Chapter 1

Validating our GP Learning System (DRAFT)

Although our brain cleverly gives us the impression we can see the whole world in one go, our attention is usually focussed on just a small part of it. This is partly to do with the way the human eye is constructed, and partly to do with the way our brain chooses to filter the information. Similarly, while most images contain a huge amount of data, many tasks in vision are reliant on finding and using just a small subset of that information to work on. A recurring theme in computer vision, therefore, is how we can identify of these useful parts so that the remainder can be discarded — a crude kind of attention. As we'll see later in this thesis, the critical learning component of our vision system is a classification algorithm which aims to do exactly that.

However, such considerations are not confined to computer vision alone. Now that we humans can collect and store more information faster than ever before, finding algorithms to make sense of the piles of data we've amassed has been a topic of data-miners and information theorists for a long time. Consequently there are a plethora of algorithms which can be trained to classify sets of information.

In a previous study (by Lim, Loh, and Shih, 2000) 33 different classification algorithms were tested and compared on the same datasets. Cielsielski and Loveard later added their results after using GP. With the algorithms compared in rank order, one particular GP representation (DRS, to be discusse later) was found to be quite competitive with the 33 other techniques for solving binary problems (ranking as high as fourth), although in general the GP programs' rankings were mediocre.

While it is true that there is no such "best" classification algorithm out there, which consistently outperforms all others on any data set, it is certainly true that some algorithms can find good solutions substantially more efficiently than can Genetic Programming. Genetic Programming is often regarded as being slow and inefficient; this is not necessarily unjustified. The nature of Genetic Programming is to produce a large population of programs and to discard the majority. By contrast, techniques such as XCS aim to evolve and improve a single system through less blunt reinforcement and thus converge upon a solution significantly more quickly. Most other classification algorithms, outside the realm of evolutionary computation, are not encumbered by populations of individuals at all.

Although populations are inhererently wasteful, the nature of the genetic operators is also of concern. The process of crossover, by which new combinations are found can be more destructive than constructive. In nature, the chance happening of mutation is lethal to the individual in almost all cases. Fortunately it is also a very rare occurrence. In GP, however, mutation rates are much higher than would be the case in the natural world.

Even if GP can be employed to classify between two clusters of data, other techniques such as SVMs also aim to maximise the *margin* between the clusters to improve generalisation performance. Although the optimisation process of SVMs can be relatively computationally expensive, newer techniques have been shown to speed up the process markedly[1]

So why choose GP at all for classification? There are good reasons to avoid using GP on certain tasks. However computer vision poses certain challenges that may suit GP better. Computer Vision is a notoriously ambitious area of research. It is difficult to see how a discipline which is arguably only twenty years old can ever seek to match the combined evolution of the human eye and brain, something that nature has been working on for millions of generations. One of the most difficult parts of computer vision is its subjective nature which makes it hard to pose problems concisely. Such difficulties may leave the researcher unsure as to where he or she should begin. Genetic Programming is not encumbered with the need for a problem definition, except in the broadest sense. Neither is it prone to following a particular line of enquiry. In many aspects one may argue that Genetic Programming is an ideal researcher, working tirelessly until a result meets a set of criteria. If the result turns out to be inadequate, then the criteria themselves may be made more rigourous, but it isn't necessary to deliberate as to *why* the result was poor.

This argument may be made for any machine learning technique, so what motivates us to use genetic programming instead of the other algorithms? Poli suggested that neural networks are unable to compete with GP because of their inherently linear nature. His experiments showed subjective results that GP could outperform a neural network, although the absolute difference in fitness was actually quite small.

Genetic Algorithms and neural networks are not as flexible as GP: their representation is relatively fixed. A GA in its basic form aims to optimise a series of values; neural networks do similarly but in a different way. The simplest adaptation of neural networks for image processing is to assign each input of the net to a pixel on the image and allow the net to learn some kind of useful relationship between different regions of the image. One may say the problem with this approach is that it needs to agglomerate a lot of separate (possibly redundant) information before arriving at any classification, which may render the net computationally too expensive. On a broader level the problem is that the net's fixed set of topologies make it hard to adapt them for generic problem solving.

