A Hybrid Neural Network and Genetic Programming Approach to the Automatic Construction of Computer Vision Systems

Cameron P. Kyle-Davidson

A thesis submitted for the degree of Master of Science by Dissertation

School of Computer Science and Electronic Engineering
University of Essex

October 2018

Abstract

Both genetic programming and neural networks are machine learning techniques that have had a wide range of success in the world of computer vision. Recently, neural networks have been able to achieve excellent results on problems that even just ten years ago would have been considered intractable, especially in the area of image classification. Additionally, genetic programming has been shown capable of evolving computer vision programs that are capable of classifying objects in images using conventional computer vision operators. While genetic algorithms have been used to evolve neural network structures and tune the hyperparameters of said networks, this thesis explores an alternative combination of these two techniques. The author asks if integrating trained neural networks with genetic programming, by framing said networks as components for a computer vision system evolver, would increase the final classification accuracy of the evolved classifier. The author also asks that if so, can such a system learn to assemble multiple simple neural networks to solve a complex problem. No claims are made to having discovered a new state of the art method for classification. Instead, the main focus of this research was to learn if it is possible to combine these two techniques in this manner. The results presented from this research indicate that such a combination does improve accuracy compared to a vision system evolved without the use of these networks.

Acronyms

ANN Artificial Neural Network.

CGP Cartesian Genetic Programming.

CNN Convolutional Neural Network.

DRS2 Dynamic Range Selection 2.

EA Evolutionary Algorithm.

ES Evolutionary Strategy.

GA Genetic Algorithm.

GP Genetic Programming.

LGP Linear Genetic Programming.

LSTM Long Short-Term Memory.

MLP Multilayer Perceptron.

ROI Region of Interest.

STGP Strongly Typed Genetic Programming.

Acknowledgements

I would like to thank Dr. Adrian Clark for his continued support throughout the course of this research project, without which I am certain that I would not have got nearly as far.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Forms of Machine Learning

Machine learning is a term that shows up nearly everywhere nowadays, among both the scientific and industrial communities. Those words can be found advertised on the products of data science companies, attached to smart phones, integrated into online music services, and even robotic vacuum cleaners can now remember objects for future avoidance. However broadly this term is applied though, it means the same thing: that somewhere in this product there exists the results of an algorithm that was capable of 'learning' - i.e. capable of improving its own performance on a task without being specifically programmed with the knowledge of how to perform that task.

The machine learning field can be generally divided into two subfields that describe how the algorithm learns, and what kind of data it is capable of learning from. One of these subfields is supervised learning, which requires a dataset that consists of a set of inputs with ideal outputs paired with them. This labelling of the training set is necessary for the algorithm to learn, which restricts the input to data that has already been seen and annotated, usually by a human. The goal of the algorithm in this case is to find a method which converts the input into the desired output. Support vector machines, Genetic Algorithms (GAs) (Chapter 2), and most forms of neural networks make use of this method (Chapter 3).

The other subfield is unsupervised learning, in which there is no labelled data, and hence no target outputs from a given input. In this case, the algorithm learns a technique that describes some structure in the dataset. Clustering, and some less common types of neural networks, are examples of an unsupervised learning technique.

Both these subfields have experienced surges of progress in recent years, and the gradual increase of artificial intelligence in our day to day life seems unlikely to slow any time soon. This is not to say that the field is without difficulties. Machine learning algorithms tend to be good at the single thing they are trained to do. Generic systems capable of learning to tackle multiple different tasks remain difficult to create, and in general, both supervised and unsupervised learning algorithms seem to require increasingly large datasets for adequate results.

1.2 Computer Vision

One aspect of artificial intelligence is computer vision. Computer vision essentially boils down to equipping computers with the necessary machinery to see (and understand) the world that they are embedded in. Computer vision techniques are ubiquitous today, from the face recognition software that is almost certainly standard in smartphone cameras (and in newer variants, as of 2018, object recognition), to self-driving cars.

Unfortunately, the more capable the vision system, the more training data it usually requires. For larger systems, this can easily amount to hundreds of thousands of images, or thousands of hours of video. And this is just to solve one specific problem, simplistic to humans, such as ‘recognise a bird’. It is not surprising in this case that the system capable of recognising birds cannot then be used to identify cars. This is a limitation that most humans do not suffer from.

The ideal goal is that of a system similar in capability to the human visual system. The human visual system can identify many different objects, and critically, is capable of learning to recognise a new object using very little ‘training data’. Perhaps humans can identify new objects so quickly by drawing on their extensive knowledge of other, similar objects. But how might this ability be transferred to a machine?

1.3 Motivation and Goal

The primary aim of this research is to determine if the superior pattern recognition and classification of pre-trained neural networks can be used by a GA. Genetic Algorithms have been used before to evolve vision programs capable of classification, but in practice the author found these algorithms have difficulty classifying multiple classes in a scene. It has also been shown that it is possible to retrain an evolved program to tackle a different problem to varying degrees of success [3]. However, such algorithms do not benefit from prior knowledge, a key difference between most machine learning algorithms and the human visual system. Shown an image containing a tree and a person, humans do not need to relearn what a tree or a person is; previous knowledge can be applied and combined.

