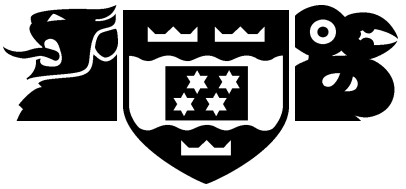
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Genetic Programming for Multi-class Object Classification

Will Smart

Supervisor: Dr Mengjie Zhang

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

This report describes the use of Genetic Programming (GP) to solve multiple-class object classification problems. Objects is reduced to feature vectors containing four features. GP is then used to evolve classifiers for the feature vectors. Objects are taken from four image datasets of increasing difficulty. Three research directions are discussed in this report.

Three *classification strategies* are compared, including two new ones, Centred Dynamic Range Selection (CDRS) and Slotted Dynamic Range Selection (SDRS). The new strategies were found to improve the performance of the system, over the standard SRS. This is especially true for harder problems with classes in an arbitrary order.

Gradient-descent search on individual programs is introduced to GP in this report. GP is still used as a global beam search, but another, local gradient-descent search is made on programs. The subject of the search is the numeric terminals of the programs. The use of gradient-descent markedly improves the rate of improvement of the population on all problems tried.

Simplification of programs during evolution is introduced in this report. An algorithm is described that removes redundancy from (simplifies) a program, and is used on all programs periodically during evolution. Simplification was found to normally improve the final accuracy of the system.

Acknowledgments

Thank you to Mengjie Zhang, my supervisor, who gave much effort to hone this project into a useful piece of literature and gave so much time and effort to get me published.

Thank you to Victor Ciesielski and RMIT for the RMITGP package which was used for all experiments.

Thank you to Bunna and Urvesh.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Genetic Programming (GP) is relatively new discipline, defining a form of Automatic Programming [1, 2]. The field is growing quickly, but there is still much room for improvement of the GP methodology. In GP solutions for a problem are represented as highly expressive *computer programs*. The program representation may be as a tree made of primitives, which may assembled in novel ways by the GP process. This creativity of program structure, and so the final solution, makes GP an interesting research area. [3] emphasizes this creativity as a design problem.

GP has its foundations in nature, such as Darwinian evolution and genetic theory [1]. As such the answers to problems in GP can often be found through this analog. One can picture programs as creatures, generations as part of the creatures' lifecycle, etc.., and through this analog find inspiration. This is a great attribute of the GP discipline, and will attract research by those who can make links between similar areas.

In order to build computers that can emulate humans in how they interact with the world, we must at some stage give them sight to some degree. This will neccesitate teaching them to classify objects.

GP has been used to determine classes of images for both classification [4, 5, 6] and detection [7, 8, 9, 10] of images, with reasonable success. Often the programs produced make reference to either low-level pixel values [5], or very high-level features [11] of the image.

The first makes for an overly-complex system, that has many features and is not very suitable to genetic programming due to the large number of terminals. Typically the systems spent a long time to train, or were terminated before reaching a solution due to a time limit.

The second makes the system domain-specific, such systems work only on the problem they are designed for. Prior knowledge of the problem is required, and an 'expert' is needed to decide how best to use the knowledge. As such, these systems are difficult to generalize to other problems.

A GP program will typically return a single real number [1]. For object classification this must be turned into a class label. For two-class systems, the sign of the result is a good discriminator [5, 6]. For problems with more classes, the situation is less simple. *Static Range Selection* (SRS) is an obvious technique that can be used [4, 9], but there are clear limitations in this method. The first is that the performance of the system depends too much upon the order that the classes are defined. The second is that this method converges very slowly on difficult problems, or those with many classes.

GP programs usually have both *feature terminals* and *numeric terminals* for object classification. While feature terminals correspond to certain attributes of the image, numeric

terminals correspond to some random numeric values set at their creation. Feature terminals define the input from the environment, and remain unchanged during evolution. Numeric terminals traditionally also remain unchanged [4, 5, 6, 7, 8, 9]. However, the numeric terminals would alter the performance of the program continuously, for better or worse, if changed.

A parallel can be drawn with Neural Networks (NNs) which have weights that have many of the same properties. However, in NNs these weights can be trained and updated using backward error propagation [12]. It would be very interesting to investigate whether the same could be applied to the numeric terminals in GP, and whether this can lead to better performance for object classification.

The program representation in GP is very expressive, able to define a vast number of program structures and make-up. However it is clear that, given the simple operators and terminals that ure used [4, 5, 6, 7, 8, 9], programs often have redundant structures. An addition operator that adds together two numeric terminals is such an example, as it is equivalent to one numeric terminal that has the value of the sum. Typically, these redundancies are removed after the evolutionary process for purpose of analysis. However, keeping these redundancies in the evolutionary process makes the search space very large. The removal of certain types of redundancy *may* affect the performance of the genetic operators that are applied to the program, or produce a population of more concise programs. Thus it would be very interesting to investigate the effect of removing these redundancies during evolution for object classification.

1.2 Goals/Research Questions

The goal of this report is to develop new approaches for application of GP to multiple-class image classification problems, while maintaining domain-independence. The new approaches will be assessed on a sequence of object classification problems of increasing difficulty, to investigate whether they represent an improvement over the basic approach. Specifically the following research questions will be addressed:

- Using GP, can we find a new classification strategy that will be more effective on multiple-class problems than Static Range Selection (SRS)? This is addressed in chapter 4.
- Can we utilize a gradient-descent search on the numeric terminals of individual programs periodically during evolution, and will it either raise accuracy or speed up evolution? This is addressed in chapter 5.
- Can we develop an algorithm to remove redundancy from individual programs periodically during evolution, and will it either raise accuracy or speed up evolution? This is addressed in chapter 6.

1.3 Contributions

This project report makes three major contributions:

1. This report shows how to dynamically change the mapping between program results and output class identifier, periodically during evolution.

   Two new *classification strategies* were developed for this purpose. The first newly established strategy is named Centred Dynamic Range Selection (CDRS). During evolution

it dynamically moves the thresholds between class regions. The second newly established strategy is named Slotted Dynamic Range Selection (SDRS). The real number line is partitioned into sections, which are dynamically assigned classes periodically during evolution.

These new methods have been found to be better than Static Range Selection [4] on most of the object classification problems described here, especially on difficult problems with many classes in an arbitrary order.

Part of this work was submitted to and accepted by the *Image and Vision Conference 2003* as the following:

> Will Smart, Mengjie Zhang. Classification strategies for image classification in genetic programming.

2. This report shows how to use a new form of hybrid search in genetic programming. The program structure and initial values are evolved using the usual genetic beam search globally. Locally the programs are refined using gradient-descent search on the numeric terminals.

   Both online (stochastic) and offline learning algorithms in gradient-descent search were developed and implemented.

   For the online learning algorithm, a learning rate formula was devised that is optimal for many of the programs that may be produced, and is a good starting point for many others. The formula is based on the total effect of the changes to the numeric terminals on the program output. It calculates the appropriate distance to move on the cost surface to reach the global minimum.

   The use of this hybrid search has been found to dramatically decrease the number of generations required to find a solution to the training problem (compared to a purely genetic beam search). More importantly final accuracy of harder problems was dramatically improved when the hybrid search was used (compared to a purely genetic beam search).

   Part of this work was submitted to *The Australasia Workshop on Data Mining and Web Intelligence* for review and publication:

   > Will Smart, Mengjie Zhang. Applying online gradient descent search to genetic programming for object recognition.

3. This report shows how to remove certain types of redundancy of the evolved programs during the evolutionary process. A new algorithm is developed and applied periodically during evolution, simplifying the programs in the population.

   Several types of redundancy are removed from the programs, including: numeric terminals added/subtracted/divided/multiplied together, `if` statements that always return one particular branch, common factors in additions and branches divided by themselves.

   On some problems simplification made for an increase in performance of the system.

1.4 Structure of Document

The rest of this report is organized as follows. Chapter 2 presents the foundations of along with the current state-of-the-art in multiple-class classification using GP, and gradient-descent methods.

Chapter 3 contains a detailed description of the datasets and settings used in experiments listed in later chapters.

Chapter 4 describes the classification strategies tried including two that are new. Results and discussion for the experiments on these classification strategies are presented.

Chapter 5 describes in detail the mathematical foundations and algorithm used to apply gradient-descent to GP programs. Results and discussion of gradient-descent experiments are presented.

Chapter 6 describes the algorithm used to simplify a program, along with the types of redundancy removed. Results and discussion of simplification experiments are presented.

Chapter 7 presents the overall conclusions for the project, including answers found for research questions asked along with lessons learnt in general, and a list of future work.

Appendix A briefly describes how to use the programs that were used in this project.

Chapter 2

Background

2.1 Genetic Theory

Genetic programming and genetic algorithms define methods by which a search is made for solutions to problems, using evolutionary principles. These evolutionary principles are often named Darwinian, after Charles Darwin who found such principles as prevalent in the shaping of earthly creatures.

Natural selection dictates that creatures are selected to produce offspring partly by their fitness. According to Darwinian evolution, in nature, creatures will only reproduce if they are adequately adept at life, and creatures that are very good at all aspects of their lives will contribute substantially to the next generation, by producing many offspring.

Offspring contain copies of the genome (information) that was used to create their parent(s). If the parents were fitter than average, ensured in general by natural selection, then the offspring may be fitter than average.

The copies may not be perfect, if they were perfect copies the offspring would always have the same potential as their parent. This is not adequate, as the fitness should increase over time, so changes are made to the offspring's genome, while still having genome based on the parent(s).

2.2 Evolutionary Computation

Evolutionary computation is a general term, referring to systems (Evolutionary Algorithms or EAs) that use models of genetic theory as the key element in computation problem solving.

As such the EAs describe a general topic, including the basic methods: Evolutionary Programming (EP), Evolutionary Strategies (ESs) and Genetic Algorithms (GAs) [13].

The basic algorithm for EAs is shown as a flowchart in figure 2.1.

The figure shows the general process: A population of individuals is constructed initially. Each individual is evaluated according to the fitness function. Then the system loops, selecting, recombining, mutating, evaluating, and finally reconstituting the selected survivors of these operations into the new population. The system will be set with a trigger to indicate when evolution should stop, such as when the training problem has been solved, or after a set number of iterations of the loop.

It is useful at this stage to look at the different EAs that have been developed in parallel, giving a useful background to GP. What follows is a very brief discussion of EP and ESs , followed by an in depth look into GAs, and the offshoot Genetic Programming (GP).

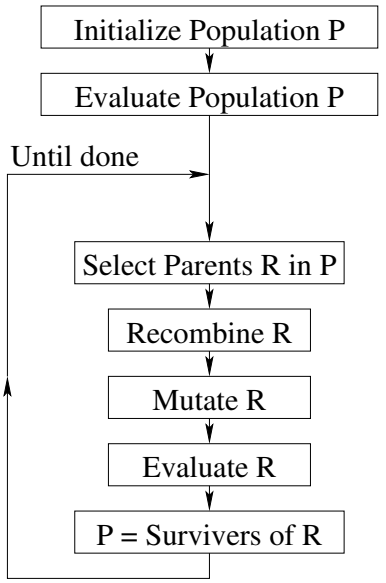


Figure 2.1: A general flowchart for operation of Evolutionary Algorithms.

2.2.1 Evolutionary Programming

Evolutionary Programming (EP) was introduced by Fogel et al. in the 1966 book *Artificial evolution through simulated evolution* [13].

The representation used by EP was generally changed according to the problem, such as a graph for the traveling salesman problem. All individuals in the population are mutated, and the next population is formed probabilistically based on fitness from the parents (before mutation) and the children (after mutation) to produce the same number of individuals as before [13].

Recombination was not used as such, but a flexible mutation operator could produce the same effect [13].

2.2.2 Evolutionary Strategies

Evolutionary Strategies (ESs) were introduced by Rechenberg in 1973 and extended by Schwefel in 1981 to include recombination and populations larger than one [13].

Parents are chosen uniformly randomly, and are subjective to recombination and mutation, with the best children, or parents and children, selected to become the new population.

2.2.3 Genetic Algorithms

Genetic Algorithms (GAs) was introduced by John Holland in the 1960s, and was described in the 1975 book *Adaption in natural and artificial systems* by John Holland [13].

GAs use *chromosomes* as a representation, such as *bitstrings*. As such GAs are very expressive for systems of a fixed number of parameters. The chromosomes contain a fixed-length, ordered set of *alleles*. A common example has the alleles as bits and the chromosomes as fixed-length bitstrings.

The systems will evolve one set of chromosomes to another through operators inspired by genetic theory. The common operators are crossover (or recombination) and mutation.

Crossover is a distinguishing feature of GAs. It is crucial to the success of the algorithm. Crossover comes in a number of varieties: Single-point, two-point, and 'parametized uniform crossover' [14].

Unlike EP and ESs, mutation does not play a large foreground role in GAs [14]. It is used largely to avoid fixation of the system onto a particular locus.

Selection Methods in GAs

This section describes some methods of selection used in GAs, as applicable to our use of GP.

Selection is essential to the success of any evolutionary algorithm as selection is the only time that the fitness of an individual is taken into account. This selection must be performed so as to prefer fitter individuals (survival of the fittest), however diversity must be maintained in the population by some factor, otherwise progress will be slowed. The following are described in [14].

- Roulette-Wheel Sampling: A roulette wheel is invisiged, a slice is placed on the wheel for each individual, with the size of the slice proportional to the individual's fitness. The wheel is then spun (a random point is determined) and the owner of the slice is selected. In this way fitter individuals will have a greater chance of being selected.
- Elitism: Refers to the technique of selecting the fittest individuals straight into the next population, in order to at least maintain the current population fitness.

2.3 Genetic Programming

Genetic Programming (GP) represents an extension to both GAs and automatic programming [1].

The term was coined in 1990 by Hugo de Garis in [15]. However more relevant introductions to today's meaning of GP were by John Koza in [1, 2].

In GP, the representation of individuals in the population is by *computer programs*, which can vary in size and shape [2].

2.3.1 Program Representation

The particular representation often used for GP programs is LISP S-expressions [1].

LISP contains only two forms of entity: atoms and lists. An example of an atom is `7` or `A`. A list is a sequence of items enclosed by parentheses, such as `(+ 7 A B C)`.

A symbolic expression (S-expression) is simply a single list or atom.

It is easy to see that a list could contain another list as one of its items, and that list could contain another, and so on. As such the complexity of an S-expression is potentially infinite.

S-expressions can be pictured as a parse tree, such as that of figure 2.2.

Thus it can be seen from figure 2.2 that the S-expression has two forms of atoms. Internal nodes are *functions*, which can perform arbitrarily complex computational functions, taking a fixed number of input values, and returning a single output value to the next level. At the leaf nodes of the parse tree are *terminals* which may perform some calculation and return a single result to the next level.

The value of the S-expression may be *evaluated* by moving from left to right, evaluating functions and using terminals as they pass. This is due to the nature of the S-expression encoding the parse tree.

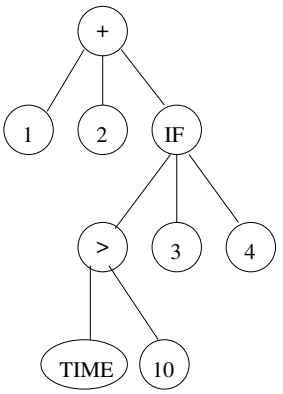


Figure 2.2: A parse tree built from the S-expression : `(+ 1 2 (IF (> TIME 10) 3 4))`.

Functions and terminals can be seen to return values. In the normal case, return types are restricted to one type, often real numbers.

In Strongly Typed Genetic Programming (STGP) [16], this restriction is relaxed. However STGP brings difficulties in selecting a fitness function, and function/terminal sets.

The value returned from the evaluation of the parse tree is termed the *result* of the program in this report. The result is used to perform the desired task. For example in regression the result may indicate the value of the function for a given x (input as a terminal). In training, this value could be turned into an error by taking into account the desired value, and this will contribute to the program fitness.

2.3.2 Primitive Sets

In GP the primitive set consists of a terminal set and a function set. The *terminal set* is the set of allowable terminals. Typically these are either variable atoms (inputs from the environment) or constant numerics [1].

The *function set* of the system is the set of functions that can be used in programs evolved by the system. This may include standard arithmetic operators, such as addition, or in fact any function that returns a result and takes a set number of parameters.

Both the functions of the function set and terminals of the terminal set are chosen for closure and sufficiency.

Closure demands that any function be able to produce valid output for any values of arguments that could occur. For example, if standard division were defined as a function, care would be required to handle the case of a division by zero, possibly by using a 'not-a-number' value, or just returning zero (referred to as 'protected division').

Sufficiency demands that the functions and terminals available be able to form a solution to the problem. For example, a regression problem aiming for the function x^2 will never reach the target if the function set only contains addition, the function set is insufficient.

2.3.3 Fitness Function

The *fitness function* is a function that takes as its input a program tree, and outputs a value related to how fit the program is, similar to that of GAs [1]. This value could be a real number between some best and worst values. The fitness function chosen depends on the task to be performed, and in some ways *defines* the task. For example, for regression problems Mean Squared Error (MSE) may be used, from selected points along the input dimension(s). This

would give a high value for a bad match, and a low value for a good match, though other fitness functions may be in the opposite direction.

2.3.4 Genetic Operators

In GP three main operators are used to produce a population from the previous generation. They are along the same lines as those of GAs. The first is reproduction, or elitism. The second is mutation where a parent is selected. Some random subtree is removed from the parent and a randomly generated subtree is put in its place and the resulting tree is used as the child.

The third is crossover. The standard crossover is as follows: two parents are selected and random subtrees from both are chosen, removed, swapped, and replaced. The resulting programs are used as children.

2.3.5 Parameters

GP has many other parameters such as population size, maximum generations, rates for the various genetic operators, maximum program depth, and others.