One distinct flaw among other classification algorithms is their inability to manipulate features in a non-linear way. While one supplies the GP system with the same features as any other classification system, there is no encumbent need for GP to use them in their raw form. Perhaps the distinction between two classes is based on the difference between two of their features, rather than their absolute values. In this case GP should be able to find the solution (by dividing the two absolute values of the features and creating a new feature), but could other learning algorithms. In our quest to find generic vision systems, it seems prudent to choose a learning system that can continue to learn, even when the solution is beyond our preconceptions of the problem. Following the house-buying adage that one should always "choose the worst house in the best neighbourhood rather than the best house in the worst

[1]The problem can be re-formed from a geometric perspective and converted into a problem involving reduced convex hulls, the upshot of which is the same solution can be discovered more quickly, Dimitriodis, 2008)

neighbourhoor", we choose not to use the best classifiers at this moment, but a classifier with the most potential for solving solutions in the future. Consequently a large part of this thesis investigates the ways in which GP can be made less parameter heavy, and how one can create good clasifier more effectively.

Genetic Programming is a rather abstract learning system, indeed most genetic programming systems are constructed without any concept of the problem or solutions that GP may be required to find. This is because virtually any nodes and training information can be slotted into the Genetic Programming interface, after which the system evolves solutions based purely upon results. For this reason our first task in this thesis is to describe how one particular GP system may be constructed, and then decide whether it is a suitable learner for the tasks at hand.

1.1 Building a GP System

Although the author's initial experiments used ECJ[1], a popular, freely-available genetic programming toolkit, it became apparent that in order to experiment with changes to the GP environment itself, it would be more worthwhile to develop the GP system itself from scratch. Our own toolkit is better equipped for solving problems involving vision but crucially allows us to tinker with some under-the-hood aspects of Genetic Programming to investigate different ideas.

1.1.1 GP System Features

Our GP system includes a variety of features that do not exist in other software. These include:

Strong-er Typing Depending on the set of operators and functions used and the maximum tree size permitted, the number of different program permutations can be vast. The purpose of Genetic Programming is to find good solutions without resorting to an exhaustive evaluation of the program space. One way of reducing the search space is through the use of strong-typing, where the tree builder, which is responsible for constructing the programs (and new mutations) is constrained to only connect nodes which make semantic sense. For instance, the *LessThan* operator, which returns a boolean value, should not be supplied as an argument to the *division* operator - one would get divide-by-zero errors and a rather inconsistent logic. Although Koza developed a clunky solution to this problem, Montana[2] introduced the notion of strong typing to GP toolkits.

Strong typing in Montana's form has its disadvantages. While it is possible to define a node's output as *numeric*, or *boolean*, these categories may be too prescriptive. If one is looking for a count for a loop, for instance, a floating point number is not necessarily appropriate - it is better to have an integer. However, if one defines integer as a new *type* to accommodate this, then the arithmetic operators would no longer accept it. Essentially a node should be able to return more than one type, for instance to say that it returns a number that is also an integer. Our GP system includes this flexibility, where each node may implement multiple types. This permits the system to make use of more complex structures while making fewer trees that don't make semantic sense.

Tree Checking Although GP is capable of evolving many means of procrastination, our stronger-typing approach still permits the creationg of “useless” code that performs no function. Code may be considered useless if it is never executed, or if it always returns the same value regardless of input. The former is usually to be found in branches of if statements whose condition is accidentally a constant.

The tree builder in our GP toolkit generally avoids creating such code by following additional criteria, set by the function nodes. For instance our *lessThan* node insists that at least one of the child nodes in the subtree beneath it be a feature from the image. This ensures that every sub-tree has the potential to perform some useful processing[2].

Automatic Optimisation Our GP system automatically optimizes the best individual at the end of evolution, in order to make it suitable for deployment in the computer vision task. After running the individual on the data, it collects statistics about each node, then removes nodes that are never used and replaces nodes that always return the same value with constants.

1.1.2 Comparison to ECJ

Before discussing the merits of genetic programming in a more general sense, it is first necessary to validate our own GP toolkit in terms that it is at least as effective as the popular ECJ toolkit. To do this we will run equivalent experiments on each toolkit and evaluate which is the most efficient, and which produces the better result.