The goal is to show that by providing a GA with multiple pre-trained neural networks, and having it select only those relevant to completing the task of the algorithm, that the accuracy of an evolved vision system can be improved. With this method, the GA will not have to re-invent the wheel for every task, and can benefit from existing knowledge.

Additionally, instead of the GA learning to use a single complex neural network capable of classifying multiple classes, it would be ideal if the algorithm could piece together different smaller networks capable of classifying one class, into a system capable of identifying multiple items. Such a system would mean that the neural networks provided could be much simpler and trained much faster, without sacrificing classification accuracy.

Ultimately, the desire is to advance the capabilities of evolved vision software, and answer the questions:

Can neural networks be integrated into a GA that evolves object classifiers in a manner that increases the accuracy of the evolved system?

And if so, can we use much simpler networks in the confidence that the GA can assemble them for accurate classification of many regions in one image? This is discussed in yet more depth in Chapter 4

1.4 Challenges

One of the main challenges in this research will be tuning the genetic algorithm used to evolve the vision system. Specifically, GAs rely heavily on the fitness function that evaluates the suitability of a solution for a problem. A poor fitness function will return poor solutions. It is also not necessarily clear when a fitness function is deficient, versus other potential issues, such as lacking the necessary expressiveness required to solve the problem. (For example, a GA that is instructed to draw a square will never solve this problem if only given a tool that can draw circles). The fitness function also tends to be computationally expensive (and several improvements to the existing systems fitness function are shown in this project in this regard).

GAs also do not cope well with increased complexity. An increase in parameters causes a corresponding exponential increase in the size of the search space. In this research, this is somewhat mitigated by inclusion of neural networks, as the effectiveness of several possible components can be condensed into one.

Finally, because the GA is provided with another form of machine learning, this also comes with any error inherent in that method of machine learning. In the case of neural networks, this means that a poor performing network could compound an already existing error in the GA. However, with a suitably tuned fitness function, poor performing networks should not be selected by the algorithm, as individuals that utilise it would not be considered fit.

These challenges and some solutions are discussed in more depth in Chapter 4.

1.5 Work Conducted

The following is an outline of the work conducted during the course of this research project, including a novel contribution.

- Developed two standard genetic programming cost functions for the purpose of image recognition that improve on DRS2, by moving away from per-pixel based classification of the input image towards per-region based classification (page 57).
- Extracted structural images from a convolutional neural network via gradient ascent

(page 62), and a component that uses structural comparison of these images for the purposes of object classification was designed (page 65). The developed component was then tested on a toy dataset to determine efficacy of this technique (page 88).

- Implemented high level feature descriptors such as area, perimeter, aspect ratio, and others, to allow a genetic programming system to evolve programs that can store information about regions of interest using these features (page 66), as well as components that can classify regions of interest using Boolean operators over the stored information (page 67).
- Implemented binary decomposition for image classification problems (page 69).
- Created a final cost function that supports binary decomposed classification problems, as well as being simpler to implement while also handling the skewed dataset that arises from binary decomposed problems (page 70).
- Integrated multiclass neural networks as components for the genetic programming algorithm (page 73), and evaluated this combined system over subsets of the MNIST and Fashion MNIST datasets (page 94).
- Integrated binary classification (decomposed) neural networks trained on less data for shorter amounts of time as components (page 75). This combined system was evaluated over relevant subsets of the MNIST and Fashion MNIST datasets (page 6.6).
- The work on integrating multiclass neural networks and binary classification (decomposed) neural networks as components for computer vision systems evolved via genetic programming, and the results of such a system over the MNIST dataset, were published in [4]

1.6 Thesis Overview

This thesis covers three main areas: Genetic Algorithms, Neural Networks, and the combination of Convolutional Neural Networks (CNNs) with Genetic Programming (GP), with the overall goal of creating a system capable of evolving vision systems that can classify regions in images using neural networks.

Chapters 2 and 3 provide a detailed overview of the current state of the art of GAs and neural networks, two techniques that are the main focus of this research. These chapters look at how these algorithms work, the variants of these algorithms that have been developed over the years, the motivation for choosing these algorithms, and the limitations that these approaches to machine learning have. Chapter 2 focuses more on genetic programming, which is the approach used in this research, but Chapter 3 remains more general, as theoretically any neural network could be integrated into the developed system.

Chapter 4 looks at previous work on combining GAs with neural networks, and also introduces the method implemented in this research, and why theoretically this may give superior performance that using an evolved vision system that does not have neural networks in its arsenal of tools for classification.

Chapter 5 describes the implemented system and the steps taken to reach the final system. It is in this chapter that the actual integration of neural networks with genetic programming is described, as well as the development of a suitable fitness function. An experiment based around extracting filters from convolutional neural networks and applying them directly to the GA is examined. This chapter also details the implementation of a binary decomposition system as well as a higher level feature extractor and classifier, based on previous work.

Chapter 6 explains the experiments run to test the individual parts of the system as the research progressed and it developed, as well as a set of experiments based around the final system. The results of the experiments are analysed and the effectiveness of each different method compared. The overall effectiveness of the integrated neural networks is evaluated against the stated goal of improved effectiveness of an evolved vision system.