Heuristics for using these parameters have been sought [17, 18]. Possibly the best settings for these parameters are found by experience with the system, and the problem.

2.4 Gradient-Descent Search and Neural Networks

Machine learning involves teaching a system to do a task. We assume here that the performance of the system at the task can be numerically quantified, for example a cost measure C which is lower for a system that performs better at the task. Furthermore the system has a fixed number of parameters which affect this cost. So machine learning, in this instance, involves searching for the system parameters that will minimize the value of C.

As such C is a surface over the parameters' values, that is, a surface over N dimensions where N is the number of such parameters.

2.4.1 Gradient-Descent

If the gradient of the cost surface C can be found, then a method to find the parameters at a minimum (or at least low) point on the cost surface is to simply move the parameters proportionally to the gradient. The gradient can be seen as being in the uphill direction along the surface at the point of current parameters. Moving negatively proportionally to the gradient would move the parameter point toward a point of lower cost [12, 19].

This forms the basis of the gradient-descent method, also called gradient-ascent and hill-climbing with minor differences.

2.4.2 Neural Networks

Neural Networks (NNs) are described (very briefly) here in order to clarify the analog to be made with GP programs.

NNs consist of a number of interconnected nodes. These nodes are generally arranged in an order from input nodes to output nodes [19].

The nodes are connected by links, and each link has a numeric *weight*, which determines the strength of connection between the nodes. Internally, nodes produce output according to some function of their inputs. Thus, if the desired output for a set of inputs is known, an

error can be found as the squared difference between this and the output. Over the dataset a cost function can be built.

The weights of the links will affect the way the signals entering the input nodes are transformed to the output nodes. As such the weights can be seen as parameters of the cost surface.

The gradient of this cost surface is found by differentiating with respect to the weights, which can be performed using the chain rule, so long as standard node types are used.

Given this gradient vector, gradient-descent can be used to find a region of lower cost than the current set of parameters (weights). The method used for this is called backward-error-propagation and was introduced in [12].

2.5 Multiple-Class Object Classification

Object classification is a task of Data Mining [20] where a class value is to be sought from details of an object, which is a computer image. Unlike object detection, where position is to be found, only the class is required for classification. In the case of multiple-class classification, the class to be found may be one of a number of classes [7]. For binary classification the number of possible classes is two.

Objects in classification are transformed into patterns. These patterns contain a set of features that describe the objects, and possibly a class identifier that denoted the correct classes of the objects, for training.

Classification is similar to clustering, but is a form of *supervised learning*. In classification a dataset of patterns is created, each containing the class identifier for the pattern. The system can *learn* the mapping of feature space to class.

A training set is used to teach the system, producing a classifier. A test set is used to measure the performance of a learned classifier.

A validation set can be used to control overfitting, where the system learns the training set too well, while not generalizing to the test set. The validation set is used to detect this, and allows measurement of system performance after training, but before overfitting.

2.5.1 The Features

Great variation exists in the features used for object classification The features range from very low-level pixel values [5] to very high-level, domain-specific features [11]. The former has problems of tractability, as the required program size goes up, and a requirement for large training set sizes to avoid overfitting. The later is designed for the problem, and may be herd to transfer to other problems.

Modern systems often use pixel-statistics, such as average intensities of regions, as features [4, 7, 8, 9]. This approach combines the advantages of domain-independence found with low-level features, and tractability found in high-level features.

2.6 Multiple-Class Object Classification in GP

In order to classify objects using GP, three modifications must be made to the basic GP engine: fitness function, feature terminals and classification strategy.

2.6.1 Fitness Function

In object classification the fitness function is often *accuracy*, that is the fraction of objects correctly classified [5]. This is the simplest measure, though others could be used, such as

mean squared error [4].

2.6.2 Feature Terminals

The object attributes are input into the GP programs via the feature terminals. A feature terminal will randomly select a feature to return at program creation, and will not change this feature index throughout evolution.

2.6.3 Classification Strategy

The classification strategy refers to the algorithm that converts the result of a program to a class index/title.

Binary Classification Strategy

A binary classification strategy will discern between two classes (eg object and background) and represents a simple problem, generally one class is assigned if the result of the program is negative, and the other if it is zero or positive.

Multiple-Class Classification Strategy

Multiple-class classifications refer to those intended to discern between more than two classes.

Possibly the obvious strategy for classifying objects is Static Range Selection (SRS) [4]. Other options include Dynamic Range Selection, Class Enumeration, and Evidence Accumulation as stated in [4].

2.7 Related Work

2.7.1 GAs, NNs and GP

[13] presents a broad overview of evolutionary computation, focusing on and contrasting the three major arms: GAs, ESs and EP. [14] describes understandably GAs and their origins and applications, including a flexibility of chromosome representation.

John Koza in [1, 2] presents the major early work in GP, presenting the new LISP S-expression representation, and conforming genetic operators. [16] introduces Strongly Typed Genetic Programming (STGP).

[17] compared crossover and mutation in genetic algorithms. [18] did the same for GP, concluding that the difference between the effects of mutation and crossover is small, but, interestingly no combination of the two performs consistently better than either one alone.

[20] described a comprehensive, very general coverage of Knowledge Discovery in Databases (KDD). [12] introduced the backward-error-propagation method for training feed-forward NNs using gradient-descent.

[19] described NNs in the context of evolving NNs using EAs, in what was termed EANNs.

[3] hilighted the creativity inherent in evolutionary algorithms, by enforcing a set of real-world rules on individuals, and defining fitness as how far the individual can move, given an evolved structure of actuators and joints. The creatures were then built using rapid prototyping, with very interesting results.

2.7.2 GP for Object Classification and Detection

[4] introduced some useful strategies for dealing with multiple-class classification problems in GP. Methods discussed include binary decomposition, static range selection (SRS), dynamic range selection (DRS) and class enumeration. [5] used GP to evolve classifiers for textures, a task similar to the classification described in this report, but raw pixel values were used. [6] described the application of GP to image detection of ships. The approach used a range of pixel statistics as features. [7, 8, 9] applied GP to three detection problems ranging from easy to very hard. Pixel statistics were used as features with [8, 9] emphasizing domain-independence of the system. [8] also introduced a new fitness function which included a *false alarm area*.

[11] described use of GP to evolve *processing trees* for processing of images, and classification as target or non-target. A large number of domain-dependent features in used. GP was compared to a binary tree classifier and an NN and achieved better results. [10] used a similar technique to detect the centres of faces in images but did so in a more domain-independent way. [21] produced very good results using GP to segment MRI scans using very basic pixel neighbourhood statistics.

2.7.3 Summary

GP has been successfully used to perform object classification and detection tasks, but with limited success for multiple-classes. Backward error propagation has arisen as an effective method to train NNs.

However, some points missing from the literature include:

- Better classification strategies for multiple-class classification problems.
- Use of gradient-descent on individuals in GP.
- Analysis of redundancies contained within GP programs during evolution, and the effect of removal of the redundancy.

These topics will be investigated in this report.

Chapter 3

Datasets and Experimental Setup

The five datasets are in two categories: computer-generated shapes, and coins.

All patterns in the datasets originated as compound images, each containing many (non-overlapping) objects of interest. The position of each object of interest was determined manually, allowing the objects to be *cut-out* into a large number of small images, each with a single centred object. Each dataset defined a standard size for all its cutouts, the size is just big enough to fit the largest object.

3.1 Features used in Dataset Patterns

Each cutout object was then reduced to four, domain-independent features, together the features made up the *pattern* for the object. The features used are found in table 3.1 with regions in figure 3.1, where s is the width and height of the (square) cutouts.

The features could easily be improved upon, especially for the harder datasets, but this is beyond the scope and aim of this project.

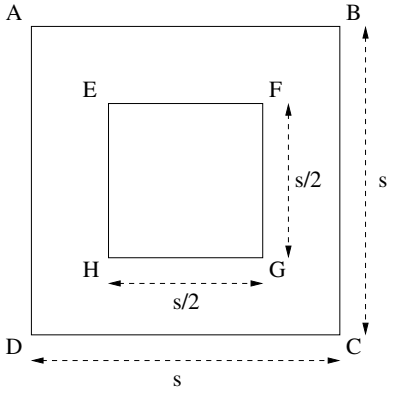


Figure 3.1: The regions used for features.

The four features were chosen for the following characteristics:

- Rotation invariance to a reasonable degree.
- Intensity and variance for the whole cutout, as the major discriminators for the classes are the intensity and variance of the whole cutout.
- Intensity and variance of the centre region, often considered to be important in classifying objects.

Table 3.1: Features used for experiments.

Index	Feature (region in figure 3.1)
1	Average intensity of region covering the entire object (ABCD)
2	Intensity variance of region covering the entire object (ABCD)
3	Average intensity of region covering the centre square (EFGH)
4	Intensity variance of region covering the centre square (EFGH)

The patterns from each dataset were split into three sets: training, validation and test. Each set contains approximately one third of the patterns in the dataset. The sets are produced while ensuring that the proportions of each class in each set is approximately the same.

3.2 Datasets

3.2.1 Shapes

The shape images used feature computer-generated noisy squares and circles in classes of different intensities on a noisy background. In total there were 20 images.

An example image is shown in figure 3.2.

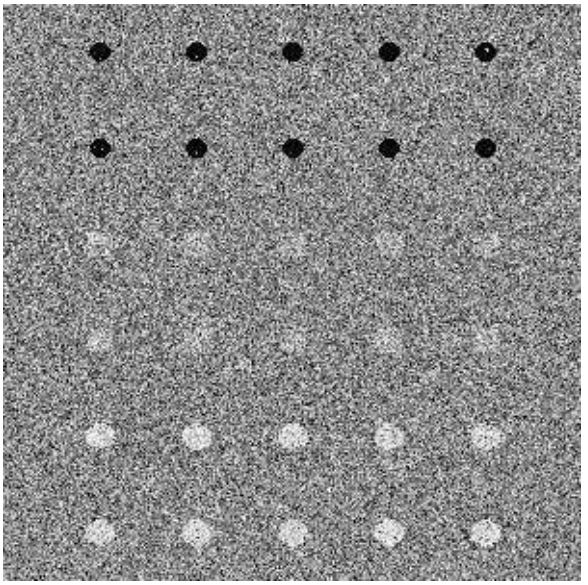


Figure 3.2: An example Shape dataset image.

Two datasets, Shape1 and Shape2, were formed from the same images. Shape1 has its classes ordered by intensity of the objects, but Shape2 is in an arbitrary, different order. This makes Shape2 far harder than Shape1 when using some classification strategies, such as SRS, where the order of the classes is important.

Table 3.2 list the order of the classes, along with the numbers of objects in each class, for the shape datasets.

Table 3.2: Classes and set sizes in the Shape1 and Shape2 datasets.

Object Class	Intensity	Shape1 class index	Shape2 class index	Number in training set	Number in validation	Number in test set
Black Circles	10 ± 5	1	2	70	70	60
Background	140 ± 50	2	1	42	37	34
Grey Squares	180 ± 50	3	3	70	70	60
White Circles	220 ± 30	4	4	70	70	60

This dataset had the objects in each image logically cut out to form 713 16×16 images, each of these object cutouts is either centred on some shape in an image, or positioned in an area of background away from any shapes. These cutouts then had the features extracted to form patterns for use in the classification system, see section 3.1 for details of the features used.

3.2.2 Coins

Three datasets, named Coin1, Coin2 and Coin3, containing New Zealand coins were used.

These problems were of increasing difficulty, with Coin1 similar in difficulty to Shape1, Coin2 is moderate, Coin3 was very difficult (with these features).

Coin1 contained 10 cent coins on a noisy background. Coin2 contained 5 cent and 10 cent coins on a "white", almost uniform background. Coin3 contained 5 cent and 10 cent coins on a noisy background. Examples of images in the coin datasets are shown in figure 3.3.

Each image contains equal numbers of all classes, including heads and tails side coins. Table 3.3 shows a list of the object classes and their indices in these datasets.

Table 3.3: Classes in the Coin datasets.

Object Class	Coin1 dataset class index	Coin2 dataset class index	Coin3 dataset class index
Noisy background	3		1
White background		1	
5c tails-up		2	2
5c heads-up		3	3
10c tails-up	2	4	4
10c heads-up	1	5	5

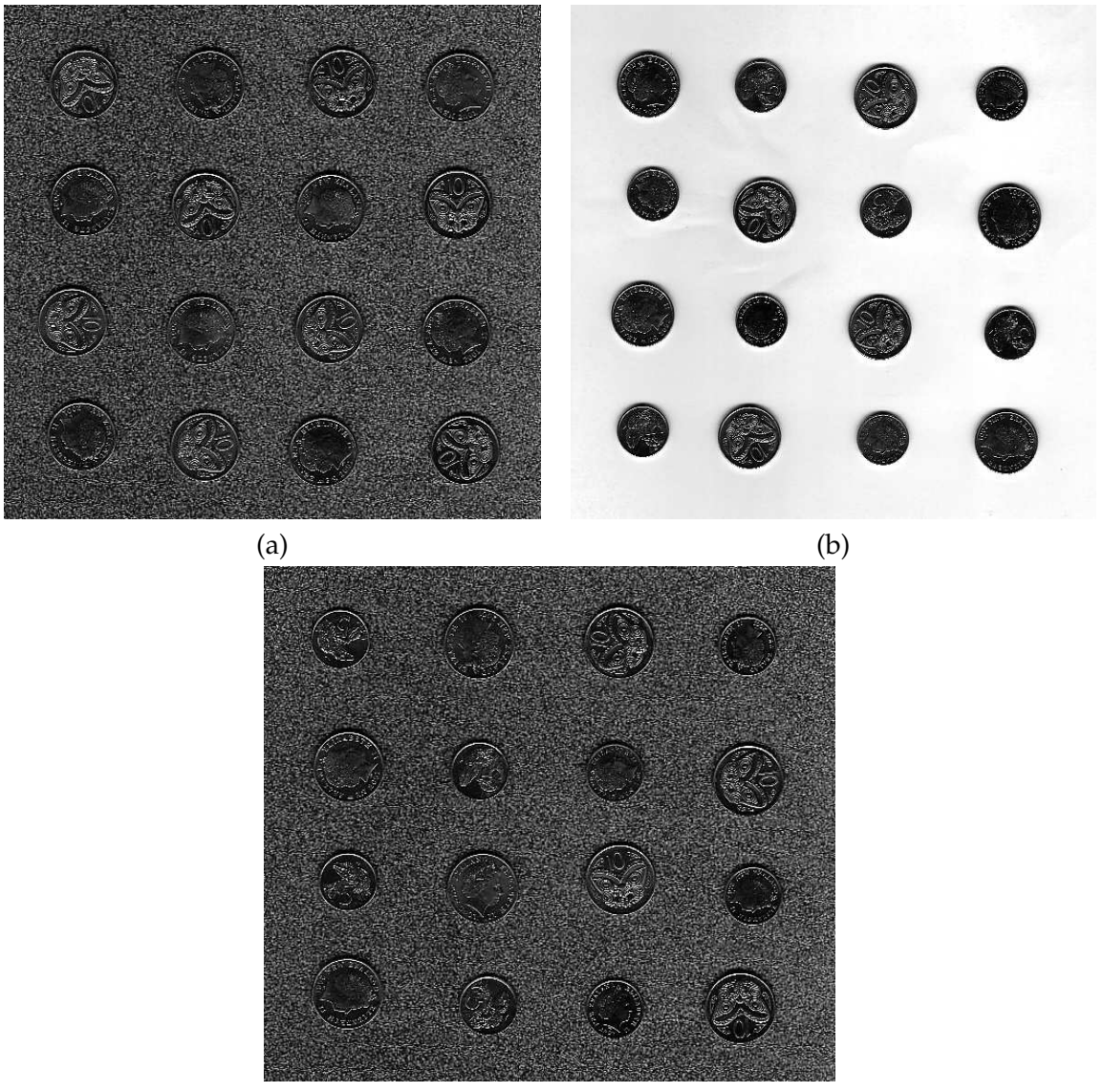
Table 3.4 lists the training, validation and test set sizes for the coin datasets.

3.3 Experimental Setup

GP has many parameters, such as population size, maximum number of generations, maximum program size, and others. Most experiments in the project used similar settings for most parameters. This section describes this common experimental setup. Note that these

Table 3.4: Set sizes in the Coin datasets.

Object	Number in Coin1 dataset train/val/test	Number in Coin2 dataset train/val/test	Number in Coin3 dataset train/val/test
Noisy background	64/64/64		12/8/12
White background		32/32/32	
5c tails-up		32/32/32	12/8/12
5c heads-up		32/32/32	12/8/12
10c tails-up	64/64/64	32/32/32	12/8/12
10c heads-up	64/64/64	32/32/32	12/8/12



(a) (b)

(c)

Figure 3.3: Examples of the coin dataset images, Coin1 (a), Coin2 (b) and Coin3 (c).

are indications only, and some experiments modify some of these settings locally. Table 3.5 list the common parameters used.

Table 3.5: Common GP Parameters for Experiments.

Parameter Kinds	Parameter Names	Values
Search	population-size	500
	max-depth	5
Parameters	max-generations	50
	reproduction-rate	20%
Genetic	crossover-rate	50%
	mutation-rate	30%
Parameters	crossover-term	15%
	crossover-func	85%

3.3.1 Fitness Function

The fitness function used was simply training set accuracy, that is the fraction of objects in the training set correctly classified by the program. As such the worst case is 0.0, and the best is 1.0 (100%).

3.4 Primitive Set

Five functions made up the function set for all experiments and two kinds of terminals made up the terminal set for all experiments, they are listed in table 3.6.

Table 3.6: Primitive Set.

