A New Program Structure in Genetic Programming for Object Classification

Y. Zhang and M. Zhang

School of Mathematics, Statistics and Computer Science,
Victoria University of Wellington, New Zealand
{yz, mengjie}@mcs.vuw.ac.nz

Abstract

This paper describes an approach to the use of genetic programming (GP) for multiclass object classification. Instead of using the standard GP approach where each genetic program returns just one floating number that is then translated into different class labels, this approach invents a new program structure called *Modi* with multiple outputs. A voting scheme is then applied to these output values to determine the class of the input object. The approach is examined and compared with the basic GP approach on four multiclass object classification tasks with increasing difficulty. The results show that the new approach always outperforms the basic approach with controllable proper setting.

Keywords: Genetic programming, image recognition, multiple class object classification, modifying-based program structure, Modi.

1 Introduction

Object classification arises in a very wide range of applications, such as detecting faces from video images, recognising digits from the postal codes, and diagnosing all tumors in a database of x-ray images. In many cases, people (possibly highly trained experts) are able to perform these tasks well, but there is either a shortage of such experts, or the cost of human expert is too high. Given the amount of image data that needs to be classified, automated object classification systems are highly desired. However, creating automated object classification systems that have sufficient accuracy and consistency turns out to be very difficult.

Genetic programming (GP) is a relatively young and fast developing machine learning and searching paradigm [1, 3]. Different from traditional learning methods such as *hill climbing* [5] and *artificial neural networks*[5], which take more advantage of the detail of an individual creature hence learn by that individual alters the configuration of itself, the evolutionary learning procedure of GP learns by following a genetic beam search to produce skillful survival though creatures do not themselves learn during their individual lifetime.

GP research has considered variety kinds of classifier program representations, including decision tree classifiers and classification rule sets [3]. Recently, a new representation – numeric expression classifiers – has been developed for GP [4, 8]. It has been successfully applied to real world classification problems such as detecting and recognising particular classes of objects in images [7, 8], demonstrating the potential of GP as a general method for classification problems.

The output of a the GP learnt classifier is a numeric value that would be later translated into a class label. For simple binary classification, the translation can just base on the sign of the numeric value [4, 2, 6, 9], hence is generally fair; For multiclass problems, to find appropriate boundary values to separate different classes is more difficult. Current solutions are mostly to separate the program output space into regions, one per class. These include a primary static method such as *static range selection* [4, 8], more complicated *dynamic range selection* [4], and *centred and slotted dynamic class boundary determination* [6, 9]. Past works have demonstrated the effectiveness of these approaches on a number of object classification problems.

In the static method, the program's output space needs to be manually partitioned into fixed regions in a fixed ordering, resulting in either the expenses on human experts on the task, or inappropriate partitioning hence poor performance. In the dynamic methods, class boundaries can be automatically learnt, though requires much longer learning time, and often results in unnecessarily complex programs [8, 9].

This paper solves the problem from a different respect. Rather than allowing a program to return one floating point value then translating it into class labels, we developed a new program structure *Modi* that can "return" multiple outputs, each corresponding to a class. The class label can then be determined by a simple voting strategy, thus avoids the painful class label translation step. The elite of our Modi structure is that it still takes the advantage of the standard GP on the way the program is learnt.

The rest of the paper is organised as follows. Section 2 describes the new program structure, Modi. Section 3 describes the experiment configuration and image data sets. Section 4 presents the results along with discussions. Section 5 draws the main conclusions and gives future working directions.

2 The New Program Structure – Modi

2.1 Modi Program Structure

The Modi program structure has two main parts: (a) a program tree, and (b) an associated *output vector* for holding outputs, as shown in Figure 1.

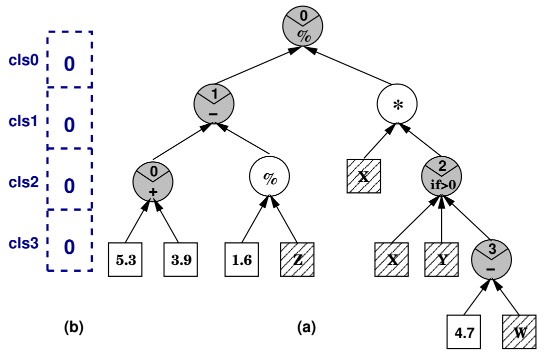

Figure 1: An example Modi program structure. (a) Modi program tree; (b) Output vector.

Similarly to the standard program tree structure, the Modi tree also has a root node, internal nodes and leaf nodes. Feature terminals (slashed squares) and random constants (clear squares) form the leaf nodes. Operations in the function set form the root and internal nodes (circles).