Finally, Chapter 7 concludes the thesis by assessing the entirety of the research, including the advantages and disadvantages of this technique, and looks to the future research that could be done based on what has been achieved so far.

CHAPTER 2

Overview of Evolutionary Algorithms

Evolutionary Algorithms (EAs) are not a new machine learning technique and have existed in some form for nearly seven decades. The idea of a machine evolving via mutations and survival of the fittest was first hypothesised by Turing [5]. Over the years the implementation has varied [6], but the core idea remains the same; they are a class of algorithms loosely based around the theory of evolution. In general, evolutionary algorithms are made up of populations of individuals (which may be represented in a myriad of ways, be this programs, or mathematical equations, or something else entirely), each which attempt to solve the given problem. They have a function which can evaluate how well a given individual solves the problem, and a method to select the best individuals from the population. Individuals can 'breed' with each other to create offspring using techniques inspired by biological processes, such as gene recombination and mutation [7]. The overall goal for an initial population of an evolutionary algorithm is, much like in reality, that the fittest individuals survive and have more offspring, and the weaker individuals die out, with the eventual hope that after so many generations of evolution a best individual gives a good solution to the given problem.

Evolutionary algorithms have been used in many different industries, and often produce solutions that a human may never have thought of. They have been employed to design antennas [8] for spacecraft [9], to design circuit boards [10], and to develop walking gaits [11] for robots. These examples all share a common theme in that they are optimisation problems. Hence solving them requires finding the a solution that is good enough to address the problem, which is a natural fit for an evolutionary algorithm!

In this chapter, a few different types of evolutionary algorithms are examined, and the key algorithm used in this research, typed genetic programming, is described in detail.

2.1 Evolutionary Algorithm Operations

There are three main genetic operators that are used in most evolutionary algorithms. They are **selection**, **crossover** and **mutation** [12]. In operation, they are similar to, (and inspired by) their real life biological counterparts. While the theory behind the operators remains the same despite the EA being used, in practice the implementation varies. Theoretically, crossover and mutation techniques can be implemented in many different ways, limited only by the creativity of the implementer, though in practice there are a few standard implementations for each common form of EA.

2.1.1 Selection

Selection is the method with which the surviving individuals are chosen from the population and used to create a new set of individuals. By doing this, the evolutionary algorithm can explore the search space of the problem, and hopefully approach the global optimum.

There are several different methods of choosing the best individuals, described below.

2.1.1.1 Fitness Proportionate Selection

In fitness proportionate selection, the fitness of all individuals in the population is evaluated, and this fitness value is used to assign a probability to that individual of it being chosen from the pool of individuals for breeding. The more fit the individual, the more likely it will survive to pass on its genes. Fitness proportionate selection tends not to be used in favour of other algorithms that are easier to implement, and have lower noise. Fitness proportionate selection also performs poorly when some individuals in the population have a very large fitness compared to other members in the population [13] [14].

2.1.1.2 Stochastic Universal Sampling

Stochastic universal sampling is a variant of fitness proportionate selection that guarantees an individual be selected as many times as probability indicates it should be selected, on average. Hence, an individual with a 10% chance of being selected from the population, and where 100 individuals are selected, that individual would be selected ten times.

2.1.1.3 Truncation Selection

Truncation selection is a simple method of selection where a certain proportion of the of the best individuals in the population is selected. The population is ranked in terms of fitness, and a chosen amount (e.g, top half, top third, top quarter, and so on) of the top ranks are picked to reproduce.

2.1.1.4 Tournament Selection

Tournament selection is where individuals are chosen randomly from the population, and compared against other individuals in a series of tournaments until only the best individual remains from those randomly selected. This individual then is allowed to reproduce.

2.1.1.5 Rank Selection

Rank selection is similar to fitness proportionate selection, except each individual's probability of being selected is based upon that individuals overall fitness rank (i.e. the population is ranked according to individual's fitness) in the population, not the fitness value itself. This also works with negative values, and has the advantage that it performs well when fitness values are close amongst the population.

2.1.1.6 Which is the Best?

Which selection method is the best? In practice, like functions in many machine learning algorithms, the answer is really 'it depends'. The best selection method to choose is the selection method that gives the best results for the chosen problem!

2.1.2 Genotype and Phenotype

Two frequent terms are used when referring to evolutionary algorithms: genotype, and phenotype. Much like other terms used in this field, they acquire their meanings from their biological equivalents.

The genotype of an individual is its representation, i.e. what the individual is actually composed of. This could be a string, an array of 'genes', a program tree, or anything else! In biology, the genotype of an organism is the genes that it carries.

The phenotype of an individual is the individual's behaviour. In an evolutionary algorithm individual, this is what the individual actually **does**.

When the individual's fitness is evaluated, usually only the individual's output is significant, which is equivalent to testing the individual's phenotype.

2.1.3 Fitness Evaluation

In an evolutionary algorithm, the fitness of an individual must be calculated and returned. This is done by some form of 'fitness function'. A good fitness function is important for overall good performance of the algorithm, and a bad fitness function can mean that the algorithm never finds a solution. The fitness function can be anything that appropriately tests the output of an individual and compares it to the desired output. For example, if the desired output of an evolutionary algorithm is the integer '5', then a suitable fitness function could take the integer output of an individual, calculate the Euclidean distance from '5', and use this as the fitness. For a minimisation problem, the ideal distance, and hence fitness, is zero.