Type	Name	Description
Function	+	Returns sum of two real numbers
Function	-	Returns difference of two real numbers
Function	$\times$	Returns product of two real numbers
Function	/	Protected division, returning one real divided by another, or 0 iff divisor is 0
Function	`if<0`	Returns second argument if first is < 0, third otherwise
Terminal	Feature Terminal	Returns the value of a feature in the current feature vector, The index of the feature returned is set at the creation of the terminal, and doesn't change
Terminal	Numeric Terminal	Returns a real number. The value is randomly set at creation within the range [-1,1].

3.5 Other Parameters

Unless otherwise stated all experiments were run with early-stopping, that is the training was stopped once an individual was produced with a fitness of 1.0, classifying the entire

training set correctly. In case this has not happened before the maximum number of generations, the evolution is stopped at that point.

Programs were created using ramped-half-and-half program generation, and limited at all stages to depth of 5. One exception to this rule is during simplification, where programs may, if neccesary, simplify to over 5 depth.

Chapter 4

Multiple-class Object Classification Strategies

In standard GP, a program, run on an input pattern, outputs a single real result. This creates a difficulty when GP is used on classification problems, as this real result must indicate a which class the program is outputting.

In binary classification problems, the sign of the program can be used to differentiate between the two classes. However, for multiple-class classification problems, the division of the possible outputs is not so obvious.

The role of the classification strategy is to perform this mapping. In this chapter three alternatives are compared, including two that are new.

4.1 Classification Strategies

Three classification strategies were compared in experiments: Static Range Selection (SRS), Centred Dynamic Range Selection (CDRS) and Slotted Dynamic Range Selection (SDRS). CDRS and SDRS are new in this project, SRS is an established, standard method.

4.1.1 Static Range Selection

When using SRS the real number line is split into regions at the beginning of evolution and these regions do not move during evolution.

Assume there are N Classes, assigned numbers from 1 to N. Classes are sequentially assigned slots of equal width along the real number line from some negative number through to the same number as positive. Also class 1 will also be allocated all numbers less than the first slot, and class N all numbers greater than the last slot.

This method is simple conceptually and in implementation. One can think of fixed *boundaries* on the number line, between class regions. For a five class problem, for example, there will be 4 boundaries (also called thresholds) between adjacent classes:

$< T_0, T_1, T_2, T_3 >$.

As shown in table 4.1

The boundary positions are set with one parameter, called *slotsize*, this denotes the distance between boundaries. For example if the slotsize were 6, we would have:

$T_0 = -9, T_1 = -3, T_2 = 3, T_3 = 9$

It can be seen that the distance between any adjacent boundaries is 6.

Table 4.1: SRS class mapping for a five class problem.

Program result (r) conditions	Corresponds to class
$r < T_0$	1
$T_0 \leq r < T_1$	2
$T_1 \leq r < T_2$	3
$T_2 \leq r < T_3$	4
$T_3 \leq r$	5

4.1.2 Centred Dynamic Range Selection

The first new classification strategy developed is Centred Dynamic Range Selection (CDRS), which is based on SRS. In this method the positions of the boundaries between classes may move periodically during evolution. Also the order of the classes associated with the regions between boundaries may change.

Figure 4.1 shows the CDRS process on some program results in a three class problem.

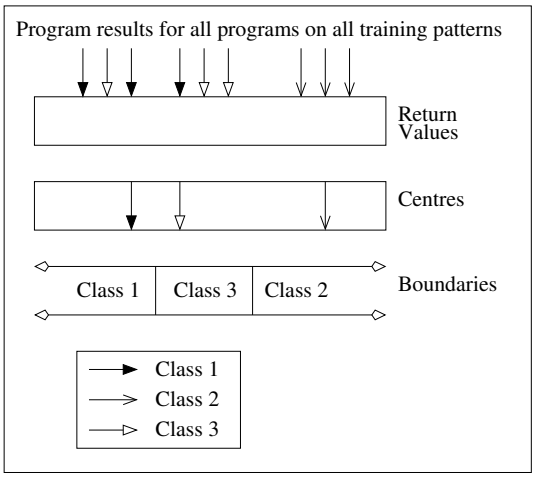


Figure 4.1: CDRS forming class regions based on some program results.

Periodically during evolution, for example every 5th generation, a routine is called to shift the boundaries and alter the order of classes to better fit the data and population.

In order to decide the new boundaries and order the following routine has been implemented and tested.

The training patterns are split into their respective object classes. The centres of program results for each of the class groups are then found according to equation 4.1.

$$\text{center}_c = \frac{\sum_p \sum_{\mu_c} f(\text{fitness}_p) \times \text{result}_{p\mu_c}}{\sum_p \sum_{\mu_c} f(\text{fitness}_p)} \tag{4.1}$$

$$f(\text{fitness}) = \frac{2 \times (\text{fitness} \times (\text{weight} - 1) + 1)}{\text{weight} + 1} \tag{4.2}$$

Where the sum over p includes all programs p in the population. The sum over μ_c includes all patterns μ_c in the training set that are of class c. center_c is the calculated center

of class c, fitness_p is the fitness of program p with a best fitness of 1 and a worst fitness of 0. $\text{result}_{p\mu_c}$ is the output of program p run on pattern μ_c. f is a function to determine the relative weighting of programs. weight is a parameter to the system.

After the class centres have been found boundaries are formed at the midpoints of adjacent centres. This also determines the order of the classes of regions between boundaries.

The boundaries are then shifted to ensure a minimum boundary to boundary width, called *slotsize*. For example, if *slotsize* was 3, a class had a boundary at 1, and another class had a boundary at 2 then the boundaries are too close together, and may be moved to 0 and 3 respectively. This may cause other regions to shrink to less than 3. The routine used here simply sweeps in one direction extending boundaries in that direction only.

Finally, all boundaries are forced within a certain total range, so that no boundary can be set beyond that range (for example from -25 to 25). This is done by discarding any program results (in an earlier step) that lie outside of this range. Also a second sweep is added to the step that ensures the slotsize. The first ensures all boundaries are above the minimum value, the second sweep goes in the opposite direction, cleaning up any boundaries that have strayed outside the range.

The function f (equation 4.2) is designed to provide a weighting for different programs, depending on a used defined weight factor **weight**. For example, if $\text{weight} = 3$, a program with perfect fitness (one) has $f(\text{fitness}) = 1.5$, while a program with the worst possible fitness (zero) has $f(\text{fitness}) = 0.5$. It is seen that the value of $\text{weight} = 3$ caused the best program to have 3 times the value of f as the worst. This is similar for other values of weight.

So the value of weight determines the relative influence on the centres, of programs with different fitnesses. If $\text{weight} = 1$ all programs contribute the same.

Using a value of $\text{weight} > 1$ produces some interesting issues not applicable to a weight of 1. There is a feedback cycle made, as the fitness of a program contributes to the boundary positions which, of course, contributes to the fitness of the program.

In the evaluation step the object patterns with the program results less than the first boundary are classified as the class that has a centre before the first boundary. The object patterns that have program results between two adjacent boundaries are classified as the class that has a centre between the two boundaries. The object patterns that have program results greater than or equal to the last boundary are classified as the class that has a centre after the last boundary. This is much the same as SRS.

Table 4.2 shows an example of calculating class centres. In this example, there are three programs on four training patterns of two classes. It can be seen that Prog 3 has 3 times the influence on the result relative to Prog 1, and Prog 2 has 2 times the influence.

CDRS is better than SRS when the classes have an optimal order. An example is classifying objects of different intensities, where far better results are achieved using SRS if the classes are arranged by order of intensity (For example the difference in results of dataset Shape1, and dataset Shape2, see table 4.3). This is because this simplifies the program required to classify well. So, using SRS, this is a required prior knowledge. Using CDRS this knowledge is not required, as both the order and position of the classes may change. So long as the best programs in a generation are fairly similar in the outputs they give for the same classes, the ranges will closely approximate the ideal positions of the thresholds.

4.1.3 Slotted Dynamic Range Selection

The second new classification strategy developed is Slotted Dynamic Range Selection (SDRS) which is an adaption of Dynamic Range Selection (DRS) introduced in [4].

In SDRS, a region of the real number line is divided into many slots. All slots are of the

Table 4.2: Example of CDRS finding centres.

Program	**Fitness**	$f(\text{fitness}_p)$	**Program results on training data**			
		(weight=3)	**Class1**		**Class2**	
			Pattern 1	**Pattern 2**	**Pattern 3**	**Pattern 4**
Prog 1	0.0	0.5	0.0	0.1	1.3	1.2
Prog 2	0.5	1.0	0.3	0.7	1.9	1.5
Prog 3	1.0	1.5	0.4	0.9	1.7	2.0
Sum $f \times \text{result}_p$			0.9	2.1	5.1	5.1
Sum for classes			3.0		10.2	
Sum f			3.0	3.0	3.0	3.0
Sum for classes			6.0		6.0	
Centres of classes			0.5		1.7	
Class regions			< 1.1		≥ 1.1	

same width (called *slotsize*, though is does not relate well to SRS or CDRS). For example, if the slotsize was 1, and the region of the real number line used was -25 to 25, there would be 50 slots.

Each slot classifies as a particular class any program result that falls into its range. These classes are dynamic, and are initially arranged in a way that mimics the effect of SRS (see section 4.1.1).

Periodically during evolution, in much the same way as in CDRS, a routine is called to *reclassify*, that is alter the slot classes, to better fit the data and population.

Figure 4.2 shows the SDRS process on a three class problem.

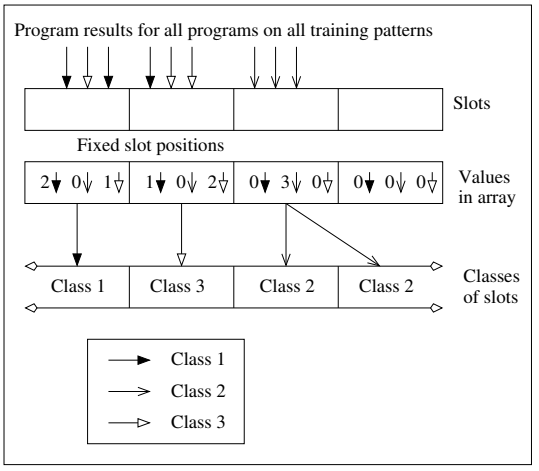


Figure 4.2: SDRS forming class regions based of some program results.

An array is made with values for each class in each slot. All values in the array are initially set to zero. All programs are then evaluated on all training patterns. For each program result that falls into a slot when run on a pattern, the value in the array for the slot and class (of the pattern) is increased by the value of a function f which takes the fitness of the program used as an argument. This f function is described in equation 4.2 on page 20 relating to CDRS and is identical here.

After these steps the array holds information about the amount of use each slot has by programs run on each class. It is then a simple task to determine which class a slot should classify as, simply take the class with the largest value in the array at the slot, as that class is often desired when a program result falls into the slot. However there may be slots into which no program result fell. For such slots the class of the nearest neighbouring slot (with counts) is used.

Using this method classes can be situated arbitrarily on the real number line, subject to a certain range and quantization. This approach has the following advantages:

- Where a class has two or more general areas of return values, they can be allocated separate regions.
- Where there is a situation where there are groups of programs that return different (but correct) results, each can be allocated separate regions.

SDRS versus DRS

DRS [4] defines a method that is similar to SDRS. In DRS each program classifies according to a map of slots, as in SDRS. However the slots contain only entries for the current program, evaluated on some portion of the training set that has been set aside for this purpose. The rest of the training set is then used to determine the fitness.

There were two main reasons to adjust DRS to SDRS with the global slot map: quantity of samples, and speed.

When DRS is used, the number of samples used to set the classes of the slots is small, maybe half the number of training examples. Using a global map that is contributed to by all programs on all training data, the number of samples is far more, giving a more conclusive result.

Since the reclassification procedure is not neccesarily called each generation, the system has the potential to be faster.

A potential drawback of SDRS as opposed to DRS is that the fitness of a program might be adversely affected by other programs, as classifications change according to the results of the (possibly unfit) majority.

4.2 Results Comparing Classification Strategies

This section presents the results of CDRS and SDRS on the five object classification datasets, compared to those of SRS. For each case, 10 runs of experiments were performed and the average results are presented.

4.2.1 Discussion of Results on Shape Databases

Table 4.3 shows the object classification results for shape databases.

Table 4.3 shows the results of experiments with the classification strategies applied to the shape databases. Table 4.4 shows the results using the coin databases.

In the table "Average final generation" refers to the average number of generations required by each of the ten runs, which was stopped either at 50 generations or when some program classified the training set perfectly (accuracy of 100%).

"Test set accuracy at best Validation" refers to the average test set accuracy at the highest validation set accuracy. The validation accuracy was measured each generation. If the validation set accuracy that is measured is the highest seen, the test set accuracy is also measured, overwriting the previous value. After the run the stored test set accuracy therefore

Table 4.3: Comparison of classification strategies on shape databases.

Image database	Number of classes	Classification strategy	Weight ratio	Average final generation	Test set accuracy at best Validation
Shape1	4	SRS		9.2	99.90%
		CDRS	1	17.0	99.69%
			3	15.9	99.25%
			5	11.6	99.58%
		SDRS	1	35.0	98.59%
			3	33.1	99.53%
			5	33.5	99.16%
Shape2	4	SRS		50.0	96.87%
		CDRS	1	26.5	99.67%
			3	23.9	99.39%
			5	15.5	99.72%
		SDRS	1	43.5	98.46%
			3	28.1	99.35%
			5	24.3	99.25%

contains the test set accuracy at the best validation set accuracy. This measure was used to avoid overfitting, as the validation set accuracy will start to wane when overfitting occurs, so the test set accuracy is measured at the peak.

For CDRS and SDRS techniques, the routine to update the boundaries or slots was called after each five generations.

The main point seen here is that the SRS procedure performs well when the classes are in their natural order (Shape1 database has the classes ordered by intensity, which is one of the features) but not nearly so well when the classes are out of this natural order (as in Shape2). The reason for this difference in performance is that a high degree of non-linearity is required to map the class regions on the real number line to the object features if two or more classes are in different orders. However if the classes are in their natural order, according to some main feature, SRS performs well. One would also expect SRS to perform well on two-class problems.

CDRS and SDRS perform slightly less well than SRS on Shape1, but show their advantage of being able to change the order of classes when applied to database Shape2. Results for CDRS and SDRS for Shape2 are very similar to those for Shape1. This suggests that both CDRS and SDRS can cope with Shape2 better than SRS can, since they can dynamically change the order and position of class regions (SRS is unable to do this).

CDRS had better results than SDRS, with similar accuracies but faster training.

For both CDRS and SDRS, the training was faster as the weight ratios was raised, though for SDRS the resulting accuracy dropped slightly as the weight ratio was raised for shape2.

Figure 4.4(a) shows a comparison of the classification strategies applied to the Shape2 problem. The x-axis is the generation number. The y-axis is the best test set accuracy seen up to the generation, averaged over 10 runs. A weight ratio of 3 was used.

As can be seen from figure 4.3 the figure it is easily seen that CDRS and SDRS are more suited to the problem than SRS. The two dynamic methods not only produced much better results than SRS, but also converged much faster. SRS moves toward the goal of an accuracy of 1.0 far slower than the other methods. SDRS is seen to become slightly more accurate than CDRS on this problem after the initial climb.

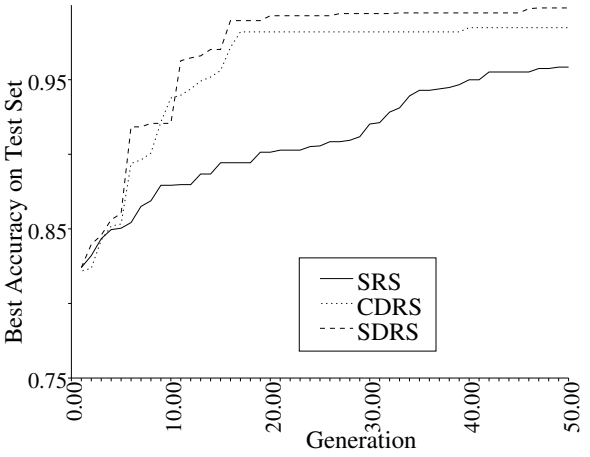


Figure 4.3: A comparison of classification strategy applied to the Shape2 dataset, best test accuracies reached vs. generation, accuracies are averaged over 10 runs.

4.2.2 Discussion of Results on Coin Databases

Table 4.4 shows the object classification results for coin databases.

These coin problems are harder than the shape problems. The shape datasets may be classified using one feature (the average intensity), this is not possible with the coin datasets. Due to this difficulty, it is seen that most runs of the 5-class coin problems did not achieve perfect accuracy on the training set before 50 generations.

Accuracy results for the 3-class problem (Coin1) were similar for the different classification strategies, but SDRS again was seen to be slower than the other two. CDRS was found to be slower than SRS. This problem, being of only a few classes, was found to be suited to SRS.

The harder 5-class problems (Coin2 and Coin3) were seen to be suited to CDRS or SDRS above SRS. SDRS was seen to perform slightly better than CDRS, in terms of final accuracy.

Of those tried, the weight ratio for CDRS is seen to be best at about 3 for these problems. The weight ratio for SDRS is good at 1, with this being the best result on the hardest dataset (Coin3).

Figure 4.4(b) shows a comparison of the classification strategies applied to the Coin2 problem. The x-axis is the generation number. The y-axis is the best test set accuracy seen up to the generation, averaged over 10 runs. A weight ratio of 3 was used.

In the figure it is seen that SDRS produces better results after only a few generations, but CDRS slowly improves, finally outperforming SDRS in accuracy. SRS is seen to improve more slowly after 10 generations, not quite reaching the level of the other methods.

4.3 Chapter Conclusions

As expected, the new classification strategies, CDRS and SDRS, were seen to outperform the standard SRS on the harder problems (Shape2, Coin2 and Coin3).

SRS was seen to perform well on the easy Shape1 and Coin1 problem, but showed limitations when dealing with datasets that were either out of order or had many classes.

SDRS defined a very flexible classification strategy, able to form very complex mappings from program output to class value.

