose of the error function in a genetic algorithm is to allow the algorithm to determine a threshold fitness below which individuals are eliminated. In this sense, genetic algorithms are really looking for a ranking of the individuals, and not absolute error values or slopes. Obviously, the different error functions must be producing significantly different rankings of individuals in order to produce differing overall results. This analysis of the fitness functions also shows that the improvements reported by Unnikrishnan et al. [UHT91] by using the information theoretic error function, do not carry over to the GALSINE algorithm. The main benefit of the information theoretic error function, the smoother error surface, does not give any benefits when there is no gradient descent being performed.

The correctness bonus, *CB*, is the other major parameter of the fitness function. It determines the importance of high correctness relative to a small error, by multiplying the error on examples classified correctly by a constant less than 1. A higher value of *CB* makes this constant smaller, meaning correctness is favoured more. A value of 0, eliminating the correctness bonus, give the worst results on the two datasets studied in detail, the n-parity and the binary trees datasets. Very high values of the correctness bonus, on the other hand, do not produce any adverse effects. In fact, any value above 0.5 seems to give similar results. The default value in GALSINE , 0.83, is a good choice and can be left unchanged.

14.4 Parameters Influencing the Hidden Representation

When dealing with the hidden representations of individuals, it is important to consider how large the representation should be. This size is the number of output neurons in the term reduction networks that create the hidden representation of each component. Different components could, in principle, have different sizes for the hidden representation, although GALSINE assumes a consistent size across all components.

The default size is 2, and this seems to work quite well for all the datasets tested. A size of 1, such as was used by Ng, certainly produces worse results. However, increasing the size above 2 did not help on the problems chosen, namely the n-parity, binary trees or predictive toxicology challenge datasets. This means that the default value of 2 is a good value to use in general, although I would expect that on more complex problems, a large value may be helpful. It is worth remembering that larger hidden representations create larger neural networks.

Another option discussed earlier is the labeller, that is, the classifier which is attached to the final hidden representation. To my surprise, the labeller has proved less useful than I expected, but the importance of its presence depends greatly on the problem. On all the problems but one, the presence of the labeller tended to decrease the classification performance on the validation data by a very small amount, about 1%, when the number of individuals or generations is small. However, the n-parity problem performed much worse without the labeller – giving a correctness of around 60% in most cases, instead of well over 90%. Therefore, despite it not playing a large role most of the time, it is an essential component to allow learning on arbitrary datasets. By default, the labeller is present. Note that if the labeller is removed, in order for the number of outputs to be correct and allow the results to be classified, the size of the hidden representation has to be identical to the number of possible labels.

14.5 Genetic Operation Parameters

There are a number of parameters that affect the types of genetic operations that occur, and how disruptive they are.

The first parameter determines if new neurons that are inserted are automatically linked to the bias node. Linking neurons to the bias node makes the ANN have many more links, but helps performance significantly. This is understandable, because having a bias on neurons certainly allows them to learn more complex functions. This bias link is not created by default but simply requires a flag on the command line.

One of the fundamental steps in genetic algorithms is where a new population is made from the previous population. As mentioned earlier, this step relies on crossover being non-destructive, and so crossover is sometimes replaced with a straight copy. By using copying to create a given fraction of the new population, and crossover to create the rest, we get better results. Having 100% of the individuals created by crossover makes learning the n-parity dataset worse, but has little effect on the binary trees dataset. Having straight copy for 100% of all the individuals also reduces performance, although to a lesser extent. A value of 50% copy and 50% crossover performs relatively well, but the best results were achieved with a value of 75% of individuals created by crossover, which is the default for GALSINE .

By far the most common mutation to occur is that of changing the weights. Each weight has, by default, an 80% chance of being modified at each generation. If a given weight is modified, it has, by default, a 3% chance of being initialised to a completely new random value. If instead of 3%, the chance of a large change to the weight is higher, both n-parity and binary trees learning suffers, each having the correctness on the validation data drop by about 10% when the chance of a large change is 100%. Having a more moderate chance of 10% also performs worse, particularly for the binary trees dataset. On the other hand, a chance of 0% of a large weight change also decreases performance. Hence the value of 3% is a good compromise.