2.1.4 Crossover

Crossover produces a new individual from parent individuals. The genotype of the parents is combined by selecting a point, and switching the genotype of one parent at that point with the genotype of the other parent at that point. Often this point is selected randomly.

Crossover can be performed with only one point, with two points, and generalised to n-point crossover, where n represents the number of points in the genotype that is switched between the parents. A technique called uniform crossover [15] also exists, where crossover is performed for each individual part of the genotype. For example, in an individual where the genotype is represented by an array of bits, crossover would be performed for each entry in that array. Which parent to take the bit from can be selected with equal probability, or selection of the parent can be biased one way or another [12].

2.1.5 Mutation

Mutation is where a part of the genotype is replaced by a new, usually randomised value. This creates a new individual, but only requires a single parent with which to do so. This operator is inspired by biological point mutation, where a single gene is mutated, and in the genetic operator, this remains the same. For example, in a genetic algorithm that utilises bit arrays as a genotype, each bit has some probability of mutating (flipping). Similar implementations exist for other evolutionary algorithms dependent on genotype format. For example, if the genotype consists of real numbers, the value may be replaced by any number between two bounds to a certain degree of accuracy. Mutation is used to increase the diversity of the genetic population and prevent individuals from having genomes that are two similar to each other, which in turn helps to prevent the evolutionary process from becoming stuck in a local minimum [16].

2.2 Types of Evolutionary Algorithms

Several implementations of evolutionary algorithms have been developed, mostly differing in the method with which the individual is represented and which genetic operations are available. However, they all have the same core loop. First, an initial population of individuals is generated, usually randomly. Then the fitness of each individual in that population is evaluated via a fitness function tuned for the problem. Then the following steps are performed until a good solution is found or the algorithm is stopped:

1. Select the best individuals in the population to become ‘parents’.
2. Create new individuals from the parents by using genetic operations such as mutation or crossover (recombination).
3. The fitness of the new individuals is calculated.
4. The worst performing individuals are replaced by the new ‘children’.

A couple of the more notable evolutionary algorithms are described below, and genetic programming is described in Section 2.3.

2.2.1 Evolutionary Strategies

Evolutionary Strategies (ESs) represent some of the earliest work into evolutionary algorithms, predating genetic algorithms by roughly a decade, with the first evolutionary strategy appearing in the early 1960s. In practice, they are very similar to real valued genetic algorithms, with the main difference lying in the choice of primary genetic operator. In genetic algorithms, this is crossover, and in evolutionary strategies, mutation. [17]

Evolutionary strategies also tend to have fewer individuals in their population, in the simplest case, only two. Mutation is applied, the best individual selected, and the worst performing individual is replaced. However, many variations on this theme exist, varying in population size, and number of individuals replaced. Evolutionary strategies generally fell out of favour as an evolutionary algorithm, until interest in their potential applications to reinforcement learning arose. Recently it has been shown that modern ESs can rival the performance of neural networks for reinforcement learning [18], and evolutionary strategies in general are experiencing a resurgence.

2.2.2 Genetic Algorithms

Genetic algorithms inherit all the properties that evolutionary algorithms have, but are defined by how each individual is represented. In a genetic algorithm the individuals are represented by arrays of numbers. Usually these numbers are integers, but they may also be floats, in the case of 'real valued genetic algorithms'. In both cases the arrays are a fixed size for all individuals, which simplifies the crossover (recombination) operation. Crossover and mutation in integer genetic algorithms often use some form of n-point crossover, or uniform crossover [19]. As integers are often represented as binary, mutation is easiest to implement as a simple bit flip.

In real valued genetic algorithms these options remain, with the additional choice of using interpolated crossover and mutation. This is possible because the genotype is real-valued and hence well suited to interpolation between parents values. The new child gene is a random number that lies somewhere between the two parent genes. In the case of mutation, the new gene is a random number some distance from the starting value. This is known as the BLX-α algorithm [20].

One can create as many new types of crossover and mutation as there exists methods to combine two different numbers. The above are mentioned as they have been used before and appear to work well.

2.2.3 Neuroevolution

Neuroevolution is using evolutionary algorithms to evolve and tune neural networks. A recent addition to the field of machine learning, exploration of this technique and a review of the notable research in this area is given in Chapter 4

2.3 Genetic Programming

Genetic Programming is where a genetic algorithm evolves computer programs that solve a specific problem. The idea of evolving computer programs was first proposed in the 1950s, but the first practical implementation came several decades later, designed by John Koza, and research in this field continues to this day. In many cases, GP has been shown to be able to evolve computer programs competitive with human-created programs [21].

There are several implementations of genetic programming. The most common implementation represents individuals as tree structures, because such structures lend themselves towards easy evaluation of the evolved program. Because of this, the programming languages used for evolved programs tend to be those that lend themselves naturally to tree structures, such as languages designed for functional programming (Lisp is also frequently used) [12]. A few other implementations of GP are briefly described in subsections 2.3.3 and 2.3.4

2.3.1 Fitness Evaluation

Fitness evaluation for genetic programming simply runs the program and compares the output to some metric. The actual representation of the program itself (its genotype) does not matter for the purposes of fitness evaluation.