CDRS has a more restricted range of expression. CDRS was designed as a method that

Table 4.4: Comparison of classification strategies on coin databases.

Image database	Number of classes	Classification strategy	Weight ratio	Average final generation	Test set accuracy at best Validation
Coin1	3	SRS		4.0	99.74%
		CDRS	1	6.1	99.69%
			3	8.9	99.95%
			5	6.6	99.69%
		SDRS	1	11.7	99.48%
			3	11.8	99.69%
			5	12.0	100.00%
Coin2	5	SRS		50	82.94%
		CDRS	1	48.4	85.56%
			3	48.6	90.75%
			5	50.0	88.54%
		SDRS	1	48.5	89.44%
			3	47.4	87.81%
			5	50.0	93.19%
Coin3	5	SRS		50.0	72.78%
		CDRS	1	50	76.48%
			3	50	78.00%
			5	50	77.33%
		SDRS	1	49.5	84.50%
			3	47.3	75.83%
			5	46.5	75.83%

allows the thresholds in SRS to move, and change in order. Results prove this to be a useful idea, and prompt further research.

The next chapter investigates another research goal, whether gradient-descent search can be applied to individuals in GP, for object classification problems.

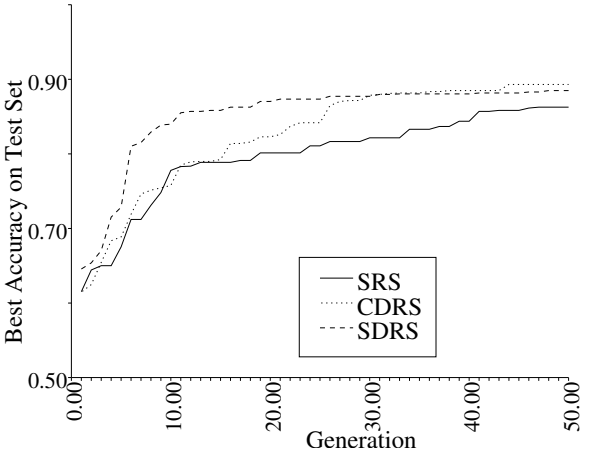


Figure 4.4: A comparison of classification strategy applied to the Coin2 dataset, best test accuracies reached vs. generation, accuracies are averaged over 10 runs.

Chapter 5

Gradient-Descent in Genetic Programming

In this chapter a new adaption of a highly established search method, gradient-descent, to GP is introduced. This method has been found to markedly improve the speed and accuracy of the GP system when used in tests, see section 5.4.

This research represents a new area to apply this method to. Gradient-descent is used widely in areas of machine learning such as Neural Networks (NN's), However this method has not been applied to GP in the way presented here, making this very exciting research.

Gradient-descent defines a group of methods that allow the minimum of a surface to be found, based on a small number of samples. The surface is differentiated, then the resultant gradient vector 'points' the way to go in order to move closer to the minimum of the surface like a ball rolling down a slope.

5.1 How Gradient-Descent Fits into GP?

The application of this method to GP involves the view of each evolved program as a separate learning system similar to a neural network (NN). This is feasible as each program has some parameters which, if altered, may make the program better at its goal of a low error rate (or a higher accuracy).

In NNs the parameters that are altered are the weights of the links attaching one node to another. See figure 5.1 for a diagram of a simple NN and a simple GP program. In a GP program, such as that of figure 5.1(a) the parameters that are analogous to the weights of the NN are the numeric terminals.

The reason for this analog is as follows: Both weights and numeric terminals affect the outcome of the NN or GP program. The cost of the system is differentiable by the weight or terminal value, allowing for gradient-descent. The value of both are constant when learning is not used, that is, they are not changed between input patterns.

This change of perspective allows the normal gradient-descent techniques to be applied to the individual programs, quite aside from the GP process. It is important to note that this application of the algorithm to the individual programs does not affect the normal GP process. The normal GP operators, such as crossover, mutation and reproduction, and all the selection operators, etc.. do not get replaced or even affected. Instead, the gradient-descent search algorithm is embedded into the GP beam search and evolutionary process.

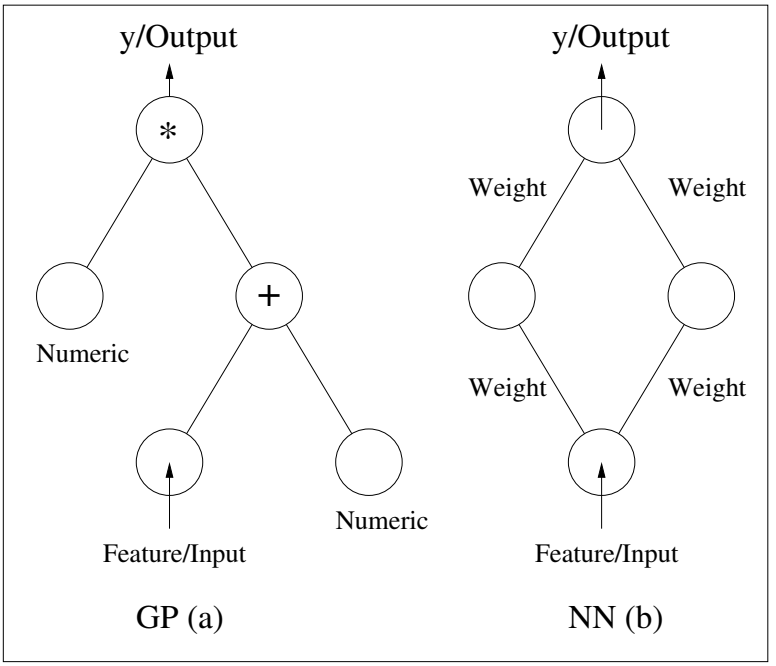


Figure 5.1: A GP program (a), and an NN (b).

5.2 Gradient-Descent Search

5.2.1 The Model of a Learning System

Any machine-learning system can be reduced to an instance of a general model.

Fundamentally the system performs some *task* on some set of data.

The measure of performance of a system may be some numerical distinction between the desired outputs (Y) and computed outputs (y), forming an error, such as $(Y-y)^2$, taken over some data (eg. summed). Such an error is a continuous measure of the system's performance at the task, on the data, which is referred to here as the cost C of the system.

The system uses some *parameters*. The parameters are the values stored in the system that affect how it performs the task.

Parameters could include, for instance, weights and of a neural net, or numeric terminal values of a GP program.

It is these parameters that are most important for the view of learning in a system presented here, as they can be changed during evolution, continuously affecting the cost of the learning system. This allows the gradient-descent method to be used.

The aim then can be more clearly defined as trying to minimize C by changing these parameters. In this way the cost C can be seen as a surface over the parameters. Thus the aim of learning is the search for a point (representing the parameters' values) on the surface that minimizes C.

Note that the cost surface may correspond to a single NN, program, or other learning system.

There are some assumptions about the surface model that are essential in choosing gradient-descent:

- Is C differentiable with respect to the parameters we are trying to change? In the cases of NN and GP the cost surface is easily differentiable with respect to the weights and numeric terminals respectively for the most obvious cost functions.

- Are the samples so cheap that the entire surface may be checked for the minimum of C? With NN's and GP the samples are cheap, but the surface may be of a very high dimensionality, making a brute force search infeasible for all but the smallest networks or programs.
- Is the C surface continuous or very noisy? With NN's or GP the cost surface will typically be quite continuous.

5.2.2 Basic Idea of Gradient-descent

Gradient-descent is based on the principle that, on real world surfaces, the best way to find the minimum of a surface C is to determine the gradient of the point where you are, and head downhill. This would seem intuitive, probably because it will work for most earthly landscapes that are encountered, to head to the bottom, go downhill.

If, however, we assume that the surface is fairly smooth, such as the earthly landscape or C, gradient-descent will work fairly well.

The gradient of a surface C of n dimensions (i.e. with n parameters to be changed) at a point $\mathbf{p}$ can be found by differentiating the surface at the point. This gives a vector $\mathbf{v}_{up}$ in n dimensions by equation 5.1, pointing as a tangent to the surface in the steepest direction, and of length proportional to this steepness. This gradient vector can be seen as pointing along the surface and towards the *uphill* direction. To go *down* the surface the opposite vector ($\mathbf{v}_{down} = -\mathbf{v}_{up}$) is chosen.

The parameters are each changed in a way proportional to the values in $\mathbf{v}_{down}$ as shown in equations 5.2 and 5.3.

$$\begin{aligned} \mathbf{v}_{up} &= \frac{\partial C}{\partial \mathbf{p}} && (5.1) \\ \mathbf{v}_{down} &= -\mathbf{v}_{up} \\ \Delta p_i &= \alpha \cdot v_{down,i} && (5.2) \\ \mathbf{p}' &= \mathbf{p} + \Delta \mathbf{p} && (5.3) \end{aligned}$$

Where α is a rate parameter that determines how far to move relative to the slope of the surface at the point $\mathbf{p}$.

The new point $\mathbf{p}'$ should be lower on C than $\mathbf{p}$ for a sufficiently small rate, but may not for a larger rate if the calculation overshoots a bowl or valley on the surface. The rate is normally an *a priori* knowledge required by the user when configuring the system. In the domain of GP programs however an *optimal* rate can be calculated for many programs, see section 5.2.4.

One disadvantage of the gradient-descent algorithm is that the point $\mathbf{p}$ may get 'stuck' in a local minima, like a bowl on the surface. The method still works well, though, and in the case of GP the program will be better than how it started anyway, since the genetic beam search will play a role to improve the "local minima" situation.

5.2.3 The Chain Rule

The main support for gradient-descent in a layered network is the chain rule, that is:

$$\frac{\partial a}{\partial c} = \frac{\partial a}{\partial b} \cdot \frac{\partial b}{\partial c}$$

If a depends on b and b depends on c.

This rule allows a complex derivative to be broken down into several simple ones, so long as some natural break point exists. For example:

$$\begin{aligned}
& \frac{\partial \log(\sin x^2)}{\partial x} \\
= \quad & \frac{\partial \log(\sin v)}{\partial v} \cdot \frac{\partial v}{\partial x} \\
= \quad & \frac{\partial \log u}{\partial u} \cdot \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial x} \\
= \quad & \frac{\partial \log u}{\partial u} \cdot \frac{\partial sin(v)}{\partial v} \cdot \frac{\partial x^2}{\partial x} \\
= \quad & \frac{1}{u} \cdot \cos v \cdot 2x \\
= \quad & \frac{2x \cos(x^2)}{\sin x^2}
\end{aligned}$$

where $v = x^2, u = \sin x^2$.

This rule is useful in layered networks such as NN's or GP programs, as they are made of many functionally separate operators that feed to the layer above.

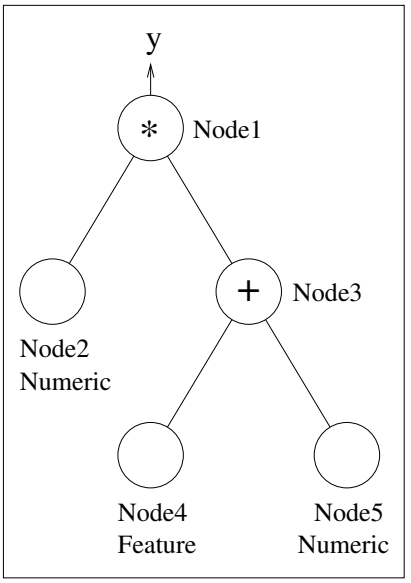


Figure 5.2: An example genetic program.

Referring to the simple program tree in figure 5.2 and defining f_i as the operation performed by the function at node i, and O_i as the value of terminal node i, or output of function node i. The output y is:

$$y = f_1(O_2, f_3(O_4, O_5))$$

so the derivatives of y with respect to the numeric terminals are:

$$\begin{aligned}
\frac{\partial y}{\partial O_2} \quad &= \quad \frac{\partial f_1(O_2, f_3(O_4, O_5))}{\partial O_2} \\
&= \quad \frac{\partial (O_2 \cdot f_3(O_4, O_5))}{\partial O_2}
\end{aligned}$$

$$
\begin{aligned}
&= f_3(O_4, O_5) \\
&= O_3 \\
\frac{\partial y}{\partial O_5} &= \frac{\partial f_1(O_2, f_3(O_4, O_5))}{\partial O_5} \\
&= \frac{\partial f_1(O_2, x)}{\partial x} \cdot \frac{\partial f_3(O_4, O_5)}{\partial O_5} \\
&= \frac{\partial (O_2 \cdot x)}{\partial x} \cdot \frac{\partial (O_4 + O_5)}{\partial O_5} \\
&= O_2 \cdot 1 \\
&= O_2
\end{aligned}
\tag{5.4}
$$

Now, we can gain the gradient vector of C as containing:

$$
\begin{aligned}
\frac{\partial C}{\partial O_2} &= \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_2}, \\
\frac{\partial C}{\partial O_5} &= \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_5},
\end{aligned}
\tag{5.5}
$$

Implying that once the cost surface C has been derived with respect to the program output y, we will have the gradient we need.

Doing products with the chain rule, similar to this example, gives an easy way to find the derivative of C with a program of any depth, with respect to any node.

5.2.4 Scheme 1: Online Learning in GP

Online learning refers to the method where the surface descended refers to a single input pattern. That is that the parameters are changed for each input pattern.

The cost surface C we want to descend is the squared error of the system on a datum μ. More formally $C^\mu = (Y^\mu - y^\mu)^2 \div 2$, where Y^μ is the desired output of the system, and y^μ is the calculated output.

For readability the μ is not included in subsequent equations.

If we use SRS (Static Range Selection) or CDRS (Centred Dynamic Range Selection) for the classification strategy, the Y can be the centre of the region that will be classified as the correct class. If we use SDRS (Slotted Dynamic Range Selection) for the classification strategy, the Y can be the centre of the closest region that will be classified as the correct class.

In order to perform gradient-descent on C, we find the gradient, i.e. differentiate with respect to the values of the numeric terminals.

For example: figure 5.3 shows a four level program, with two numeric terminals:

The differentials we need are:

$$
\begin{aligned}
\frac{\partial \mathbf{C}}{\partial O_2} &= \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_2} \\
\frac{\partial \mathbf{C}}{\partial O_6} &= \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_6} \\
\frac{\partial C}{\partial y} &= \frac{\partial \frac{(Y-y)^2}{2}}{\partial y} \\
&= Y - y
\end{aligned}
$$

The derivatives of y with respect to O_2, O_6 are found in a manner similar to that of the previous section, using a backward propagation algorithm. This is explained in section 5.3.

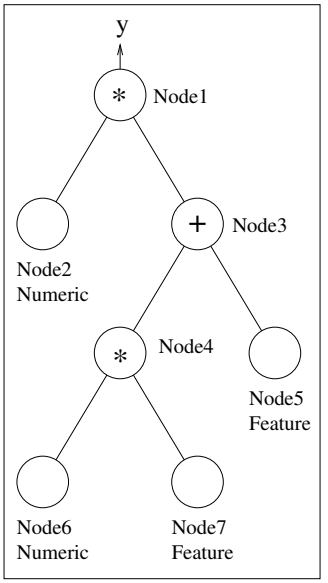


Figure 5.3: A deeper example genetic program.

An Advantage of Online Learning: α factor Calculation

Recall that the general gradient-descent update equation for a parameter O_i is:

$$O_i' = O_i - \alpha \cdot \frac{\partial C}{\partial O_i} \tag{5.6}$$

In this and later sections we will refer to α as 'factor α', or just α.

The advantage of online learning is that, if the program is *linear* (see below), the factor α can be calculated as in the equation:

$$\alpha = \frac{\eta}{\sum_i^N (\frac{\partial y}{\partial O_i})^2} \tag{5.7}$$

where N is the total number of numeric terminals in the program, and i indexes only the numeric terminals. η is a *learning rate* that is used to fine tune the system.

When this is chosen as α, $\eta = 1$ and online learning is used, the error of some programs can be reduced to zero for any pattern.

Intuitively, if all numeric terminals affect the output of the program individually and the change of output with each change in value is known, then the equation 5.7 will distribute changes through the numeric terminals, so that the output will change by a value required to get the output to change to equal the desired output.

Proof

This effect depends on the following aspects of the program:

- The program must have only linear operators performed on subtrees with numeric terminals, that is equation 5.8 is constant for all values O_i for all numeric terminals i.

$$\frac{\partial y}{\partial O_i} \tag{5.8}$$

  where y is the program output, O_i is the value of the numeric terminal i.

- The effect of changes to the numeric terminals on the program output be independent, that is, for any i, equation 5.8 is constant for all values (O_j) of all other ($j \neq i$) numeric terminals.

With $\eta = 1$ the proof of the error being set to zero is as follows:
First, the change in a single numeric terminal when gradient-descent is used is:

$$\begin{aligned} O'_i - O_i &= -\alpha \cdot \frac{\partial C}{\partial O_i} \\ &= -\alpha \cdot \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_i} \end{aligned}$$

so, if the operator is linear, the change in y due to a single numeric terminal is:

$$\begin{aligned} \Delta y_{O_i} &= (O'_i - O_i) \cdot \frac{\partial y}{\partial O_i} \\ &= -\alpha \cdot \frac{\partial C}{\partial y} \cdot \frac{\partial y}{\partial O_i} \cdot \frac{\partial y}{\partial O_i} \\ &= -\frac{\frac{\partial C}{\partial y} \cdot (\frac{\partial y}{\partial O_i})^2}{\sum_j^N (\frac{\partial y}{\partial O_j})^2} \\ &= -\frac{\partial C}{\partial y} \cdot \frac{(\frac{\partial y}{\partial O_i})^2}{\sum_j^N (\frac{\partial y}{\partial O_j})^2} \end{aligned}$$

so, if the effect on y by each O_i is independent, as in the assumed situation, the change in y due to all the numeric terminals is:

$$\begin{aligned} \Delta y &= \sum_i^N \Delta y_{O_i} \\ &= \sum_i^N -\frac{\partial C}{\partial y} \cdot \frac{(\frac{\partial y}{\partial O_i})^2}{\sum_j^N (\frac{\partial y}{\partial O_j})^2} \\ &= -\frac{\partial C}{\partial y} \cdot \sum_i^N \frac{(\frac{\partial y}{\partial O_i})^2}{\sum_j^N (\frac{\partial y}{\partial O_j})^2} \\ &= -\frac{\partial C}{\partial y} \cdot \frac{\sum_i^N (\frac{\partial y}{\partial O_i})^2}{\sum_j^N (\frac{\partial y}{\partial O_j})^2} \\ &= -\frac{\partial C}{\partial y} \\ &= -(y - Y) \\ y' &= y + \Delta y \\ &= Y \end{aligned}$$

proving that the new output of the program, y', is equal to the desired output Y when the α in equation 5.7 is used as in equation 5.6 and $\eta = 1$.