Modifying the chance of a weight change from 80% also affects the learning. If the chance is small, learning is less effective, but I suspect this could be made up for by having more generations. Even a value of 50% produces a measurable decrease in performance. Hence the value of 80% is good.

14.6 Speciation Parameters

The last class of parameters I studied are those relating to speciation. Observing the behaviour of the automated speciation threshold, we see the number of species oscillating over time. As discussed in the previous chapter, this is due to the complexity of individuals slowly forcing down the speciation threshold as the fractional difference between individuals becomes smaller over time. Without automated control of the speciation threshold, we would get fewer and fewer species as the generation number increases.

The optimal species size is a parameter to GALSINE , by default set to 25, with a buffer of 10. This means that, as long as there are between 15 and 35 individuals per species, the speciation threshold is not adjusted. Changes to this parameter were tested, and showed that increasing the optimal species size to 60 individuals resulted in worse learning results. Reducing it to 10 did not hurt performance, but was also not beneficial. This is consistent with speciation allowing crossover to work better by limiting hard cases where very different individuals are crossed over. As such, the default value appears to work well.

Species patience is the other speciation parameter. It determines how long each species is given to produce a new best individual. If a species does not create a new individual which performs better than any previous individuals in that species within a given number of generations, the entire species is not allowed to reproduce. The default species patience is 25 generations. Increasing this value modestly, for example to 60 generations, does not appear to change the results meaningfully. Increasing it further also does not increase or decrease performance directly, but appears to slightly increase the standard deviation of the validation correctness (although this may be a statistical fluctuation). If true, such a result would confirm that if we have a population made up of a number of species, and some of the species do not improve over time, it is beneficial to remove those non-improving species. This allows other species to expand, effectively meaning GALSINE can try again to find a good solution.

Chapter 15

Automation

Having set out with a goal to allow learning on structured data with minimal human intervention, this chapter presents how far this work has progressed towards that goal.

15.1 Generalisation

The first question to answer is how generally applicable the GALSINE algorithm is. Data presented to GALSINE must be representable using basic terms. Assuming there is such a representation for some training examples, that data can be learnt from. The genetic operations in GALSINE can be applied to any problem, not relying on any particular characteristics of the dataset. Fitness evaluation is done through a fitness function which aims to avoid overfitting by automatically splitting data into training and testing sets, and using the testing data to check generalisability, again meaning it should work on any data. The generalisability of solutions is confirmed by testing the best individual from each run on another separate set of data – the validation dataset.

Each experiment with GALSINE currently runs a 10-fold cross validation, i.e., it runs the entire learning process ten times. This results in ten best individuals, one from each run. These individuals are saved and can then be used to classify any other data in the same format. Hence, to use GALSINE , the user would pick one of the best individuals, or alternatively use a number of them with some sort of "voting" system, to classify additional data.

Finally, GALSINE allows the user of the algorithm to annotate the training data with additional features. This allows the user to attempt to make learning easier for the algorithm by providing background knowledge. The user does not need to perform the more tedious task of feature extraction.

Thus, I claim that GALSINE is generalisable to any structured or flat dataset.

However, the results have shown that there are datasets where GALSINE performs poorly. The reason for the poor performance of these datasets remains to be determined. Moreover, the way structured data is automatically analysed does not presently allow the user to specify a richer definition of the structure, for example specifying that two enumerated elements at two positions in the one tuple are in fact of the same type.

15.2 Tuning

GALSINE does have a very large number of parameters, however I have shown in this thesis that most of these parameters can be left at the recommended values, with most of these values being default. The only parameters the user really needs to be concerned with are:

- The population size.
- The number of generations.
- The three mutation rates.

Similar values in these parameters have been shown GALSINE to produce similar results, meaning that the particular values are not critical. As such, provided a generous population size is set, with a sufficiently large number of generations, the user should be able to get a reliable result even with just a single run of the learner. If GALSINE proves to learn very well on the user's dataset, the algorithm can then be run a few more times with slightly different values of some of the parameters in order to observe any improvements in learning.