2.3.2 Population Instantiation

To create the initial population that serves as the seed for all future individuals in the population pool, John R. Koza describes three potential methods for generating random trees, assuming the presence of valid terminal and function sets, as well as a maximum depth that constrains all trees to equal to or less than the given depth [7].

The three methods are the **grow** method, the **full** method, and **ramped half-and-half** method.

The grow method starts from the root node and then determines if that node requires any children. The children are then randomly selected, and the process is repeated until either a terminal node is selected as a child node, or the maximum depth of a tree branch is reached, and a terminal node is selected to finalise that branch.

The full method works in a similar manner to the grow method, except it ensure that all branches in the program tree extend to the maximum depth. However, this method restricts the search space to only considering trees that are 'complete' and hence may result in an optimal non-full solution being missed.

Koza then proposed a method combining these two previous tree builders, the 'ramped half-and-half' method. In this method, half the initial population is created using the grow method, and half is created using the full method, and the population is built using an increasing depth parameter until the max depth is reached. This generates initial program trees of many different shapes and depths.

2.3.3 Linear Genetic Programming

In Linear Genetic Programming (LGP), individuals are not represented by a tree structure, but are instead represented by a sequence of instructions. This means that the evolved programs can be written in imperative languages, rather than the declarative languages favoured by tree structure programs [22].

However, during evolution linear genetic programming creates blocks of ineffective code, known as introns. Introns biologically are genes that appear to do nothing, and have no effect on the phenotype. In genetic programming, this definition remains the same. They are essentially 'dead' blocks of code, that have no effect on the output. Introns are not desirable as they increase program length and complexity without increasing fitness. However, LGP does often allow fast and efficient removal of the same introns it tends to generate [23].

This approach to genetic programming is suited to mathematical modelling where the use of an imperative language is desired. LGP can also result in programs that are faster to execute than tree based genetic programming as individuals can be represented directly in machine code. In such a case, understanding the output of the evolution system directly is difficult unless one is fluent in binary [24].

2.3.4 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) represents programs as connected graphs, and tends to use an evolutionary strategy for fitness improvement rather than a genetic algorithm. The genotype of this approach is composed of integers that are reference either an entry in a look-up table that contains functions available to the individual, or to data in available memory.

This approach has been shown to be competitive with other kinds of genetic programming, and in particular is well suited to evolving programs that have multiple outputs, making it suitable for evolving circuitry or neural networks, and in fact is one of the more modern evolutions of genetic programming [25].

This technique was recently used by Wilson et al. [26] to evolve programs capable of playing atari games, achieving competitive performance with other artificial agents. Wilson et al used a floating point representation of CGP, and used mainly mathematical, statistical and list-processing functions.

Cartesian genetic programming could have been used for this research project, but re-implementing a vision system evolver using this technique rather than standard genetic programming would be an entire project in itself, and additionally, multiple output evolved programs are not required for this project.

2.3.5 Structure, Crossover, and Mutation

In tree-based genetic programming, the system has access to both a set of **functions** and **terminals**. These sets define what kind of programs the genetic algorithm is capable of evolving [12], and are effectively the 'components' that the genetic algorithm builds the evolved programs out of. If the problem cannot be solved by any combination of these functions or terminals, then genetic programming cannot evolve an individual to solve the problem. This means that good selection of the members of these sets correlates with how well genetic programming can solve the given problem. Of course, it can be difficult to tell what kind of terminals or functions lend themselves to the problem, so the selection step often comes down to human heuristics ('a directed guess').

The functions are as expected, a set of 1 to n -arity functions that return a value. An example of such functions could be an ‘even’ function that returns if the input number is even, or a 2-arity function that adds two numbers. These functions become the nodes in the tree structure.

Terminals are usually a set of initial starting values, such as useful integers, floats, images, or any other kind of data. 0-arity functions may also be included in the terminal set. These values and functions make up the leaves in the program tree, and feed into the functions.

Crossover in a tree structure is slightly more complicated that of a genetic algorithm. The two parents may either exchange an entire subtree (selected at a random point, and grafted onto the other parent at a random point) or they may exchange an individual node or terminal. As these trees represent programs, in standard genetic programming, this runs the risk of creating an invalid individual. For example, if a terminal that represents a string is grafted onto a function that expects an integer, or a subtree that returns one value is grafted onto a 2-arity function, the child is no longer valid and would immediately ‘die’ [27].

Mutation in tree-based genetic programming selects either a terminal or a function, and randomly changes it to a different terminal or function. This has a similar problem to crossover in that an invalid program may result [27].

This is inefficient unless all functions and terminals are compatible, and the more complicated the functions and terminals, the less likely this is. It would be preferable to only pair functions and terminals with other functions that can accept their output. This idea gives rise to **Typed Genetic Programming**..

2.4 Typed Genetic Programming

Few real world programs have only one data type used across the entirety of the program. Imagine writing a web server where the only variable type that could be used are integers. Likely possible, but also much more complicated than using strings, floats, and Booleans in addition to integers.