Note that the assumptions made about the programs are fairly restrictive. Programs are not included if they contain any divides by subtrees containing a numeric terminal, or two subtrees both containing numeric terminals multiplied together.

Tests will be carried out both using equation 5.7, and just setting $\alpha = \eta$, these options are referred to as 'summed factor' and 'simple factor' respectively.

5.2.5 Scheme 2: Offline Learning in GP

Offline learning refers to the method where the surface ascended/descended refers to all input patterns. That is that the parameters are changed for the whole training set.

The cost surface C we want to descend is a sum of costs over all patterns. More formally $C = \sum_\mu (Y^\mu - y^\mu)^2 \div 2$, where Y^μ is the desired output of the system on pattern μ, and y^μ is the calculated output with pattern μ.

In order to perform gradient-descent on C, we find the gradient, i.e. differentiate with respect to the values of the numeric terminals.

For example: with figure 5.3 as in online learning:

The differentials we need are:

$$\begin{aligned} \frac{\partial \mathbf{C}}{\partial O_2} &= \sum_\mu \frac{\partial C^\mu}{\partial y^\mu} \cdot \frac{\partial y^\mu}{\partial O_2} \\ \frac{\partial \mathbf{C}}{\partial O_6} &= \sum_\mu \frac{\partial C^\mu}{\partial y^\mu} \cdot \frac{\partial y^\mu}{\partial O_6} \end{aligned}$$

Thus offline learning is performed in a way similar to online learning, but summing the derivatives that would be used after each pattern, until all patterns are seen. The numeric terminals are then changed proportionally to the sum.

5.3 The Gradient-Descent Algorithm in GP

In this section, we also used the example of figure 5.3 to describe the gradient-descent algorithm.

The derivatives of y (y^μ) with respect to O_2, O_6 are found using an efficient recursive algorithm.

These are multiplied by the derivative of the cost C by the program output y, giving the components of the gradient vector, and a way to change the parameters leading to a lower cost.

5.3.1 Some Preliminaries

The only values that are required to find the gradient are known output values which can be obtained by evaluating the program tree, and storing the output value of each node in the node's space.

We will adopt the notation of:

- O_i being the stored output of node i *evaluated using the current pattern*, i.e. there is a latent μ.
- $f_i(\mathbf{args}..)$ being the mathematical operation of function node i.

The only functions derived are those in the function set: multiplication, addition, division, subtraction and `if`. These derivatives are listed in table 5.1.

5.3.2 The Online Algorithm

The patterns first have their order randomized, so as to prevent the same class from training the program toward itself, in independence of other classes. Without this random order the programs were seen to overtrain often, leading to bad generalization.

Table 5.1: Operator derivatives for online learning.

Operator	$\frac{\partial f(\mathbf{args})}{\partial a_1}$	$\frac{\partial f(\mathbf{args})}{\partial a_2}$	$\frac{\partial f(\mathbf{args})}{\partial a_3}$
$a_1 + a_2$	1	1	n/a
$a_1 - a_2$	1	-1	n/a
$a_1 \times a_2$	a_2	a_1	n/a
a_1/a_2	a_2^{-1}	$-a_1 \times a_2^{-2}$	n/a
if $a_1 < 0$ then a_2 else a_3	0 0	1 if $a_1 < 0$ 0 if $a_1 \geq 0$	0 if $a_1 < 0$ 1 if $a_1 \geq 0$

Another option would be to sequentially step from one class to another while moving through the patterns. This method, however, would be difficult if the numbers of objects of each class in the dataset were different.

The online learning algorithm implemented has the following steps, performed for each program, on each pattern in the training set:

Step 1 Evaluate the program, and store the value evaluated as the output of each node.

Step 2 Move from the root node towards the leaves (used a depth-first-search) performing the chain rule. Store the value of the derivative of the output y^μ with respect to the numeric terminals in the program. This step also allows accumulation of the magnitude of all such derivatives by return values. See section 5.3.4 for details of this step.

$$r = \alpha \cdot (Y^\mu - y^\mu) \tag{5.9}$$

$$\tag{5.10}$$

Step 3 Determine a factor r based on equation 5.11 or 5.12 depending on a algorithm setting.

$$\alpha = \frac{\eta}{\sum_i^N (\frac{\partial y}{\partial O_i})^2} \tag{5.11}$$

$$\alpha = \eta \tag{5.12}$$

$$\tag{5.13}$$

Step 4 Distribute this factor r amongst the numeric terminals using a depth-first-search. Each will then change its value by an amount as in equation 5.14.

$$O_i'^\mu = O_i^\mu + r \cdot \frac{\partial y^\mu}{\partial O_i^\mu} \tag{5.14}$$

This algorithm is summarized as follows.

```
function DoOnline
  for programs p
    for training patterns m
```

```
      Evaluate p using m
      totderiv = p.root.PropError( 1 )
      Calculate r
      p.root.AlterNumerics( r, 1 )
function end
```

The functions `PropError` and `AlterNumerics` are described in sections 5.3.4 and 5.3.5 respectively.

"`Calculate r`" uses equations 5.9, and either 5.11 or 5.12 depending on a setting.

5.3.3 The Offline Algorithm

The offline learning algorithm implemented has the following steps, performed for each program:

Step 1 For each pattern:

Step 1.1 Evaluate the program using the pattern, and store the value evaluated as the output of each node.

Step 1.2 Move from the root node towards the leaves (used a depth-first-search) and obtaining the value of the derivative of the output y^μ with respect to the numeric terminals in the program. See section 5.3.4 for details of this step.

Step 1.2.1 add the derivative times the factor r to an average **avgderiv** as in equation 5.15. r is found using 5.12 and 5.9.

$$\mathbf{avgderiv}'_i = \frac{\mathbf{avgderiv}_i \times (n-1) + r^\mu \times \frac{\partial y^\mu}{\partial O_i^\mu}}{n} \tag{5.15}$$

$$\tag{5.16}$$

where n is the number of the pattern being seen, i.e. for the j'th pattern $n = j$.

Step 2 Update the value of each numeric terminal by the average **avgderiv** stored in it, using equation 5.17.

$$O'_i = O_i + \mathbf{avgderiv} \tag{5.17}$$

This algorithm is summarized as follows.

```
function DoOffline
  for programs p
    Clear all numeric terminal numdydo's
    for training patterns m
      Evaluate p using m
      totderiv = p.root.PropError( 1 )
      Calculate r
      p.root.AlterNumerics( r, 0 )
    p.root.AlterNumerics( 0, 1 )
function end
```

The functions `PropError` and `AlterNumerics` are described in sections 5.3.4 and 5.3.5 respectively.

"`Clear all numeric terminal numdydo's`" performs a depth-first-search routine that clears the `numdydo` values stored in each numeric terminal to zero.

"`Calculate r`" uses equation 5.12.

5.3.4 Finding Derivative of Output with respect to Numeric Terminals

Here we discuss the algorithm to find the derivatives:

$$\frac{\partial y^{\mu}}{\partial O_i^{\mu}}$$

for numeric terminal i.

The algorithm utilizes a standard depth-first-search and efficiently moves through the program to calculate the derivatives.

The function for a function node with two arguments is as follows:

```
function operator2.PropError( dydo )
  ret = arg1.PropError( dydo * dodarg1( arg1.output, arg2.output ) )
    + arg2.PropError( dydo * dodarg2( arg1.output, arg2.output ) )
  return(ret)
function end
```

A three argument function, such as **if** is similar, but with another argument:

```
function operator3.PropError( dydo )
  ret=arg1.PropError(dydo*dodarg1(arg1.output,arg2.output,arg3.output))
    +arg2.PropError(dydo*dodarg2(arg1.output,arg2.output,arg3.output))
    +arg3.PropError(dydo*dodarg3(arg1.output,arg2.output,arg3.output))
  return(ret)
function end
```

The procedure for a numeric terminal is as follows:

```
function numeric.PropError( dydo )
  store dydo
  return dydo*dydo
function end
```

`dydo` is a variable within the numeric terminal.

Other forms of terminals simply return 0 in `PropError`.

The program is traversed by calling `PropError` on the root node with an argument of 1.

The functions `dodarg1, dodarg2, dodarg3` vary for different functions. They are the derivatives:

$$\begin{aligned} \textbf{dodarg1} &= \frac{\partial f(a_1, a_2, a_3)}{\partial a_1} \\ \textbf{dodarg2} &= \frac{\partial f(a_1, a_2, a_3)}{\partial a_2} \\ \textbf{dodarg3} &= \frac{\partial f(a_1, a_2, a_3)}{\partial a_3} \end{aligned} \tag{5.18}$$

Where f is the operation performed in the function, and a_1, a_2, a_3 are the arguments to the function. If the function only takes two arguments the third is simply ignored (see table 5.1 for a list of the derivatives for different functions used).

After the call the values of `dydo` stored in the i'th node is equal to $\partial y / \partial O_i$, if it is a numeric terminal.

5.3.5 Altering the Numeric Values

Here we discuss the algorithm to change the values of the numeric terminals using a factor `r`, and the previously stored derivative. The algorithm is used both for online and offline learning, through use of a parameter `doit`, which allows averaging to take place in the case of offline learning.

The algorithm utilizes a standard depth-first-search an efficiently move through the program to alter the numeric terminals.

The procedure for a function node is as follows:

```
function operator2.AlterNumerics( r, doit )
  arg1.AlterNumerics( r, doit )
  arg2.AlterNumerics( r, doit )
function end
```

A three argument function, such as **if** is similar, but with another argument call. The procedure for a numeric terminal is as follows:

```
function numeric.AlterNumerics( r, doit )
  avgdydo = ( avgdydo * numdydo + r * dydo ) / ( numdydo )
  numdydo = numdydo + 1
  if doit
    value = value + avgdydo
    numdydo = 0
function end
```

`numdydo` and `avgdydo` are variables within the numeric terminal, initially zero. `dydo` is also a variable, calculated in a previous step. `value` is the value of the numeric terminal.

Other forms of terminals simply ignore the call to `AlterNumerics`.

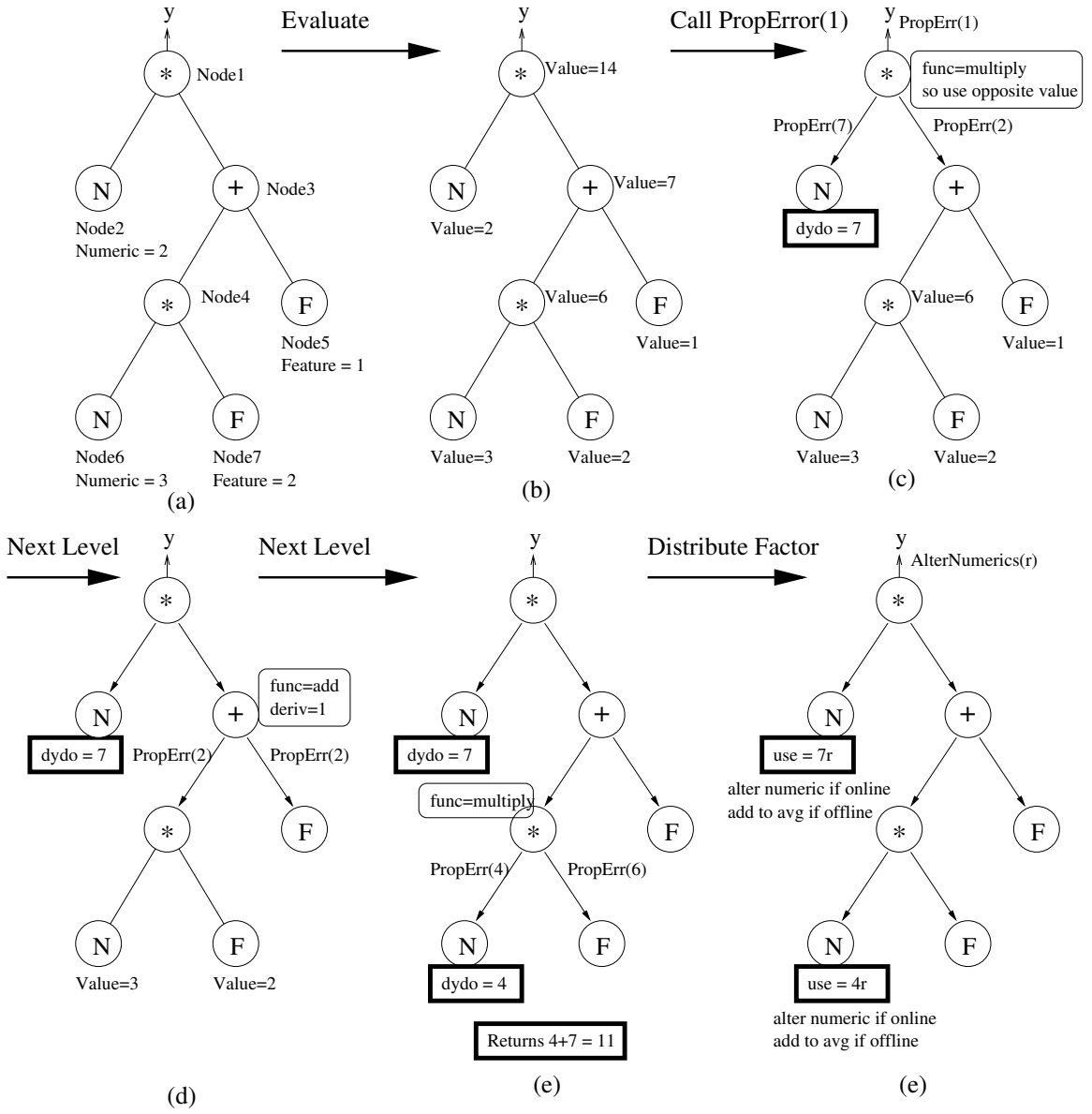


Figure 5.4: Steps to calculate derivatives of numeric terminals

5.3.6 An Example

Figure 5.4 shows the process of section 5.3.4 applied to the example program tree.
In the figure the steps are:

- (a) The program, with values of terminals for the current pattern.
- (b) The evaluated program with results of all nodes.
- (c) The first level of derivatives $\partial y/\partial O_{node}$. The multiplication operator derives to the value of the opposite argument.
- (d) The second level of derivatives. The chain rule is applied, and the addition operator derives to 1.
- (e) The third level of derivatives. Again the chain rule is applied.

- (f) The factor `r` is distributed to the numeric terminals, and they either use it, or add it to an average.

5.4 Results and Discussion

Table 5.2 shows the results of the routine with various rates α, using three variations of the algorithm.

Table 5.2: Comparison of η learning rates, and learning methods, Averages over 10 runs.

Image database	η	**Average final generation**			**Test set accuracy (%) at best Validation**		
		Online		**Offline**	**Online**		**Offline**
		summed factor	**simple factor**	**twice per gen**	**summed factor**	**simple factor**	**twice per gen**
Shape1	0.0	9.56	9.56	9.56	99.48	99.48	99.48
	0.2	1.00	1.00	3.20	100.00	100.00	99.77
	0.4	1.00	1.00	2.20	100.00	100.00	99.86
	0.7	1.00	1.00	2.80	100.00	100.00	99.81
	1.0	1.00	1.40	3.30	100.00	99.91	99.63
	1.4	1.00	2.20	5.50	99.95	99.95	99.77
Shape2	0.0	43.80	43.80	43.80	96.68	96.68	96.68
	0.2	11.30	16.10	46.70	99.53	99.53	96.31
	0.4	9.80	36.10	31.60	99.67	98.88	98.13
	0.7	15.00	33.00	33.80	99.63	98.79	97.71
	1.0	13.60	40.80	28.80	99.63	98.41	98.88
	1.4	36.80	38.90	38.60	99.30	98.74	97.80
Coin1	0.0	8.00	8.00	8.00	99.53	99.53	99.53
	0.2	1.00	1.00	6.20	99.95	100.00	99.43
	0.4	1.00	1.00	4.00	99.84	99.84	100.00
	0.7	1.00	1.00	2.20	99.79	99.84	99.32
	1.0	1.00	1.20	2.10	99.95	99.95	99.90
	1.4	1.00	1.30	1.60	99.95	99.95	99.69
Coin2	0.0	51.00	51.00	51.00	82.12	82.12	82.12
	0.2	20.40	38.70	35.70	98.94	97.00	94.19
	0.4	25.40	29.80	43.40	98.94	98.31	93.06
	0.7	23.30	44.10	46.60	98.81	97.38	91.81
	1.0	36.70	40.10	47.30	97.69	95.38	92.44
	1.4	34.00	44.40	37.50	97.94	92.62	94.38
Coin3	0.0	51.00	51.00	51.00	73.83	73.83	73.83
	0.2	51.00	51.00	51.00	83.50	81.67	72.17
	0.4	51.00	51.00	51.00	82.83	84.33	77.67
	0.7	50.20	51.00	51.00	85.17	84.67	78.83
	1.0	51.00	51.00	51.00	86.50	81.67	84.17
	1.4	51.00	51.00	51.00	80.83	80.17	79.00

From the table it is clear that the use of a *summed factor* equation generally gives better results than use of the *simple factor*, though both are good. On easy problems (Shape1, Coin1) the online gradient-descent (using summed factor) will *always* produce a program that solves the training problem (100% training set accuracy) after the first generation. Note that these results are averaged over 10 runs, making a total of 100 runs that gave this result (5 rates on the Shape1 dataset, 5 on the Coin1 dataset, 10 runs per rate). Without the

gradient-descent routine, seen in the table as a rate of 0.0, the results are worse, taking 9.56 generations to solve the training problem (Shape1). The final accuracy on the test set is also far better for the gradient-descent runs where the summed factor is used on easy problems, implying good generalization despite the very short run time.