Unlike the standard program tree structure, which outputs just one floating point number through the root, our Modi program structure takes the *output vector* as the outputting space, hence produces multiple values, each of which corresponds to a single class in the multiclass classification problem.

The two parts of the Modi structure, namely the output vector and the Modi program tree, are not structurally connected though are connected through a special kind of function node, the *Modi node*, as shown in grey circle in Figure 1. A Modi node has two roles: (1) It would *update* an element in the output vector that is pre-associated with it, by adding its node value to the value of the element; (2) It would pass the value of its right child node up to its parent, so that the tree structure of the program could be preserved along the program evaluation.

Note that the output vector is considered a *virtual* structure (shown as dashed). It does not physically "exist" other than the moment the program *is being evaluated*. During evolution, output vectors of all programs "disappear" from being involved, only the tree part of the Modi structure are activated and take part into the evolution. Only in the program evaluation time, is the output vector realised and receives updating from the program tree.

2.2 Evaluation of Modi Programs

Figure 2 shows what happens while the example program in figure 1 is evaluated. In the very first place before the evaluation starts, the virtual output vector is realized and initialized with zeros, shown as the dash drawing of the vector becomes solid. During the evaluation, each *non*-Modi node passes its value to its parent, exactly the same as in the standard program tree. The Modi node does differently. It firstly uses its node value to update the output vector, then passes on the value of its right child to its parent node. The consequence of the program evaluation is that the output vector gets properly updated based on the input object, by all Modi nodes in the tree. A list of floating point numbers from the output vector are then produced, each of which corresponds to a class. Finally, a voting strategy would be applied to those outputs. The winner class (the one with the maximum value) is considered to be the class of the input object.

Let us go through an example. consider a four class object classification task with possible classes {`rat,cat,dog,pig`}, and an object to be classified – *doggie*, represented by an input vector that consists of six extracted feature values `[V,U,W,X,Y,Z]=[0.6,5.7,8.4,2.8,13.6,0.2]`. Assuming the Figure 2 Modi program is the learnt classifier, feeding the input vector into the classifier would *modify* the output vector to `[9.54,-4.1,13.6,-3.7]`, as shown in the left part of Figure 2. Thus *doggie* is classified into the third class `dog`, as the third output `13.6` is the *winner*.

2.3 DAG Simulation Effect of Modi Programs

Figure 3 is just a tidy-up redrawing of Figure 2 with the structural-used-only fine grey lines all been removed. The figure clearly shows that the running effect of the Modi program actually

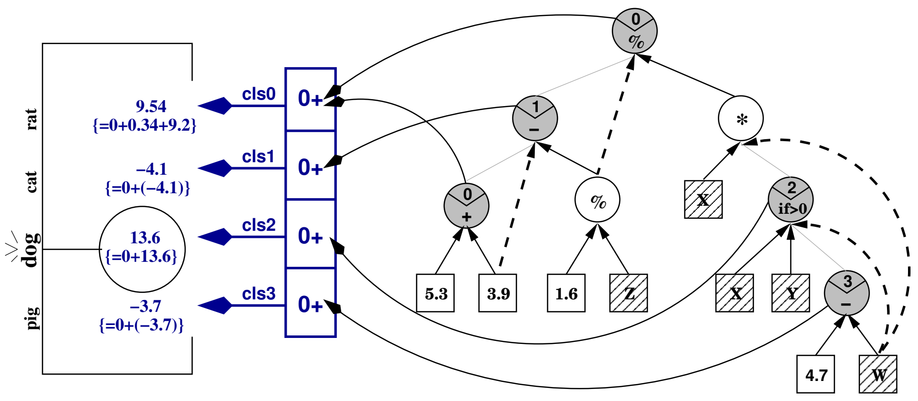

Figure 2: Evaluation of the example Modi program structure.

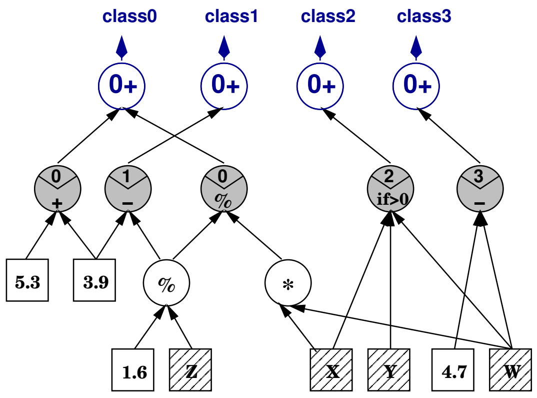

Figure 3: Modi simulated graph/network classifier.

simulates the *Directed Acyclic Graph* (DAG), although the real structure used by Modi is just a normal tree plus a vector. Compared with the multilayer feed forward neural network, which is also a kind of DAG, the Modi simulated DAG also has multiple layers, where leaf nodes form the input space, internal nodes extract higher level features, and output nodes corresponds to with class labels. Furthermore, it allows imbalance structure, shortcut (over-bipartite) connections, and non-full connections between neighbouring layers, which makes the representation much more flexible.