This is the pitfall that traditional genetic programming falls into, expecting that any function (a component that can accept values as arguments) is capable of accepting any kind of data returned from a terminal. The easiest way to implement this is to restrict inputs and outputs to a single data type, and rewrite terminals and functions to only use this data type. For example, Boolean data types can be replaced by integer data types that specify a certain value to be true and a certain value to be false. This was the approach used by Koza in his initial exploration of genetic programming [7]. This approach is limited, however, and it becomes increasingly difficult to rewrite a problem such that it can be solved with only one data type as the complexity of the problem increases. Eventually, a sufficiently complicated problem becomes impossible to solve with only one data type, such as the problems explored by Montana [28].

Montana also demonstrated a method to avoid the data type problem by making use of dynamic typing. Each function is designed to either accept any argument type and attempt to cast the input into mutually intelligible types, or throw an error if the types are inconsistent and mark that section of the individuals parse tree invalid. However, this approach falls flat if the data types cannot be cast between each other. For example, what behaviour should occur when you multiply a string by a matrix? While such behaviour could be manually defined, this adds significant complexity depending on the amount of data types implemented, and interactions that require defining. Although, interestingly, this approach appears to have been used successfully by Wilson et al. [26] to deal with mixed scalar and matrix inputs, indicating it remains suitable for problems with limited data types.

Throwing an error if the input types cannot be operated on together is also possible, yet Montana writes that this leads to inefficient evaluation, as most individuals generated by the genetic algorithm are invalid. The search for a better solution to the problem of mismatched data types eventually culminates in Montana's main contribution to the field of GP: strongly typed genetic programming.

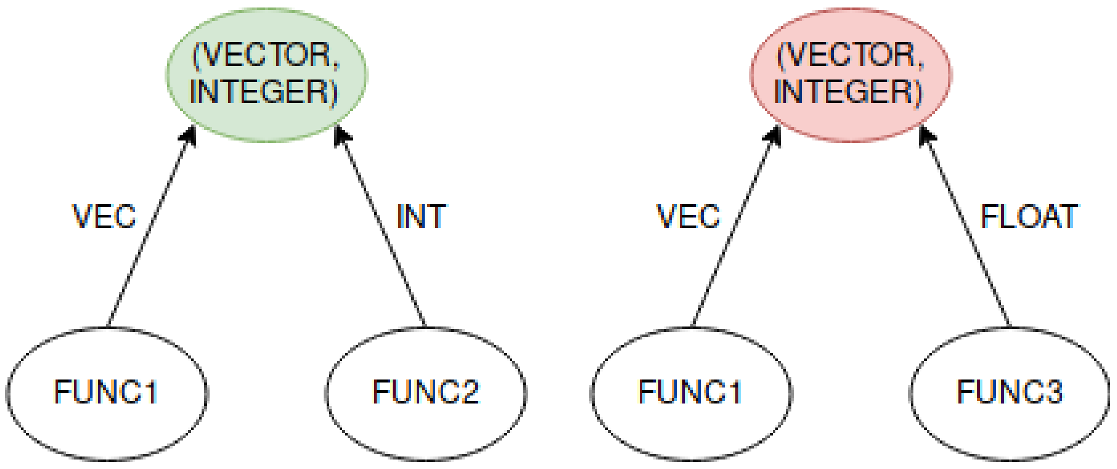


Figure 2.1: An example of a valid and an invalid strongly typed program tree

2.4.1 Individual Representation

In Strongly Typed Genetic Programming (STGP) all terminals have an assigned type, and each function has types that it can accept, and a type that it can return. Beyond this, the representation is very similar to the representation in general genetic programming, with two important exceptions. The first being that the output (root) node of the program tree returns a value of the type required to solve the problem (e.g. a program to solve combinatorial problems may return an integer or float, but not a Boolean) and the second exception is such that all nodes with children must take as input types the output types of their children.

2.4.2 Building the Initial Population

Creating the initial population remains similar to how an initial population is built in traditional genetic programming. The initial program trees are generated randomly, but rather than any terminal may be attached to any function in the program tree, in typed genetic programming, only terminals and functions with mutual types can be paired together.

This means that the programs generated by typed genetic programming are naturally less random than untyped genetic programming, but all initial programs represent valid programs. This constraint may also mean that certain programs of a certain depth cannot be generated validly, however. This caveat is dependent on the available functions and terminals, and their types. For example, if the output type is a float, all terminals are integers, and there exists a sole function that takes an integer and returns a float, then a program tree of depth two cannot be generated. In practice, this makes little difference to the population. Such a tree would still be invalid in traditional genetic programming. The advantage of typed genetic programming is that it does not have to spend processor time evaluating a useless program, as it will not be generated in the first place.

2.4.3 Fitness Evaluation

Fitness evaluation remains the same between traditional genetic programming and strongly typed genetic programming, as fitness evaluation evaluates the phenotype of the program, that is, its behaviour, rather than its representation.

2.4.4 Crossover and Mutation

Crossover and mutation also remain closely related to traditional GP, with the only difference being that the crossed-over subtrees and nodes must have matching types to their insertion point.

This constraint exists for mutation, in that nodes can only be mutated into nodes of the same type.

This prevents parents from creating invalid children, and the genetic population likely dying out, as invalid combinations typically outnumber valid combinations.