On the harder problem of Coin2 gradient descent shortened the run time. Without gradient-descent the training problem is not solved in any of the ten runs (average generation of 51). With gradient-descent (summed factor) the mean final generation is about 30 for all rates. Because of the completion of the problem the test set accuracy when using gradient-descent is better than without, averaging about 95% instead of 82%.

On the much harder Coin3 problem, test set accuracy is also found to be much better when gradient-descent (using summed factor) is used than when no gradient-descent is used.

In most cases the test set accuracy achieved using the simple factor equation for α is similar to that achieved using the summed factor, learning is slower however.

Offline learning gives results that are in general better than those that use no gradient-descent. However the offline learning results are typically worse than those of online learning. This probably comes down to the small number of times the routine can be run (on each program per generation) without causing a long time per generation. The offline learning results here are for two times per generation. Compare this with online learning which may update the weights of each program 200 times in a generation.

The online gradient-descent routine added little to the time-per-generation, and more than paid for itself by speeding evolution.

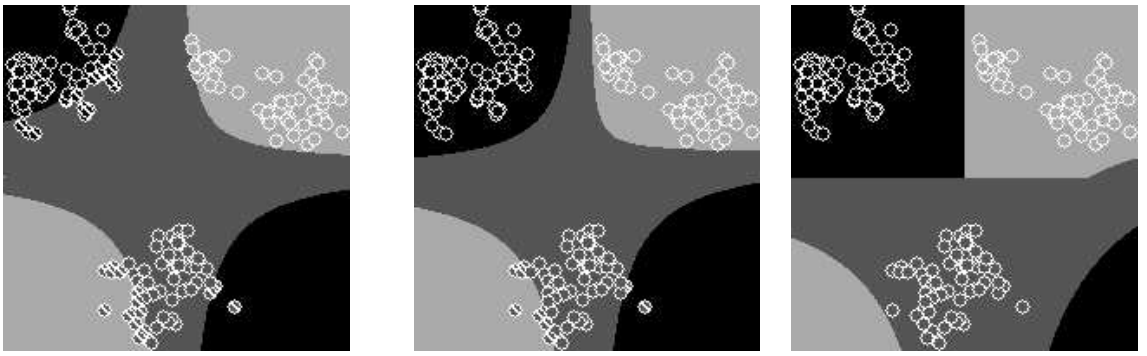
Table 5.3 shows the results of offline learning varying numbers of times per generation, with $\alpha = 1.0$.

Table 5.3: Comparison of numbers of times offline learning algorithm is call per generation. Learning rate ($\alpha = 1.0$). Averages over 10 runs.

Image database	Times routine called per generation	Average final generation	Test set accuracy at best Validation
Shape1	1	3.89	99.90
	2	3.30	99.63
	5	2.00	99.74
Shape2	1	31.80	97.76
	2	28.80	98.88
	5	30.10	98.93
Coin1	1	3.20	99.84
	2	2.10	99.90
	5	1.30	99.90
Coin2	1	48.10	88.25
	2	47.30	92.44
	5	35.30	94.88
Coin3	1	49.90	81.00
	2	51.00	84.17
	5	51.00	81.67

The average number of generations to solve the training problem is reduced slightly as the routine is used more. The accuracy is generally raised for running the routine more.

Figure 5.5 shows some maps of the classes (intensity) returned by three programs. Only two features are used, one is the vertical, the other is the horizontal. The circles denote the points in the training set. the intensities inside the circles denote the class of the point, though this is obvious given the clear clustering of the points. The (a) image shows the original programs, (b,c) show improvements by running the gradient-descent algorithm on the program once with a learning rate of 0.4 (b) or 1.0 (c).



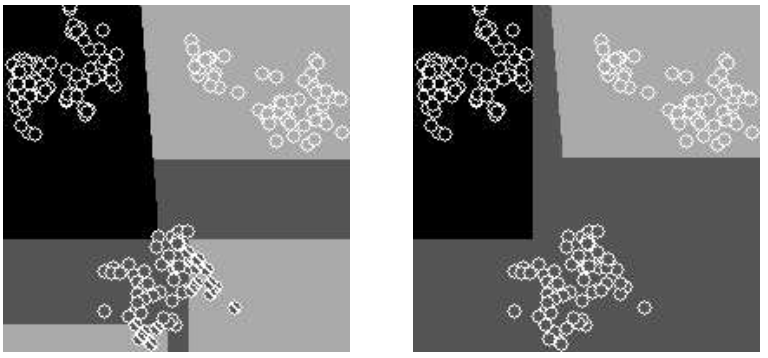
No gradient-descent, acc=82.2% (a)

$\eta = 0.4$, acc=95.3% (b)

$\eta = 1.0$, acc=100.0% (c)

Figure 5.5: Some images for a two feature system, mapping the output class (intensity) over the two features (vertical, horizontal) for one example program. Circles show the locations of points in the dataset (Coin1). Training set accuracy is shown below the figures. (a) is the original program, (b) has had online gradient-descent performed with a rate of 0.4. (c) has had online gradient-descent performed with a rate of 1.0.

Figure 5.6 shows maps of the classes returned by a different program, and the result of performing gradient-descent on it.



No gradient-descent, acc=90.1% (a)

$\eta = 0.4$, acc=100.0% (b)

Figure 5.6: Some images for a two feature system, mapping the output class (intensity) over the two features (vertical, horizontal) for one example program. Circles show the locations of points in the dataset (Coin1). Training set accuracy is shown below the figures. (a) is the original program, (b) has had online gradient-descent performed with a rate of 0.4.

It is seen that the borders between classes move due to the procedure, but the general shape of the program does not. The borders just move toward a state of lower cost. Note that these images are produced using online learning with the summed factor equation for

α being used.

The programs here are not typical. Several programs were checked before arriving at these two. However these are typical of those programs on which the gradient-descent works very well, and since we are looking for the best, the method works.

The original programs used are listed below:

```
Example 1:
(if<0 (d+ (d/ (d- F1 drand-0.102000) (d+ F2 drand-0.350000))
    (d- drand0.184000 (d+ drand-0.362000 drand0.626000)))
  (d/ (d- (d+ F2 drand-0.550000) (d* drand-0.766000 drand0.022000))
    (d/ (d/ drand-0.298000 drand-0.750000) (d- F2 drand0.186000)))
  (d/ (d- (d/ F1 drand0.958000) (d* drand-0.766000 drand0.022000))
    (d/ (if<0 drand0.526000 drand0.090000 drand-0.078000)
      (if<0 F1 drand-0.370000 drand-0.454000))))
With gradient-descent (rate=0.4) turns to:

(if<0 (d+ (d/ (d- F1 drand-0.102000) (d+ F2 drand-0.350000))
    (d- drand0.184000 (d+ drand-0.362000 drand0.626000)))
  (d/ (d- (d+ F2 drand-0.697772) (d* drand-0.757804 drand-0.089928))
    (d/ (d/ drand-0.693613 drand-0.532476) (d- F2 drand0.604766)))
  (d/ (d- (d/ F1 drand0.998450) (d* drand-0.749201 drand-0.117929))
    (d/ (if<0 drand0.526000 drand0.090000 drand-0.392933)
      (if<0 F1 drand-0.483379 drand-0.194117))))
With gradient-descent (rate=1.0) example 1 turns to:

(if<0 (d+ (d/ (d- F1 drand-0.102000) (d+ F2 drand-0.350000))
    (d- drand0.184000 (d+ drand-0.362000 drand0.626000))
  (d/ (d- (d+ F2 drand-0.762710) (d* drand-0.633993 drand-0.120593)
    (d/ (d/ drand-1.240308 drand-0.746827) (d- F2 drand0.647136)))
  (d/ (d- (d/ F1 drand0.780079) (d* drand-0.334165 drand-0.131830))
    (d/ (if<0 drand0.526000 drand0.090000 drand-0.788185)
      (if<0 F1 drand-1.067232 drand-0.040173))))

Example 2:
(d+ (d- (d* (d* F2 drand-1.651586) (d/ F1 drand0.525075))
    (d* (if<0 drand-0.894000 drand0.251203 drand0.378000)
      (if<0 F2 drand-0.172487 drand-0.604330)))
  (d/ (d+ (d+ F1 drand-0.502811) (if<0 F2 drand-0.591901 drand1.223090))
    (d* (d- drand-0.929284 drand1.737284)
      (if<0 F1 drand1.607206 drand1.384609))))

With gradient-descent (rate=0.4) turns to:
(d+ (d- (d* (d* F2 drand-2.194613) (d/ F1 drand0.713757))
    (d* (if<0 drand-0.894000 drand-0.651657 drand0.378000)
      (if<0 F2 drand-0.674684 drand-0.223879)))
  (d/ (d+ (d+ F1 drand-1.162297) (if<0 F2 drand-0.837411 drand0.809114))
    (d* (d- drand-1.376726 drand2.184726)
      (if<0 F1 drand2.647921 drand0.628708))))
```

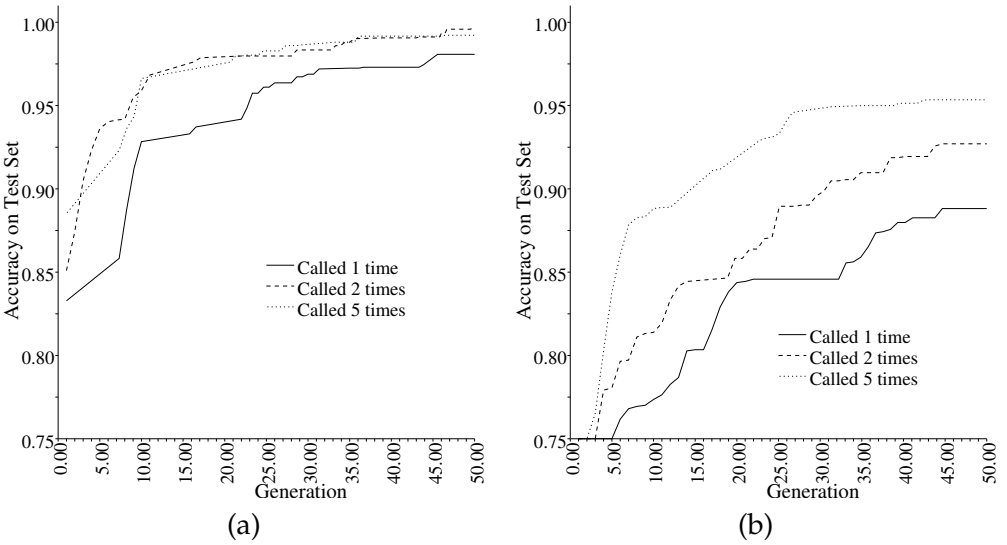


Figure 5.7: Test accuracy trend using offline learning varying numbers of times per generation. Avg over 10 runs. Dataset Shape2 (a), Coin2 (b).

Figure 5.7 shows the trend of test set accuracy with generation for calling the offline learning routine various numbers of times each generation.

The more times the routine is run per generation, the higher the accuracy trend. This echos the tabular results, and indicates that the offline learning procedure has untapped potential for these tasks.

Figure 5.8 shows the trend of test set accuracy with generation for various rates and routines on the Shape2 (a) and Coin2 (b) datasets.

A main feature is the high initial fitness when online gradient-descent is used. Some programs in the population will have good structure, but only random numeric terminal values. This emphasizes the reason that gradient-descent is a good idea when used in this way, to refine the values in order to make use of good program structure.

The use of online gradient descent using the summed factor equation, as in fig 5.8(a,b), is seen to give a much faster rise time than the other methods. Offline gradient-descent and online gradient-descent using the simple factor equation are comparable in performance.

5.5 Chapter Conclusions

The chapter describes an interesting new way to create a hybrid search for classifiers. The GP search is still performed as a global search, but a gradient-descent search is performed locally on individuals.

The use of this search produced excellent results, training times were shortened for easy problems, and test set accuracies were improved for harder problems.

Online (stochastic) learning gave better results than offline learning.

A calculation for the rate when using online learning was seen to improve results further.

This search is effective due to a property of GP. The numeric terminals in GP are not manipulated in order to find better values. Numeric terminals are changed only as all other nodes, and the structure, of programs are changed, by mutation and crossover. In essence, standard GP performs a search over *discrete* spaces. The application a search on the numeric terminals fill this gap, altering the numeric terminals *continuously*.

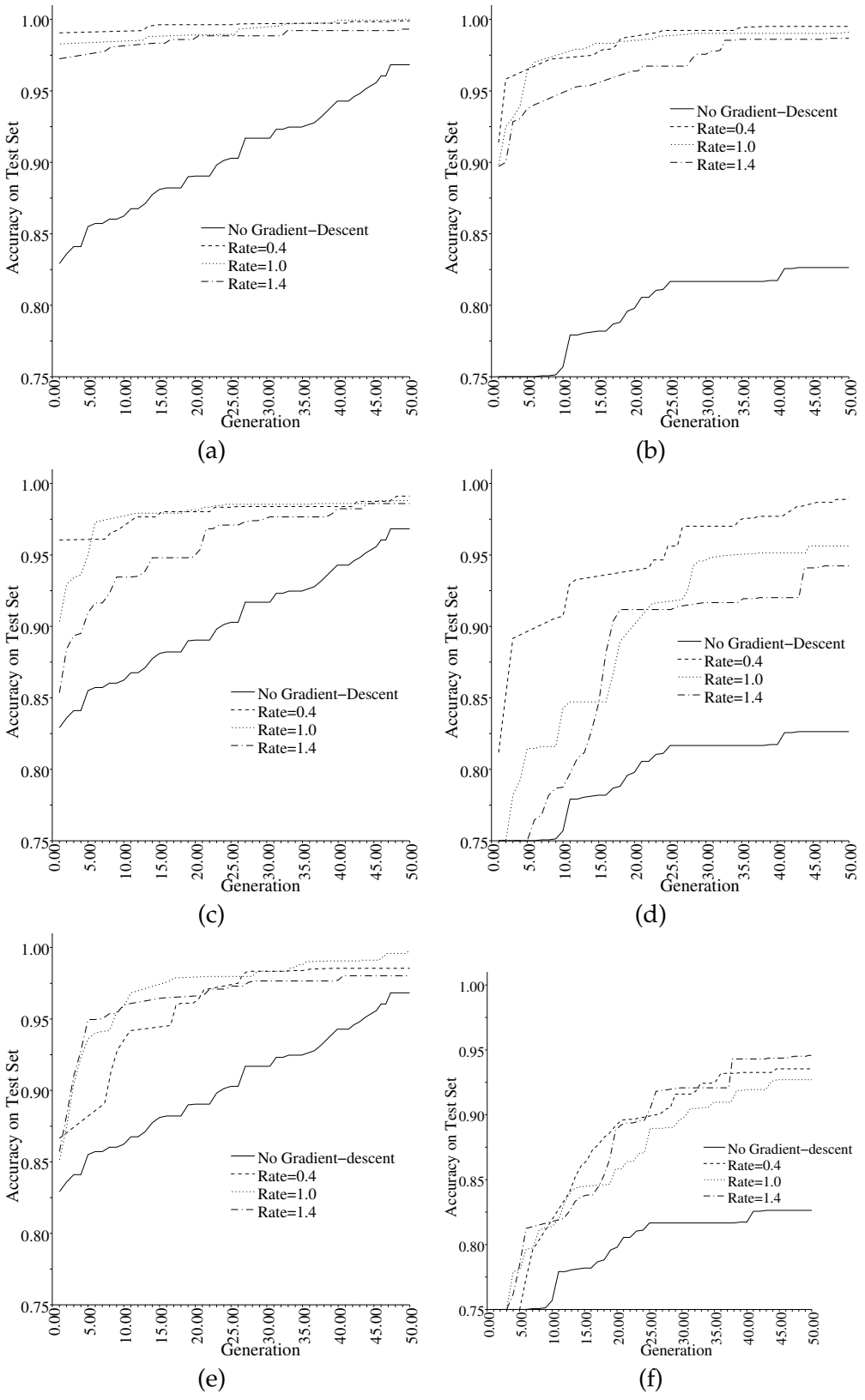


Figure 5.8: Test accuracy trend with differing learning rates (η) using: online learning with summed factor (a,b), simple factor (c,d), offline learning called twice per generation (e,f) in α equation r equation. Avg over 10 runs. Dataset Shape2 (a,c,e), Coin2 (b,d,f).

Chapter 6

Simplification of Programs during Evolution

6.1 Overview

In this chapter, an algorithm to remove the redundancy from programs during evolution is described. Several forms of redundancy are identified, such as addition of numeric terminals (one numeric terminal would do). The process is termed *simplification*, and is applied to all programs in the population periodically during evolution.

Results of using the algorithm are listed, and it was seen to, in general, improve final accuracies on the test set, especially for the harder Coin2 and Coin3 problems.

6.1.1 The Structure of a Genetic Program, and Redundancy

A program, in a GP sense, is often a tree structure. The internal nodes of the tree are called *functions* and the leaf nodes are called *terminals*. It is this tree structure that has been used in this project.