Some human involvement is essential in every learning problem. In this approach, the involvement is almost entirely limited to the knowledge representation phase. This is the correct phase for the human involvement to take place because this is the phase when the examples collected in the real world are presented in a computer readable form. By allowing for structured data, GALSINE makes the knowledge representation phase as easy for the user as possible, so that the user does not need to explicitly derive a fixed number of features from the training examples. Depending on the quality of the knowledge representation, and the detail that is recorded, the algorithm may be more or less successful in the learning process, but this is the case with any form of supervised learning. How well a learner can learn is always influenced by how well the teacher teaches.

Thus, I claim that GALSINE can learn with minimal human intervention.

Chapter 16

Conclusion

This thesis has presented a machine learning algorithm which I have demonstrated to be on par with many much more involved learning algorithms. These algorithms are more involved because, for them to achieve good results, they require a person as a partner to the machine learner throughout the learning process. By observing the processes at work in nature, we can build much more effective machine learners which perform well in their own right. This allows the human involvement to be limited to the knowledge representation phase. As such, I consider the first most important outcome of this work a statement that there is much that we can learn from nature.

GALSINE is a very promising learning algorithm. In this thesis, I have also identified possible future improvements – both suggesting methods to allow the learning process to be even more automated, and searching to find the problems on which the algorithm performs worst. This project has been successful, both in generating meaningful and impressive results, and in setting the stage for future work.

We build machine learners to help us draw conclusions from our observations of the world. As such, to allow the learner to best perform its task, we must be able present our observations in as precise a form as possible. For this reason, a general approach to structured data is an essential component of machine learning. I hope I have conveyed the importance of careful knowledge representation in this thesis.

Natural evolution is effective. By understanding the processes which occur, and influencing evolutionary selection, humanity has made immeasurable progress. Starting with the concepts of agriculture, we have selected todays fruit trees over those less useful. This process continues today through to genetic engineering and gene therapy, using the strengths of evolution to improve the quality of our lives. In a similar way, the questions beg to be asked: Could we understand learning and simulate evolution to work for us in just as effective a way? Could we guide the evolutionary learning processes to make them more effective?

In the future, it would be even more interesting to find a way to allow an evolutionary neural network learner to draw from a database of generic background knowledge that was not created specifically for the problem at hand. Given an example to classify, such background knowledge would have to be used in conjunction with artificial neural networks that can perform multiple evaluations before returning an answer. This is necessary to make possible the self-generation of new input, which allows for the background knowledge to be accessed.

Following this work, there is one question which sits most on my mind. Nature has been shown in the physical sciences to allow for very simple explanations of most natural phenomena. Physics can be largely described by a handful of equations, yet evolution seems incredibly complex. Is there an underlying process which can be described succinctly, or are the processes of life so complex simply because they are so hard to achieve?

Bibliography

[AB99] Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

[ABE00] Douglas Aberdeen, Jonathan Baxter, and Robert Edwards. 92 c /mflop/s, ultra-large-scale neural-network training on a PIII cluster. In *Proceedings of Super Computing 2000*, Dallas, TX, 2000.

[AG97] Edoardo Amaldi and Bertrand Guenin. Two constructive methods for designing compact feedforward networks of threshold units. *International Journal of Neural Systems*, 8(5):629–645, 1997.

[AHS85] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[AK96] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transporter network architecture. pages 317–338, 1996.

[Ash89] T. Ash. Dynamic node creation in backpropagation networks. *Connection Science*, 1:365–375, 1989.

[BDP00] A. Blanco, M. Delgado, and M. C. Pegalajar. A real-coded genetic algorithm for training recurrent neural networks. *Neural Networks*, 14:93–105, 2000.

[BGCL00] Antony F. Bowers, Christophe Giraud-Carrier, and John W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 81–88. Morgan Kaufmann, 2000.

[Bli98] E. Blindauer. Méthodes d'encodage génétique de réseaux neuronaux. Technical report, Université Louis Pasteur, 1998.

[BMS90] Richard K. Belew, John McInerney, and Nicol N. Scharudolph. Evolving networks: Using the genetic algorithm with connectionist learning. Technical Report CSE Technical Report CS90-174, University of California, San Diego, CA, 1990.