2.4.5 Performance

Montana found that STGP significantly reduces the search space that must be explored to return a solution for a given multiple data type problem. Montana describes good results for strongly typed genetic programming when applied to the problems of Multidimensional Least Squares Regression, Kalman filter modelling, and two cases where the typed genetic programming system attempted to model two Lisp functions (NTH and MAPCAR).

NTH returns the value in a list, L, at position N, and MAPCAR takes a function F and list L, and returns list Q where Q is the result of applying F to every element in L (i.e. the Lisp equivalent of a common map function).

In all cases the typed genetic programming system could find a good solution, though an optimal solution was not found for all problems due to the increase in complexity requiring a corresponding, fast, increase in required computation (a drawback of genetic programming).

Haynes et al. [29] applied STGP to the problem of evolving cooperation strategies. He noted that one of the main problems with traditional genetic programming is the large search space that must be searched for complex problems, and notes that even with minimal terminal and function sets, and low maximum generated tree depths, the search space of traditional genetic programming can be around 10^{40} possible solutions! Hence, Haynes et al employ typed genetic programming to reduce this search space to a more manageable size. He found that typed genetic programming outperforms traditional genetic programming in all cases, and that the generated programs were competitive with human designed algorithms for simulated cooperative strategies.

In general both Montana and Haynes et al found that STGP is superior to traditional genetic programming, and additionally outperforms random search of all potential solution trees. The drawbacks of this approach is essentially identical to the main drawbacks of all evolutionary algorithms, described in Section 2.5

2.5 Limitations and Drawbacks of Evolutionary Algorithms

Evolutionary algorithms tend to have a few noticeable limitations common among all implementations. One of the most significant drawbacks is that the overall performance of the algorithm depends upon a well written, and efficient fitness function. If the the fitness function does not suit the problem, or the solutions to the problem are hard to define programmatically, then the evolutionary algorithm is unlikely to be guided towards the optimum solution. Additionally, the fitness function for complicated problems tends to be expensive to compute, and hence the more complex the problem the slower the evolutionary process becomes.

Evolutionary algorithms in general do not to scale well with the complexity of the problem. The more complex the problem, the larger the potential search space becomes [28], and the more computation is required to find an optimum.

Another limitation of evolutionary algorithms is that it is possible for evolution to become stuck in a local minimum rather than finding the true global optimum for the problem [30]. Once in a local maximum, it becomes difficult for the evolutionary algorithm to leave, as all potential further solutions result in a decrease in fitness, even though this may bring a greater increase in fitness later. Evolutionary algorithms generally do not have the capability for short term loss in order to attain a better fitness in the future.

2.6 Evolving Vision Systems

In 2009, Oeschle et al. [31]. demonstrated that it is possible to use genetic programming to evolve segmentation and classification programs. These programs, when ran against images, were able to segment regions of interest in the image and then attempt to classify the region of interest based on the classes the genetic algorithm has seen during the training process. Using genetic programming for image analysis is not a new idea, with Tackett' [32] developing a system that can evolve separators for IR data in 1993, followed by Poli et al. [33] in 1996 who developed a method that uses genetic programming to evolve feature detectors for segmentation of items in images, and in 2004, Song et al. [34] demonstrated a method of evolving accurate classifiers that work on the textures present in images. Oeschle built upon all of these by developing a system that can evolve vision systems, both segmenters and classifiers, for many different kinds of input images, automatically. It was the first step towards a generic vision system, created mostly automatically using genetic programming.

Oeschle developed a system known as 'JASMINE', which was capable of using human-labelled training images, combined with human adjustment during the training phase (such as fine-tuning the marked regions of the training data), to create programs that can identify objects in images. This system was tested against various different data sets, such as pasta shapes and hand gestures, generally achieving a 90% accuracy rate or above, even when tested against the MNIST dataset, which is composed of hand-drawn digits. This was achieved by providing the genetic algorithm with a vast array of image feature selectors, capable of feature extraction.

The fitness function used to achieve this was an improvement to an algorithm known as 'Dynamic Range Selection' (DRS) [35]. The genetic classifier examines an input image, and for each pixel, produces an output. This output is then assigned an index in an array. Dynamic Range Selection then assigns the most common class label for that output, based on the training data which led to that pixel being assigned to that index. The improvement Oeschle et al made to this algorithm was removing the upper and lower bounds on the possible indexes, allowing any variance in output. This was found to help prevent the evolved classifier from needing to learn to scale its own output unnecessarily, and in general increased performance of the genetic algorithm.

The genetic programming system used in this research project (known as 'GPT') is based upon the research of Oeschle et al, with certain modifications and simplifications, and is the system that this author is intending to improve. GPT is further described in Chapter 5

2.7 Conclusion

While it is intractable to describe every kind of evolutionary algorithm, as countless exist, in this chapter a general overview of evolutionary algorithms and an in-depth look at genetic programming and, specifically, typed genetic programming has been provided. Generally, evolutionary algorithms provide a useful way to solve optimisation problems automatically, and genetic programming allows the harnessing of this ability to automatically create computer programs. Despite the limitations of evolutionary algorithms, it has already been shown that computer programs capable of classifying objects in images can be generated. However, in order to improve this ability, one needs to turn ones attention towards a field that has been the well-deserved focus of the machine learning community for over a decade now: the Artificial Neural Network (ANN).