GP programs can encode a number of basic operations into one complex function. For example, a string of terminals can be added or multiplied together, with `if`'s and divisions.

One disadvantage of GP programs describing a complex function built of many basic operations is redundancy. This redundancy is produced through the existence of only a small number of choices for terminals (features and numerics), along with the basic nature of operations such as add or multiply. It is this redundancy that is removed using the algorithms described in this section.

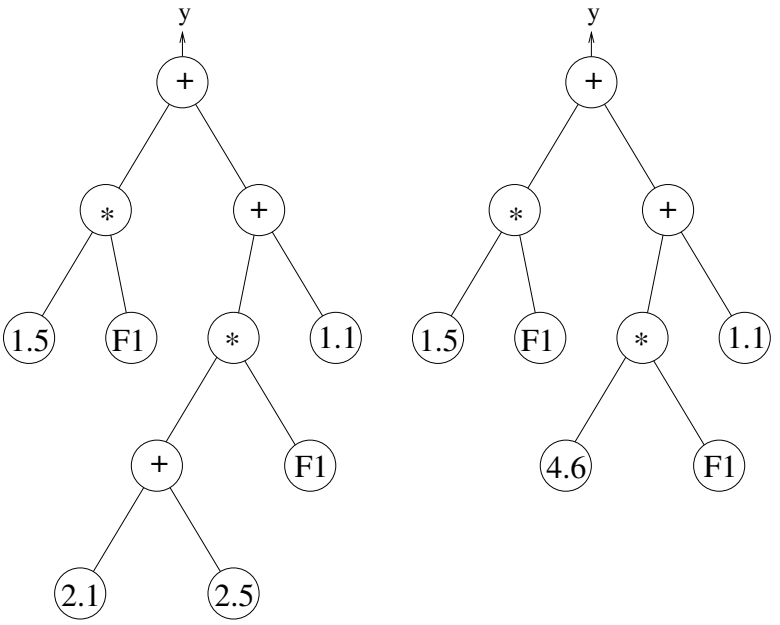
An example program is shown in figure 6.1. It shows a program (a), and some steps producing a far smaller equivalent program (d).

The smaller program (d) is described as equivalent due to its production of the same result for all values of the input features. Thus it will have the same accuracy, and therefore fitness, as the larger program (a), in the system used.

Redundancy may be a bad thing for a program for at least two reasons. One is that the program becomes slower to deal with, the other is that the program may hit a maximum program size limit and not be allowed to grow any further.

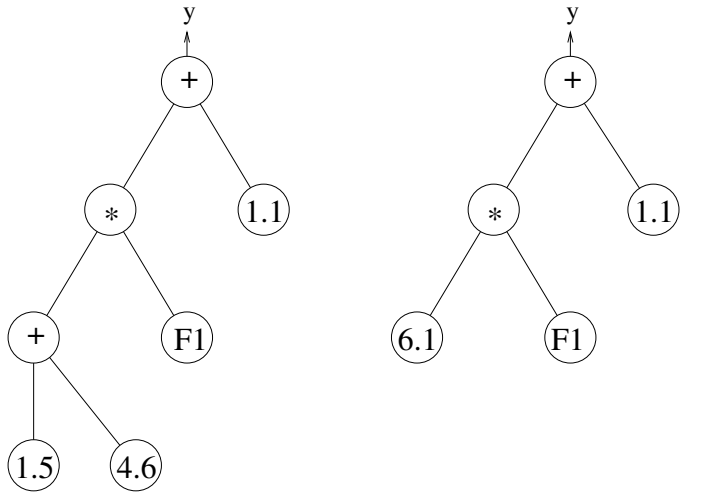
It is unclear however what advantages redundancy may have for a program, such as the preservation of good blocks, or the minimization of effect when destructive operators, such as mutation or crossover, are applied.

It may be that a GP system, in the normal case, will search through the program space for an answer to the problem without consideration for the *density* of the programs produced. In this case simplification will be useful.



F1 * 1.5 + F1 * (2.1 + 2.5) + 1.1
(a) Original program

F1 * 1.5 + F1 * 4.6 + 1.1
(b) After adding two numerics



F1 * (1.5 + 4.6) + 1.1
(c) After extracting common factors

F1 * 6.1 + 1.1
(d) After adding two numerics

Figure 6.1: An example program (a), and equivalent programs (b,c,d) that describe the same function more concisely.

6.2 Forms of Redundancy

The following function set was used in this project: addition, subtraction, multiplication, division (protected in the case of a divisor of zero) and a three argument `if`. Terminals included only numeric terminals, and feature inputs.

Using this primitive set, five easily removed forms of redundancy were identified: (1) an `If` that only ever returns one branch, (2) addition/subtraction of two of more numeric terminals, (3) multiplication/division of two of more numeric terminals, (4) addition of com-

mon factors, (5) division of something by itself. This section describes these five kinds of redundancy.

6.2.1 `if` Just Returning one Branch

When an `if` always returns one of its branches, the `if` will be removed and replaced with the branch.

This occurs whenever the first argument of an `if` is a numeric terminal, such as in the following program:

```
(if<0 -1.5 (+ F1 1.0) 2.3)
```

This is equivalent to the program:

```
(+ F1 1.0)
```

as -1.5 is *always* less than zero.

6.2.2 Addition of Numeric Terminals

When two or more numeric terminals are added together, they can be replaced with a single numeric terminal. This clearly includes numeric terminals subtracted also, as they can be seen as added if the sign of their value is reversed.

For example the program:

```
(+ (- 1.0 0.5) (+ F1 1.0))
```

is equivalent to the program:

```
(+ F1 1.5)
```

as (1.0 - 0.5) + 1.0 is 1.5.

6.2.3 Multiplication of Numeric Terminals

When two or more numeric terminals are multiplied together, they can be replaced with a single numeric terminal. This clearly includes numeric terminals divided also, as they can be seen as multiplied if inversed.

This redundancy is very similar to that of addition of numeric terminals.

For example the program:

```
(* (/ 1.0 0.5) (* F1 2.0))
```

is equivalent to the program:

```
(+ F1 4.0)
```

as (1.0 / 0.5) * 2.0 is 4.0.

6.2.4 Addition of Common Factors

When two or more branches are added together, each made of a multiplication and all sharing a common subtree as a factor, these branches can be further simplified.

This is quite a high level redundancy, and is rare in comparison to the others, but has the potential to reduce the size of a program markedly.

The added branches have the shared common factor subtree removed from the multiplication, and the whole addition is multiplied by the common factor.

For example the program:

```
(+ (* (+ F1 F2) F1) (* (+ F1 F2) 1.0))
```

is equivalent to the program:

```
(* (+ F1 F2) (+ F1 1.0))
```

as `F1 + F2` is a common factor.

6.2.5 Division Causing Unity

When a value is divided by itself, it is replaced with 1.0

For example the program:

```
(/ (* (+ F1 F2) F1) (* (+ F1 F2) F1))
```

is, of course, equivalent to a program that just returns 1.0.

6.3 The Simplification Algorithm

The following program will be used as an example to show how the simplification algorithm works.

```
(ifltz (* 0.414000 -0.406000) (ifltz F3 0.018000 (+ F2 (/ F2 0.718000)))
(* F4 0.950000))
```

The simplifying routine is called periodically (maybe every 5th generation) during evolution. An overview of the algorithm is as follows:

```
procedure SimplifyAll
  for each program p in population
    str = GenerateString( p )
    simplestr = Simplify( str )
    p = ParseString( simplestr )
procedure end
```

The algorithm has three parts: generate string, simplify and parsing the simplified string. These are described in the rest of this section.

6.3.1 Generate String

This step generates a string depicting the program. This string is not the standard expression of the program but instead allows addition and multiplication functions to have any number of arguments so long as it is greater than one.

Subtraction and division are bundled in with the addition and multiplication respectively, with a tag to say they are inverted (tilde). Nested additions and multiplications are joined into one, long operation.

So using this routine our program converts to:

```
(ifltz (* 0.414000 -0.406000) (ifltz F3 0.018000 (+ F2 (* F2 ~0.718000)))
(* F4 0.950000))
```

6.3.2 Simplify

Here the string is parsed recursively, first arguments are parsed, then the whole.

Each function that can simplify (ie. Addition, Multiplication and If) is responsible for simplifying strings of its own type.

The string is already in a suitable state, with additions and multiplications unwound, so this routine is relatively simple.

The steps taken by the addition class are:

- Add up constants and replace them as neccesary, also check if there is only one variable argument and the constants add to zero.
  `(+ F1 0.3 0.3)`→`(+ F1 0.6)`

- Check for common factors in the string, and if there are extract them and call the routine again on the resultant addition.
  `(+ (* 0.1 F1) (* 0.1 F1))→(* (+ 0.1 0.1) F1)→(* 0.2 F1)`

The steps taken by the multiplication class are:

- Multiply constants together and replace them as neccesary, also check if there is only one variable argument and the constants multiply to one.
  `(* F1 0.3 0.3)→(* F1 0.09)`

- Check for arguments that negate themselves (eg. Feature1 / Feature1)
  `(* F2 F1 ~F1)→F2`

The `if` class simply checks whether the first argument is constant, and replaces itself with one of the other arguments as neccesary.
`(ifltz -0.1 F1 0.2)→F1`

So our program goes through the following transformations:

```
   (ifltz (* 0.414000 -0.406000) (ifltz F3 0.018000 (+ F2 (* F2 ~0.718000)))
(* F4 0.950000))
to
(ifltz -0.168084 (ifltz F3 0.018000 (+ F2 (* F2 ~0.718000))) (* F4 0.950000))
to
(ifltz F3 0.018000 (+ F2 (* F2 ~0.718000)))
to
(ifltz F3 0.018000 (* F2 (+ 1.0 (* 1.0 ~0.718000))))
to
(ifltz F3 0.018000 (* F2 (+ 1.0 1.392757)))
to
(ifltz F3 0.018000 (* F2 2.392757))
```

6.3.3 Parsing the Simplified Program String

This could be seen as the inverse of the 'Generate String' step. The string is parsed into a tree form as required by the GP system.

This is neccesary as the additions and multiplications may have more than 2 arguments, where the GP operators can only take two. Also subtractions and divides are indicated only by tags (tilde) on the addition and multiplication arguments, so these must be reformed back into the appropriate operators. Some intelligence is used to ensure the resultant program is minimal.

For example a string with a multiply containing many tagged items must be dealt with carefully, as follows:

`(* F1 ~F2 ~F3 ~F4)`

should convert to something like:

`(/ (/ F1 F2) (* F3 F4))`

containing a multiply of two tagged features, as well as a divide of one tagged feature.

The slightly less optimal/smart option might be:

`(/ F1 (* (* F2 F3) F4))`

Our program string is already in a correct form for uses as a program.

The program is significantly compressed, from 15 primitives to 6.

6.4 Results of Using Simplification

All results listed here are the averages over ten runs, including both graphs and tabular results.

Figure 6.2(a) compares results of runs using the Coin1 dataset with a maximum depth of 4 and varying frequencies of simplification. The graph shows that the runs using simplification achieve a good accuracy more quickly than those without any simplification. This is an example of a very easy task.

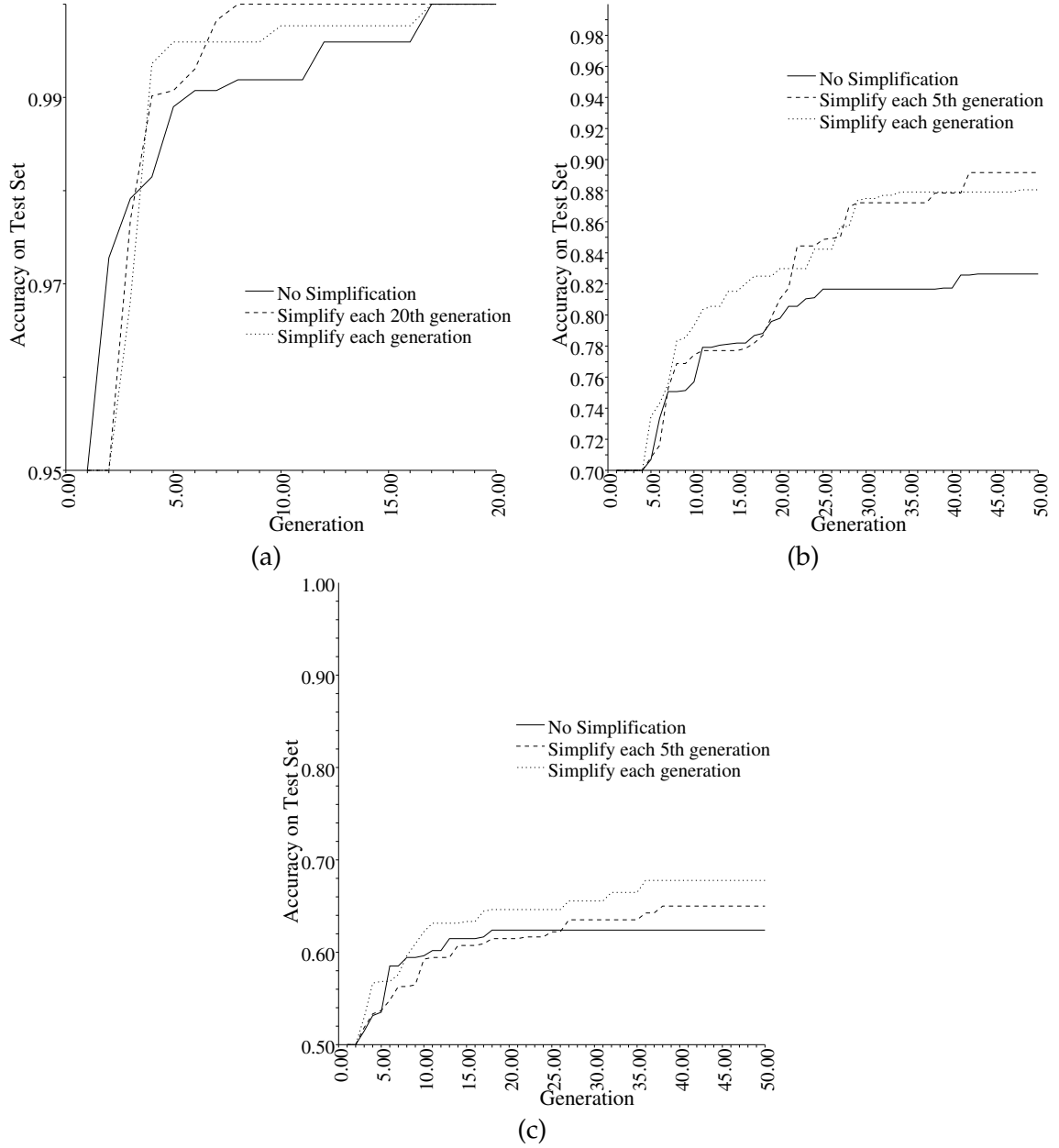


Figure 6.2: Effect of simplification on performance. Dataset is Coin1 (a), Coin2 (b) and Coin3 (c). Maximum program depth is 4 (a), 5 (b) and 3 (c), all forms of simplification used for (a), all but 'common factors' used for (b,c).

Figure 6.2(b) compares results of runs using the Coin2 dataset with a maximum depth of 5 and varying frequencies of simplification. This is an example of a problem of medium

difficulty, and in this instance the runs with simplification, especially where it was used every generation, performed substantially better than the runs without any simplification. This is seen as the dashed and dotted lines are above the solid line.

Figure 6.2(c) compares results of runs using the Coin3 dataset with a maximum depth of 3 and varying frequencies of simplification. The lines using simplification are seen to be a bit above that without simplification (solid).

Table 6.1 shows results of experiments with the simplification operator, on various datasets, and with various frequencies of use and maximum program depths, in tabular form.

Table 6.1: Results for Simplification Operator comparing various maximum program depths and simplification frequencies

Image database	**Max depth for programs**	**Average final generation**			**Test set accuracy (%) at best Validation**		
		Simplification frequency					
		Each gen.	**Every 5th**	**Never**	**Each gen.**	**Every 5th**	**Never**
Shape 1	3	6.50	6.30	8.00	99.77	99.91	99.91
	4	6.10	5.90	6.20	99.95	99.91	99.77
	5	7.10	7.60	9.30	99.81	99.77	99.53
	6	12.20	9.70	10.80	99.91	99.81	99.39
Shape 2	3	51.00	51.00	51.00	86.45	85.93	88.32
	4	51.00	50.80	46.50	94.02	93.55	93.13
	5	47.50	48.30	43.80	94.72	96.40	96.68
	6	43.90	49.00	45.60	97.20	95.65	97.85
Coin 1	3	16.60	20.40	20.60	99.84	99.53	99.74
	4	5.90	4.90	11.00	99.90	99.95	99.53
	5	5.20	6.10	8.00	99.90	99.64	99.53
	6	10.00	7.00	7.10	99.79	100.00	99.90
Coin 2	3	51.00	51.00	51.00	77.88	79.44	79.12
	4	51.00	51.00	51.00	81.50	83.25	82.50
	5	51.00	51.00	51.00	87.56	88.37	82.12
	6	48.50	51.00	51.00	87.63	89.00	89.75
Coin 3	3	51.00	51.00	51.00	64.50	63.50	58.50
	4	51.00	51.00	51.00	65.50	67.50	70.83
	5	51.00	51.00	51.00	76.17	73.50	73.83
	6	51.00	51.00	51.00	72.67	74.67	72.83

In the table it is seen that the simplification operator *usually* helps the GP process.

For the Shape1 and Coin1 datasets (quite easy problems) the simplification of programs represented an improvement to both training time and final test accuracy in all but a few cases. The best case here was for Coin1 with a depth of 4, where training time was reduced from 11.0 to 4.9 by simplifying every 5th generation.

For the Shape2 dataset (a medium difficulty problem) the simplification of programs reduced final test accuracy in most cases, but not markedly.