This structure also allows the *reuse* of child nodes. Every right most child of the Modi node is reused by the Modi node itself and the parent of the Modi node, resulting in a two-way reuse. Multi-way reuse is also possible by a sequence of hierarchically connected Modi nodes, as shown at the bottom right conner around the feature terminal node *W* in figure 3.

2.4 Modi Program Generation and Modi Rate μ

The generation of the Modi program tree is about the same as of the standard program tree, with three additional rules for Modi node generation: (1) All leaf nodes are not Modi since a Modi node requires a right child; (2) The root node is always Modi to guarantee that no part of the Modi tree is wasted; (3) For internal nodes, the probability of a node to be set to Modi is by a user specified number – the *Modi Rate* $\mu \in [0, 1]$, which can be interpreted as the expected percentage of Modi nodes over all internal nodes in programs of the *initial* population. Element indexes of the output vector are then assigned uniformly across all Modi nodes, with no extra controls. This may result in Modi programs that do nothing on some of the output vector element, with poor luck.

2.5 Summary of Modi Characteristics

The Modi program has two major properties: (1) It can produce multiple related outputs, thus classifiers of this kind can tell the class of the input object by simply voting as in neural networks. In this way, the complex translation from a single floating point value to multiple class labels can be avoided; (2) During the evolution, the Modi program structure is just like a standard program tree, thus still take the advantage of the neat evolution of standard tree-based GP.

3 Experimental Configuration

3.1 Image Data Sets

In the experiment, we used four data sets to provide object classification problems of varying difficulty, thus formed a reasonably thorough evaluation regarding Modi. Example images are shown in Figure 4.

The first set of images (Figure 4a) was generated to give well defined objects against a relatively clean background. The pixels of the objects were produced using a Gaussian generator with different means and variances for each class. Three classes of

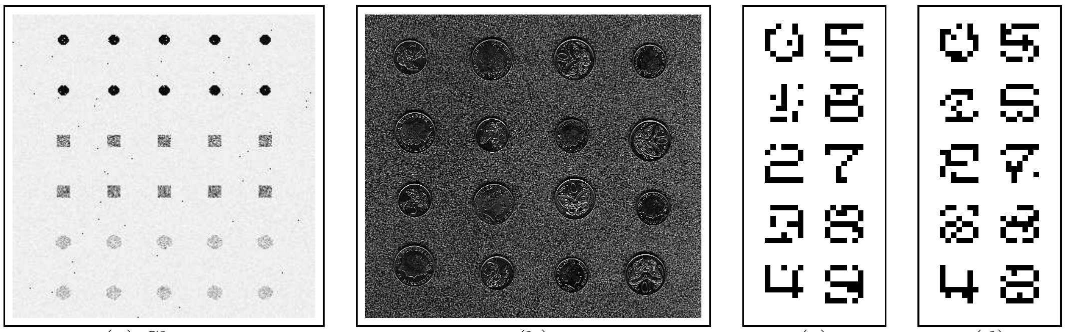
(a) Shapes (b) (c) (d)

Figure 4: Sample Data sets. (a) Shapes; (b) Coins; (c) Digits15; (d) Digits30.

960 small objects were cut out from those images to form the classification data set. The three classes are: black circles, grey squares, and light circles. For presentation convenience, this data set would be later referred to as *shape*.

The second set of images (Figure 4b) contained scanned 5 cent and 10 cent New Zealand coins. In this dataset, coins were located in different places with different orientations and appeared in different sides (head and tail). In addition, the background was cluttered. Thus all together the classifier would distinguish between different coins, with different sides, and from the background. Five classes of 801 object cutouts were created: 160 5-cent heads, 160 5-cent tails, 160 10-cent heads, 160 10-cent tails, and the cluttered background (161 cutouts). Compared with the *shape* data set, the classification problem of this kind would be much harder.