1.2 Generifying GP

One can think of a genetic programming system as a whole “world” in which the birth, life and death of thousands of individuals are simulated by a series of sub-components. As such, one of the problems of developing solutions with GP is the sheer number of parameters required by each component: there are parameters for the minimum and maximum initial depth of individuals, the kind of tree builder to use, the population size, number of generations, choice of fitness function, and so on. Moreover, it is difficult to assess the true effect of any parameter since all the components are related. Many papers tune parameters for each problem, which doesn’t offer a particularly generic solution. If we are to maintain our aspiration for a generic system, it is necessary to consider the research and theory behind different parameter choices in order to establish a set of parameters that work well on a wide range of problems. These are discussed below.

1.2.1 Genetic Operators

A standard breeding pipeline involving crossover, mutation and reproduction operators was used.[3] shows that different blends of each of these operators do not make much difference to the performance of the system, so in this work crossover accounts for about 75% of the new population, mutation for 20% and the remainder is obtained simply by replication.

[2]Although it is still not perfect: the lessThan function could compare a feature whose return values are in the range 0-255 to a value of 1000, which would again always return the same value

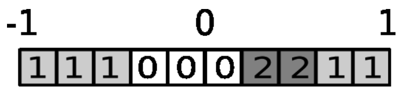

Figure 1.1: A Program Classification Map

1.2.2 Bloat Control

A common issue in GP is the presence of useless code segments in individuals, known as ‘code bloat,’ reducing the efficiency of the learning process. Indeed, one might argue that individuals that are padded out with unused, and thus expendable, pieces of code are more likely to survive intact: there is a selective pressure which encourages the proliferation of code bloat.

Various techniques are used to reduce bloat, including parsimony pressure and dynamic maximum tree depth [4], which penalise or remove large individuals respectively. However, for a generic system, it is difficult to quantify ‘large,’ so an alternative solution was employed. Elitism, a commonly used technique which copies the best individuals in the population directly into the next generation, was used as studies have shown that evolution with elitism enabled has smaller average population sizes [5].

1.2.3 Tournament Size

Genetic Programming is distinguished from random search by the evaluation of individuals’ performance and the subsequent selection of good individuals to produce next generation. The most widely used selection technique, tournament selection, selects t individuals from the population, then selects the best. Larger tournament sizes t will increase the selective pressure.

Although the genetic algorithm community has traditionally used a tournament size $t = 2$, GP researchers generally use $t = 7$ as this is thought to instill higher selective pressure on the population, tending to improve the learning rate during the limited time of a GP run (GA is potentially faster to run than GP). Our experiments show that, while $t = 7$ generally achieves lower average fitness, its performance is generally more variable — and the best individual can still occasionally be found during a $t = 2$ run. It is difficult to decide which is the best parameter so our preferred approach is simply to try both.

1.2.4 Classifier Representation

A common approach in GP is to treat the individual as an evolved mathematical function which produces values along a continuous range.

Although a threshold may be used to draw a boundary between classes, it may be inadequate if the data cannot be divided neatly along a single plane.

A better means of translating this value into some class label is by using a program classification map (Figure 1.1), in which the range of values is split into a number of slots, each slot being allocated a particular class label. In dynamic range selection (DRS), the labels for each slot are optimised for each individual during runtime by counting which samples are being assigned to which slot, then selecting the most prevalent classes’ label for the slot. DRS allows programs to be smaller as it removes the need to evolve thresholds and generally simplifies the problem. In a comparison of several classification representations, DRS was found to be the most effective strategy [6] and was therefore used in this work.

Bibliography

[1] Sean Luke. ECJ: A Java-based evolutionary computation research system v14. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[2] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[3] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[4] Sara Silva and Jonas Almeida. Dynamic maximum tree depth. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1776–1787, Chicago, 12-16 July 2003. Springer-Verlag.

[5] Riccardo Poli, Nicholas F. McPhee, and Leonardo Vanneschi. Elitism reduces bloat in genetic programming. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA, 12-16 July 2008. ACM Press. forthcoming.

[6] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077. IEEE Press, 27-30 May 2001.