For the Coin2 and Coin3 datasets (quite difficult) the simplification of programs gave good accuracy in all instances, such as with Coin2 at a maximum depth of 5.

Overall the application of simplification improves more markedly than it degrades, the examples above range from an improvement in accuracy of about 6%, to a lowering in ac-

curacy of about 2% (on the same problem).

6.4.1 Chapter Conclusions

In this chapter simplification was introduced as a method to find, for many programs, a smaller program that is equivalent in terms of output.

In general, by experiment, the compression of random programs is near 1.9, by measuring the number of characters required to print the program before and after simplification, i.e. the program string is usually about half the size after simplification than it was before.

This method was applied to all programs in the population periodically during evolution, in theory making them more dense.

So, is a population of denser programs a good thing? The results compiled for this section indicate that it can be, but is not always.

In most cases, simplification improved the classification accuracy. This is particularly true for difficult problems (such as Coin2 and Coin3).

Simplification may be related to more than one factor of the system. For instance simplification may increase the effective maximum depth, by making denser programs, but may also destroy good building blocks by joining parts of them with other parts of a program (maybe through a common factor).

Further investigation needs to be carried out.

- Does simplification affect program building blocks?
- Does simplification pay for itself, when running time is measured?
- How often should the simplification algorithm be called?

Chapter 7

Conclusions

This project aims to develop new methods which can be used with GP for object classification. The new methods produced a GP system that classifies the objects more quickly for easy problems or, equivalently, more accurately for harder problems.

sectionConclusions that a re Related to the Goals

7.0.2 Classification Strategy

One goal of the project was to find a new classification-strategy that will be more effective on multiple-class problems than SRS. In chapter 4 two new classification strategies were developed and described that do improve speed and accuracy of the GP process when used on certain multiple-class problems, over the SRS case.

The first such strategy was Centred Dynamic Range Selection (CDRS) and the second was Slotted Range Selection (SDRS).

SRS was found to perform very well on problems with either a small number of classes (Coin1) or classes laid out in some implicit order (Shape1), however as expected SRS did not perform well on problems with many classes that are in some arbitrary order. Both CDRS and SDRS were found to be suited to such problems, outperforming SRS.

Both CDRS and SDRS were found to have shorter training times and better final accuracy on all problems not suited to SRS (Shape2, Coin2, Coin3).

The new strategies also used a *weight ratio* parameter, which was seen to affect the performance when changed. For CDRS the weight ratio seemed best at 3. Training time for the Shape1 and Shape2 problems were decreased as the weight ratio was increased.

For SDRS the weight ratio relationship to performance is less clear-cut, with the harder problems performing better when using smaller ratios, and easier problems with higher ratios. The use of the weight ratio was seen to be useful on most problems.

7.0.3 Gradient-Descent of Programs

This report also aimed to show how to utilize a gradient-descent search on the numeric terminals of individual programs periodically during evolution. This was achieved in chapter 5 where algorithms were described to perform gradient-descent on a program using online (stochastic) and offline learning. This formed a type of hybrid search, with an evolutionary beam search globally, and a gradient-descent search on programs locally.

The use of this hybrid search was found to both decrease training time and increase accuracy on object classification all problems.

The online algorithm was found to outperform the offline algorithm. The use of a *summed factor* calculation for the rate also increased performance.

Sometimes the results were extremely good with use of gradient-descent. On the Shape1 and Coin1 problems which, without gradient-descent applied, required 9.56 and 8.00 generations respectively, one generation always sufficed to solve the training problem when online gradient-descent with the summed factor rate was used. More importantly, when one of the rates 0.4, 0.7 or 1.0 was used, the test set accuracy of the resultant classifier was 100%. This is a remarkable result and indicates that the gradient-descent routine could make good use of the structure of the initial, randomly generated, programs. Programs that had a low fitness due only to values of numeric terminals were *fixed* using the gradient-descent routine.

Gradient-descent also worked very well on the harder problems, such as Coin2 where the accuracy was 98.94% (average over 10 runs) on the test set and training took an average of 20.4 generations. This is considerably better than the basic approach, without gradient-descent, where an average accuracy of 82.12% was achieved after 51.0 generations.

These results show that gradient-descent of numeric-terminals is a very useful operation to use on genetic programs during evolution, and can quite markedly decrease the training time, and increase the resultant test accuracy of classifiers on these object classification problems.

Different learning rates for the online algorithm produced only slightly different results, but 0.4 seemed to be a good starting point.

Although used here for classification, there seems no good reason not to expect similar results with use of gradient-descent on other GP applications.

7.0.4 Simplification

Another goal of the report was to develop an algorithm to remove redundancy from individual programs periodically during evolution. This was achieved by the simplification algorithm described in chapter 6. This algorithm identified certain types of redundancy in a program, outputting a program *equivalent with respect to output*.

The types of redundancy removed include: numeric terminals added/subtracted/divided/multiplied together, `if` statements that always return a particular branch, common factors in additions and branches divided by themselves.

Random programs were found to compress on average by a factor of 1.9 using number of characters required to print the program as the metric.

The use of this algorithm on all programs in the population periodically during evolution was found to improve performance. On the easy problem Coin1, training times were decreased, while maintaining near perfect accuracies on the test set. On the harder Coin2 and Coin3 problems, final test accuracy accuracy was improved.

7.1 Other Findings

7.1.1 Online versus Offline

For gradient-descent, the online algorithm performed better than the offline algorithm. Offline learning in this case was worse only because online learning converged more quickly.

The results in chapter 5 for the offline algorithm perform only two passes of the training set patterns in each generation, thus numeric terminals in the programs are updated twice per generation. Compare this to the online algorithm which updates numeric terminals for each training pattern, giving for example 252 updates per generation (algorithm run once per generation) for the Shape1 and Shape2 datasets. The low frequency of use for the

offline algorithm is due to the time taken to move through all the patterns, to determine the gradient vector of the cost surface. Note that the algorithm is run on all programs (500).

So on a faster system with optimizations in place for the algorithm, such as easier access to the list of numeric terminals, the offline routine could be run more times per generation without the severe performance hit, and could exceed the results of the online routine.

There seems no reason that the *summed factor* rate calculation could not be extended to the offline algorithm.

7.1.2 Overfitting of Programs due to Training Set Order

The order of patterns was found to be important when the online algorithm was used. Initially the patterns were simply used in the order of the image they came from. This was far from random, and this introduced and interesting effect to the results. Though the final results were similar, the training process produced occasional generations where the test set accuracy was low. This was caused by overfitting of the last few patterns in the training set, which were of the same class.

Interestingly this overfitting had no effect on the system as a whole due to the *turnover* of programs used for the results. The results are found using the best program in the population, that may change from generation to generation, and only a small fraction of programs suffered from the overfitting.

After randomizing the order of the training examples, this problem was removed. This suggests that the training examples should be randomly organized if a smooth training process is seen to be advantageous. This is particularly true of problems with small numbers of examples.

7.2 Future Work

- CDRS showed much promise, which it has partly fulfilled in the results of chapter 4. CDRS has two advantages over SRS, on which it was based. The boundaries in SRS are set at the beginning of evolution and do not move. The boundaries of CDRS, however, can both move continuously and change in order.

  CDRS could be improved further by more intelligent use of the mobile boundaries. Currently they are set indirectly by the *centres* of the program outputs for the classes, a very crude approach. A better approach may be to perform a gradient-descent (or similar) search on the boundary positions using the sum squared error cost surface. This would cause a lower cost (better population) after the reclassification step, something not true of CDRS currently.

- The offline learning algorithm shows promise, and could be improved by calling more times per generation (would need to be faster) or the use of a rate calculation such as is used for online learning.

- The `if` function does not perform well with a gradient-descent search, as the output of the program is not differentiable by the first `if` parameter (the conditional part). This means that the branch coming from this argument is not searched. A better idea may be to use more soft edges for the cutoff of the other two arguments, that is use a soft-max function. This way the program output may differentiable by the first argument, and so the entire tree may be searched, while still being able to represent (fairly) discontinuous functions.

- Gradient-descent, as described in this report, may be seen as not particularly biologically plausible. This is due to the fact that there is no such refining of specific sequences of DNA contained in the cells of creatures. Consider a chromosome, corresponding to a GP program, some gene in the chromosome, corresponding to a numerical terminal in the program, is not altered in any continuous way by evolution, it is simply subject to reproduction, crossover and mutation.

  The gradient-descent is more biologically-plausible if the program is not changed by it for purposes of these genetic operators, that is the effect is like the differentiating of cells in the body to different tasks and occurs only for the lifetime of the creature. It can be seen that this approach is equivalent to using gradient-descent *only for the purposes of the fitness function, and final result*, and would just give a different view of which programs are fit, one less reliant on the values contained in the program's numeric terminals.

  This would be an easy change to the algorithm, and could be good for the diversity of the population.

- The best use for the simplification operator may simply be at the end of the evolution process, giving a smaller program than the case without simplification, but still the same performance.

- More features could be used, to provide better performance than the simple features used here.

- The lessons learnt here could be applied to detection problems.

- More difficult tasks could be attempted, such as face classification.

- A comparison could be made between the classifications used here, and binary decomposition.

- A comparison could be made to evaluate whether GP is better for these multiple-class object classification problems than other methods, such as: Neural Networks, Support Vector Machines, Decision Trees, Genetic Algorithms.

Bibliography

[1] J. R. Koza, *Genetic Programming: on the programming of computers by means of natural selection*. London, England: Cambridge, Mass. : MIT Press, 1994.

[2] J. R. Koza, "Genetic programming," in *Encyclopedia of Computer Science and Technology* (J. G. Williams and A. Kent, eds.), vol. 39, pp. 29–43, Marcel-Dekker, 1998.

[3] H. Lipson and J. B. Pollack, "Automatic design and manufacture of robotic lifeforms," *Nature*, vol. 406, pp. 974–976, August 2000.

[4] T. Loveard and V. Ciesielski, "Representing classification problems in genetic programming," in *Proceedings of the 2001 Congress on Evolutionary Computation*, (Seoul, South Korea), pp. 1070–1077, IEEE Press, May 2001.

[5] W. A. Tackett, "Texture classifiers generated by genetic programming," in *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002* (D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, eds.), pp. 243–248, IEEE Press, 2002.

[6] D. Howard, S. C. Roberts, and R. Brankin, "Target detection in sar imagery by genetic programming," *Advances in Engineering Software*, vol. 30, pp. 303–311, 1999.

[7] M. Zhang and V. Ciesielski, "Genetic programming for multiple class object detection," in *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence (AI'99)* (N. Foo, ed.), pp. 180–192, Springer-Verlag Berlin Heidelburg.

[8] M. Zhang, P. Andreae, and M. Pritchard, "Pixel statistics and false alarm area in genetic programming for object detection," in *Applications of Evolutionary Computing, Lecture Notes in Computer Science, LNCS* (S. Cagnoni, ed.), vol. 2611, pp. 455–466, Springer-Verlag Berlin Heidelburg, 2003.

[9] M. Zhang, V. Ciesielski, and P. Andreae, "A domain independent window-approach to multiclass object detection using genetic programming," *EURIASP Journal on Signal Processing, Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis*, vol. 8, pp. 841–859, 2003.

[10] J. F. Winkeler and B. S. Manjunath, "Genetic programming for object detection," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 330–335, Morgan Kaufmann, 13-16 1997.

[11] W. A. Tackett, "Genetic programming for feature discovery and image discrimination," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93* (S. Forrest, ed.), (University of Illinois at Urbana-Champaign), pp. 303–309, Morgan Kaufmann, 17-21 1993.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations* (D. E. Rumelhart and J. L. McClelland, eds.), vol. 1, ch. 8, Cambridge, MA: MIT Press, 1986.

[13] W. M. Spears, K. A. D. Jong, T. Bäck, D. B. Fogel, and H. de Garis, "An overview of evolutionary computation," in *Proceedings of the European Conference on Machine Learning (ECML-93)* (P. B. Brazdil, ed.), vol. 667, (Vienna, Austria), pp. 442–459, Springer Verlag, 1993.

[14] M. Mitchell, *An introduction to genetic algorithms*. Cambridge, Massachusettes: MIT Press, 1996.

[15] H. de Garis, "Genetic programming : Modular evolution for darwin machines," in *Proceedings of the 1990 International Joint Conference on Neural Networks*, (Washington, DC), pp. 194–197, Lawrence Erlbaum, 1990.

[16] D. J. Montana, "Strongly typed genetic programming," Tech. Rep. 7866, BBN Technical Report, Cambridge, MA 02138, March 1994.

[17] W. M. Spears, "Crossover or mutation?," in *Foundations of Genetic Algorithms 2* (L. D. Whitley, ed.), pp. 221–237, San Mateo, CA: Morgan Kaufmann, 1993.

[18] S. Luke and L. Spector, "A comparison of crossover and mutation in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 240–248, Morgan Kaufmann, 13-16 1997.

[19] Yao, "Evolving artificial neural networks," *PIEEE: Proceedings of the IEEE*, vol. 87, 1999.

[20] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus, "Knowledge discovery in databases - an overview," *Ai Magazine*, vol. 13, pp. 57–70, 1992.

[21] R. Poli, "Genetic programming for image analysis," in *Genetic Programming 1996: Proceedings of the First Annual Conference* (J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds.), (Stanford University, CA, USA), pp. 363–368, MIT Press, 28–31 1996.

Appendix A

Use of Programs

This chapter describes the basic use of the programs written for this project.

A.1 genpat

genpat generates the pattern files used in the GP system. The command line format is:

```
   genpat [win_width] [win_height] [start patt] [end patt/0]
[class_position_file]
```

[win_width] [win_height]: Feature widow width and height.
[start patt]: 1 based first pattern index to put in file.
[end patt]: 1 based last pattern index to put in file, or 0 for the last.
[class_position_file]: A position file, such as the following:

```
\coin-easy-23.pgm
1 5t 77 442
2 5h 342 451
3 10t 197 448
4 10h 474 451
0 back 269 396
\coin-easy-24.pgm
1 5t 196 79
2 5h 467 70
3 10t 333 81
4 10h 83 81
0 back 264 141
\end
```

Here, the first and seventh lines indicate image files. The last line should be \end. The other lines have: class number, class name, x, y.

A.2 clsgp

clsgp is the main program for evolving classifiers, extensive modifications were made to the basic RMITGP system.

The help screen is as follows, with comments:

```
Usage ./clsgp [option] [option] ..

   where option is

   mut:<real>         mutation rate          (default : 0.2)

   elite:<real>       elitism rate           (default : 0.1)

   slotsize:<real>    size of clsfy slots    (default : 0.25)
```

Slotsize is used in all classification strategies to specify either the minimum boundary to boundary width (SRS, CDRS) or the width of the slots (SDRS).

```
   fitweight:<real>   comparitive best reclas weight (default : 1.0)
```

The weight ratio parameter for SDRS and CDRS.

```
   rate:<real>        rate of err propagation(default : 0.0)
```

This is the control for the amount of error propagation used. 0.0 indicates none, otherwise is serves as the rate η.

```
   rategens:<int>     gens to apply rate for (default : all (-1))
```

The number of generations ot apply the rate for.

```
   ratetimes:<int>    times to apply rate per gen (default : 1)
```

The number of times the error propagation is called per generation.

```
   ratetype:[divtot,online] rate type        (default : neither)
```

This indicates whether simple or summed factor is used.

```
   gen:<int>          number of generations  (default : 99)

   pop:<int>          number in population   (default : 500)

   maxdep:<int>       maximum depth of tree  (default : 6)

   mindep:<int>       minimum depth of tree  (default : 1)

   initmaxdep:<int>   maximum depth of initial pop (default : 6)

   initmindep:<int>   minimum depth of initial pop (default : 1)
```

Unfortunately the system may crash is the init depths are the same as the other depths.

```
   trainfn:<fn>:st:en training filename(default : hardcoded)
```

Training filename, start pattern, end pattern.

```
testfn:<fn>:valst:val:tsten testing filename (default : fn:1:0:0)
```

Testing (and validation) filename, start of validation set, end of validation set and start of test set, end of test set.

```
popfn:<fn>          population filename    (default : none)

resfn:<fn>          results filename       (default : none)
```

Where to put results (of best program)

```
sumfn:<fn>          result summary filename(default : none)
```

Where to append summary (of evolution)

```
imgfn:<fn>          graph image filename   (default : none)
```

Allows for a heat-map to be drawn

```
feat:1x10y001..     Use these features     (default : 11111..)
```

Features for the heat map

```
notfunc:[+-*\%i]    exclude function(s)    (default : none)

onlyfunc:[+-*\%i]   set used functions     (default : +-*%i)
```

Allows specialization of the functions used

```
clstype:[srs/bdrs/drs/node] classify type (default : srs)
```

The classification strategy used, srs = SRS, bdrs = SDRS, drs = CDRS, node is obsolete.

```
fittype:[acc/more/2nd] fitness type        (default : acc)

recls:<int x>       reclassify each x gen  (default : 10)
```

Frequency of reclassification

```
prbest:<int x>      print best each x gen  (default : don't print)
```

Frequency to print best program

```
prfn:<fn>           generation summary filename(default : none)
```

Where to append summary each generation

```
simple:<int x>     simplify each x gen    (default : off)
```

Frequency of simplification

```
simptype:[all/basic] Simplify type        (default : all)
```

Type of simplification to use

```
seed:<int x>       rand seed (or 0 for time) (default : 0)
```

Seed for random numbers.

A.3 Other Programs

Some supporting programs are very briefly described here:

A.3.1 `inter, stats`

`inter` and `stats` allow for compilation of the results of runs, they are quite general.

A.3.2 `plan`

`plan` generates a sequence of experiments to a batch file, in so doing it can expand about 20 lines to thousands of runs of the program, with different combinations of parameters.

A.3.3 `plotter`

`plotter` draws various types of graphs from raw data, to an x-fig file.