The third and the fourth data sets were of two digit recognition tasks, each consists of 1000 digit examples, with each of the digit example an 7×7 bitmap image object, as shown in figure 4 (c) and (d). In the two tasks, the goal was to automatically recognise which of the 10 classes (digits 0, 1, 2, ...,9) each pattern (digit instance) would belong to. Note that all digit patterns had been corrupted by noises. Further difficulty were added by the fact that 15% to 30% of pixels has been randomly flipped in the datasets. In data set 3, while some patterns could be clearly recognised by human eyes such as "0", "2", "5", "7", and possibly "4", it was not easy to distinguish between "6", "8"and "3", even "1" and "5". The task data set 4 was even more difficult — human eyes could even not recognise majority of the patterns, particularly "8", "9" and "3", "5" and "6", and even between "1", "2" and "0". In addition, the number of classes was much greater than that in tasks 1 and 2, making the two tasks get even harder.

For all the four data sets, the objects were equally split into three separate data sets: half for the training set used directly for learning the genetic program classifiers, and half for the test set for measuring the performance of the learned program classifiers.

3.2 GP System Customization

In our configuration, feature terminals consisted of four local statistical features extracted from the object cutout examples in the first two tasks, and just 49 pixel values in the third and fourth tasks. The function set consisted of the four standard arithmetic operators and a conditional operator. The division operator represents a "protected" division in which a divide by zero gives a result of zero. The conditional operator returns its second argument if its first argument is negative, and otherwise returns its third argument.

The ramped half-and-half method [1, 3] was used to generate the initial population and for the mutation operator. The proportional selection mechanism and the reproduction [8], crossover and mutation operators [1] were used in the learning process. We used reproduction, mutation, and cross over rates of 10%, 30%, and 60%, respectively. The program depth was initialised from 3-6, and can be increased to 7 during evolution. The population size was 500. The evolutionary process was run for a maximum of 50 generations, unless it found a program that solved the problem perfectly (100% accuracy), at which point the evolution was terminated early. Each group of experiments was repeated 50 runs and the average results are presented.

4 Results and Discussion

4.1 Object Classification Performance

The new Modi approach and the basic GP approach are compared under the same experimental

Table 1: Results of the new Modi approach over the basic GP approach.

Methods/ Improvement	Data Sets			
	Shape	Coin	Digit15	Digit30
GP with Modi (%)	99.77	93.89	68.11	54.46
Basic GP approach (%)	99.40	85.22	56.85	44.09
Improvement (%)	0.37	8.67	11.26	10.37

setting on four datasets described above. The best results are shown in Table 1. For the shape data set, both approaches did pretty well as the task is relatively easy. In particular, the Modi approach almost achieved perfect results. For the coin data set, as the task is harder than the shape one, the Modi approach achieved 93.89% accuracy, 8.67% higher than the basic GP approach. For two digit data sets, the Modi approach performed much better than the basic GP, with improvements of more than 10%. In particular, for task four, where even human eyes could only recognise a small part of the digit examples, the GP approach with Modi program structure can recognise majority of them, achieved 54.45% accuracy. These results suggest that the new Modi approach can perform better than the basic GP approach for object classification programs, particularly for relatively difficult tasks.

4.2 The Effect of Modi Rate μ

To investigate the effect of Modi rates on the performance of Modi patched GP, we did four groups of experiments on the four data sets using different Modi rates ranging from 0.0 to 1.0. The results are shown in Figure 5 in terms of the improvement of the Modi approach over the standard GP. It shows that the Modi rate does affect the performance. Its influence is not consistently proportional across different tasks, provided by that curves in the figure are not parallel with each other. However, neither too big nor too small Modi rates are the best. No reliable way of choosing a very appropriate Modi rate for a task has been found, except that to do empirical search through experiments. If such a search can improve performance significantly, it is a small price to pay. The experiments suggest that a Modi rate between 0.3–0.6 is a good point to start searching on.

4.3 Further Discussion

The result of the two digit datasets are worse than of the shape and the coin. This is mainly due to the difficulty of the classification problem, plus that the program size (program tree depth) was set to be too small. A too-small program size causes two problems: (1) The evolved program would not have enough leaf nodes for handling input features, particularly when the number of input features is large, say 49 for the digits. Modi's *reusability* helps on this problem, though cannot generally fix it; 2) The probability of producing sufficient Modi nodes so that they all together go across the entire output vector would be very small, particularly when there are a large number of classes to be classified, say 10 for the digits. This problem reveals a disadvantage of the Modi structure, that is, it could require a larger program size than the standard program tree structure.

We originally expected a small Modi rate from 0.0 to 0.1 would lead to very bad results. However, the effect of Modi rates was not as large as we expected. This is because we used a large size of population (500), so that the population as a whole provides sufficient Modi nodes for using by the *best* evolved program.