[Cam97] C. Campbell. Constructive learning techniques for designing neural network systems. In C. T. Leondes, editor, *Neural Network Systems Technologies and Applications*, San Diego, 1997. Academic Press.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cyb88] G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical report, Department of Computer Science, Tufts University, Medford MA, 1988.

[DJ01] W. Duch and N. Jankowski. Transfer functions: hidden possibilities for better neural networks. In *9th European Symposium on Artificial Neural Networks*, pages 81–94, Bruges, Belgium, Apr 2001.

[DLLP97] Thomas G. Dietterich, Richard H. Lathrop, and Tomas Lozano-Perez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.

[Duc97] W. Duch. Scaling properties of neural classifiers. In *Third Conference on Neural Networks and Their Applications*, pages 189–194, Kule, Poland, Oct 1997.

[Fah88] Scott E. Fahlman. An empirical study of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, Carnegie Mellon University, 1988.

[FL90] Scott E. Fahlman and Christian Lebiere. The cascade correlation learning architecture. Technical Report CMU-CS-90-100, Carnegie Mellon University, 1990.

[GB99] John C. Gallagher and Randall D. Beer. Evolution and analysis of dynamical neural networks for agents integrating vision, locomotion and short-term memory. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, pages 1273–1280, Orlando, FL, Jul 1999.

[GK96] Christoph Goller and Andreas Küchler. Learning task-independent distributed representations by backpropagation through structure. In *Proceedings of the International Conference on Neural Networks*, pages 347–352, 1996.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Han90] S. Hanson. Meiosis networks. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, pages 533–541. Morgan Kaufman, San Mateo, 1990.

[Heb49] D. O. Hebb. *The organization of behavior*. Wiley, New York, 1949.

[Hin90] Geoffrey E. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46(1-2):47–75, 1990.

[HJ98] Jennifer Hallinan and Paul T. Jackway. Co-operative evolution of a neural classifier and feature subset. In *Proceedings of 2nd Asia-Pacific Conference on Simulated Evolution and Learning*, pages 397–404, 1998.

[HKKS01] Christoph Helma, Ross King, Stefan Kramer, and Ashwin Srinivasan, editors. *Proceedings of PKDD01 Workshop on the Predictive Toxicology Challenge*, 2001. http://falcon.informatik.uni-freiburg.de/˜ml/ptc/.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[HS93] B. Hassibi and D. Stork. Second order derivatives for network pruning: optimal brain surgeon. In *Advances in Neural Information Processing Systems 5*, pages 164–171, 1993.

[HSG90] Steven A. Harp, Tariq Samad, and Aloke Guha. Designing application-specific neural networks using the genetic algorithm. In *Proceedings of NIPS 1989*, pages 447–454, 1990.

[KFS95] R. King, C. Feng, and A. Shutherland. STATLOG: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3):259–287, 1995.

[Kit90] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[Koz98] John R. Koza. Genetic programming. In James G. Williams and Allen Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 39, pages 29–43. Marcel-Dekker, 1998.

[KY97] Tin-Yau Kwok and Dit-Yan Yeung. Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Trans. on Neural Networks*, 8(3):630–645, 1997.

[LDS90] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, San Mateo, CA, 1990. Morgan Kaufmann.

[Llo02] John W. Lloyd. Knowledge representation, computation and learning in higher-order logic. In preparation, 2002.

[LSS94] S. Lonardi, A. Sperduti, and A. Starita. Encoding pyramids by labelling RAAM. In J. Vlontzos, J-N. Hwang, and E. Wilson, editors, *Neural Networks for Signal Processing IV, Proceedings of The 1994 IEEE Workshop*, pages 651–660, Piscataway, NJ, 1994. IEEE Press.

[Man02] Martin Mandischer. A comparison of evolution strategies and backpropagation for neural network training. *Neurocomputing*, 42:87–117, 2002.

[MGU99] Martin Mandischer, Hannes Geyer, and Peter Ulbig. Comparison of neural networks, evolutionary techniques and thermodynamic group contribution methods for the prediction of heats of vaporization, 1999. http://citeseer.nj.nec.com/mandischer99comparison.html.