For multiclass classification tasks, the Modi approach converges much faster to the optimal solution in terms of the number of evolution used. For simple classification tasks such as the shape one, the Modi approach terminates on just a couple of evolutions, reflecting that the best program in the random initial population already very approaches to perfect. Therefore, apply the *No Free Lunch* [5] theory inversely, we theoretically conclude that the Modi approach is more appropriate on multiclass classification, which is consistent with our experimental result. However, the per evolution training time of Modi is generally slower than the standard GP, provided by that its simulation effect requires more steps to be done. This results in that in the real application, when both Modi and the standard GP are not able to terminate within the fifty generation limits, the Modi approach would take more time than the standard approach.

5 Conclusions

This paper described a new way of using the tree based genetic program structure in GP for multiclass object classification. This was done through the new program structure "Modi", which uses a *virtual* output vector to produce multiple values,

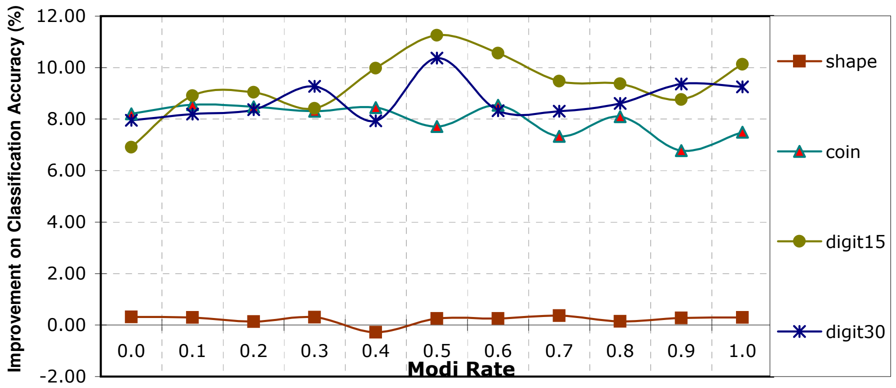

Figure 5: Effect of different Modi rates.

each of which corresponds to a class, instead of tree root node to produce a single number. The complex translation of the single number into different regions for multiclass was successfully avoided. This approach was examined and compared with the basic GP approach on four object classification problems with increasing difficulty. Results showed that the new approach outperformed the basic approach on all tasks.

The results also showed that different Modi rates led to different results. Neither too small nor too large Modi rates were good. However, it did not seem to exist a efficient and reliable way of choosing a good Modi rate for a particular problem. Rates between 0.3 − 0.6 seemed to be a good starting point to try.

Although developed for object classification problems, this approach is expected to be general and can be applied to other problems, wherever needs multiple outputs.

For future work, we will investigate the new approach for digit tasks with a larger program size and examine whether the performance can be improved. We would also like to investigate ways of using the Modi structure to evolve more general graphs such as Neural Networks, thus produce a kind of *Skinnerian creature* other than the pure GP based *Darwinian creature*. We will examine this approach on other multiclass classification problems and compare this approach with other long term learning techniques such as decision trees and neural networks.

References

[1] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. San Francisco, Calif. : Morgan Kaufmann Publishers; Heidelburg : Dpunkt-verlag, 1998. Subject: Genetic programming (Computer science); ISBN: 1-55860-510-X.

[2] Daniel Howard, Simon C. Roberts, and Richard Brankin. Target detection in SAR imagery by genetic programming. *Advances in Engineering Software*, 30:303–311, 1999.

[3] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Mass. : MIT Press, London, England, 1994.

[4] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[5] D.J.C. MacKay: *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press (2003).

[6] Will Smart and Mengjie Zhang. Classification strategies for image classification in genetic programming. In Donald Bailey, editor, *Proceeding of Image and Vision Computing Conference*, pages 402–407, Palmerston North, New Zealand, November 2003.

[7] Mengjie Zhang, Peter Andreae, and Mark Pritchard. Pixel statistics and false alarm area in genetic programming for object detection. In Stefano Cagnoni, editor, *Applications of Evolutionary Computing, Lecture Notes in Computer Science, LNCS Vol. 2611*, pages 455–466. Springer-Verlag, 2003.

[8] Mengjie Zhang, Victor Ciesielski, and Peter Andreae. A domain independent window-approach to multiclass object detection using genetic programming. *EURASIP Journal on Signal Processing, Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis*, 2003(8):841–859, 2003.

[9] Mengjie Zhang and Will Smart. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing*, volume 3005 of *LNCS*, pages 367–376, Coimbra, Portugal, 5-7 April 2004. Springer Verlag.