[MIA00] Malik Magdon-Ismail and Amir F. Atiya. The early restart algorithm. *Neural Computation*, 12(6):1303–1312, 2000.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

[MP88] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1988.

[MS89] M. Mozer and P. Smolensky. Skeletonization: a technique for trimming the fat from a network via relevant assessment. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. Morgan Kaufmann, 1989.

[MSG99] D. Moriarty, A. Schultz, and J. Grefenstette. Reinforcement learning through evolutionary computation. *Journal on Artificial Intelligence Research*, 11:241–279, 1999.

[MTH89] G. F. Miller, P. M. Todd, and S. U. Hedge. Designing neural networks using genetic algorithms. In D. J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, June 4-7, 1989, Arlington, VA*, pages 379–384, 1989.

[Ng01] Kee Siong Ng. Neural networks for structured data, 2001. Department of Computer Science, Australian National University, Honours thesis.

[Par85] D. B. Parker. Learning logic. Technical Report 47, Centre for Computational Research in Economics and Management Science, MIT, 1985.

[PF97] Foster J. Provost and Tom Fawcett. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions. In *Knowledge Discovery and Data Mining*, pages 43–48, 1997.

[PK94] D. S. Phatak and I. Koren. Connectivity and performance tradeoffs in the cascade correlation learning architecture. *IEEE Transactions on Neural Networks*, 5(9):930–935, November 1994.

[Pol90] Jordan B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1–2):77–106, 1990.

[PWKJ02] N. S. Philip, Y. Wadadekar, A. Kembhavi, and K. B. Joseph. A difference boosting neural network for automated star-galaxy classification. *Astronomy and Astrophysics*, 385(3):1119–1133, 2002.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[RdR00] Jan Ramon and Luc de Raedt. Multi instance neural networks. In *Proceedings of the International Conference on Machine Learning 2000 Workshop on Attribute-Value and Relational Learning*, 2000.

[RHW86] David E. Rumelhart, Geofferey E. Hinton, and Ronald J. Williams. *Parallel Distributed Processing*, volume 1: foundations, chapter 8: Learning Internal Representations by Error Propagation. The MIT Press, 1986.

[RS00] Edmund T. Rolls and Simon M. Stringer. On the design of neural networks in the brain by genetic evolution. *Progress in Neurobiology*, 61:557–579, 2000.

[Sel02] Florin M. Selaru. Artificial neural networks distinguish among subtypes of neoplastic colorectal lesions. *Gastroenterology*, 122(3):606–613, 2002.

[SFL$^+$00] S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan. Cost-based modelling for fraud and intrusion detection: Results from the JAM project. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 130–144. IEEE Computer Press, 2000.

[SH96] Jürgen Schmidhuber and Sepp Hochreiter. Guessing can outperform many long time lag algorithms. Technical Report IDSIA-19-96, Instituto Dalle Molle di Studi Sull Intelligenza Artificiale (IDSIA), 1996.

[SM01] K. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. Technical Report AI01-290, Department of Computer Science, University of Texas as Austin, 2001.

[Spe93] A. Sperduti. Labelling RAAM. Technical Report TR-93-029, ICSI, University of California Berkeley, 1993.

[SS97] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, May 1997.

[Teo] Teoma search engine. http://www.teoma.com/.

[Tes95] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[TJ88] G. Tesauro and R. Janssens. Scaling relationships in back-propagation learning. *Complex Systems*, 2:39–44, 1988.

[TL90] M. F. Tenorio and W. T. Lee. Self-organizing network for optimum supervised learning. *IEEE Transactions on Neural Networks*, 1(1):100–110, Mar 1990.

[UHT91] K. P. Unnikrishnan, J. J. Hopfield, and D. W. Tank. Connected-digit speaker-dependent speech recognition using a neural network with time-delayed connections. *IEEE Transactions on Signal Processing*, 39:698–713, 1991.

[UV94] K. P. Unnikrishnan and K. P. Venugopal. Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks. *Neural Computation*, 6(3):469–490, 1994.

[Wer74] P. J. Werbos. *Beyond Regression: News Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[Wer90] P. J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78:1550–1560, 1990.

[Yao98] X. Yao. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91:83–90, 1998.

[Yao99] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–144, 1999.