CHAPTER 3

Overview of Neural Networks

Like evolutionary algorithms, neural networks are inspired by biology. But rather than taking cues from abstract representations of evolution, neural networks are instead modelled after the best learning system available: the brain.

Because neural networks are actively being used and developed today, it is easy to think of them as a recent invention, but this is not the case. Artificial neurons were first developed (if only on paper) in the early 1940s by McCulloch and Pitts.

This initial idea developed over the next few decades into something more similar to the neural networks that are known today, only to stagnate in the late 60s as the computer processing required to run a network outstripped the available resources at the time. It wasn't until a decade later that the key to overcoming this issue was discovered, discussed more in Section 3.3.

Today neural networks are one of the key pillars of modern machine learning, and the technique has experienced numerous successes in areas that previously were declared intractable. State of the art neural networks have beaten grandmasters at Go [36], are able to predict the evolution of a physical system given only a few frames of video [37], and can synthesise speech that is almost indistinguishable from a human [38]. It looks like neural networks are here to stay.

While the above mentioned a few of the key moments in the development of this machine learning technique, it is worth examining the history of neural networks in more detail, given in Section 3.1. A few of the more common neural networks are then inspected in Section

3.2, before inspecting their inner workings in Section 3.3. Finally, the current limitations of neural networks are discussed in Section 3.4.

3.1 A Brief History of Neural Networks

In the early 1940s, Warren McCulloch and Walter Pitts attempted the task of representing biological neural networks, such as those found in the brain, in a simplified and abstract mathematical model [39]. It is this initial theorem that laid the groundwork that eventually developed into the neural networks that are used today. In fact, the artificial neuron that McCulloch and Pitts describe remains similar to modern artificial neurons. McCulloch and Pitts set out to discover how complex behaviour can emerge from simple components in an effort to understand the brain. They modelled their mathematical neuron after the real biological neuron, specifically, the idea of **excitation** and **inhibition**. In general, as neural activity flows through the biological brain, some neurons will respond positively to such activity, and some neurons will respond negatively. These signals, fed via nerves into one of trillions of neural cells, either cause that specific neural cell to excite and feed electrical current forward into whichever neurons it is connected to, or inhibit that neuron, and prevent any more activity.

It is this idea, that the excitation and inhibition of these trillions of neurons leads to complex emergent behaviour, that McCulloch and Pitts wished to model. Hence, they gave their artificial neurons broadly similar behaviour. Information flows through the artificial network, and for a given neuron, if the current inputs to that neuron (from other neurons in the network) amount to a value over a certain threshold, then that specific artificial neuron sends a value forward to the neurons it is linked to. (Artificial neurons are described in more detail in Section 3.3)

The next notable development in neural networks was developed by Frank Rosenblatt, nearly two decades after McCulloch and Pitt's formalisation of biological neurons. Rosenblatt [40] developed the **Perceptron**, an algorithm intended to be capable of recognising patterns, in 1958. While Rosenblatt's model was based on prior biological work, it was certainly not intended to fully explain how a human mind works, merely to provide a model that allowed machines to learn in a similar fashion to how the human brain was suspected to learn. Even today, despite advances in neural networks, it still cannot be said that they model the brain in any appreciable fashion beyond surface level. Certainly no neural network can be said to be as complex (even in design) as the brain itself.

Two years later, in 1960, Rosenblatt proved such an algorithm can learn any linearly separable function that can be encoded in variations of its parameters. The perceptron itself is equivalent to a single layer of McCulloch-Pitts neurons (the inputs) feeding forward into a single layer of McCulloch-Pitts neurons, which serve as outputs. However, single layer perceptrons are significantly limited. They cannot solve any problem that is not linearly separable. However, it was known that perceptrons with more than one layer could solve such problems [41], yet no efficient method of training perceptrons with multiple layers yet existed.

Over a decade later, in 1982, Rumelhart et al. [42] published work on a method of training Multilayer Perceptrons (MLPs) via a generalisation of the *delta rule*. The delta rule describes a method for correcting error in a neural network. Essentially, it arises as gradient descent (an algorithm for descending to some minimum of a function) is applied to reduce the output error of a network. Rumelhart's work generalises the delta rule into a function known as *backpropagation*, capable of efficiently calculating the error of each individual layer of the neural network, allowing fast training of MLPs. Multilayer neural networks are particularly interesting as they have been shown to be universal approximators [43], and hence technically capable of approximating many different functions. Backpropagation is still used today, and inspected more closely in Section 3.3

This sudden access to the ability to train neural networks that can learn internal representations of the input data led to MLP being applied to many different computer vision problems, such as optical character recognition [44], handwriting analysis [45], and Hindi character classification [46]. Interest in multilayer perceptrons waxed and waned as the years went on, and various modifications to their structure arose towards the later end of the millennium. Particularly of note is the *convolutional neural network*, a type of multilayer perceptron introduced by LeCun et al that used mathematical convolution for image classification, and the Long Short-Term Memory (LSTM), a form of recurrent neural network useful for time-series based classification. (Both described in more detail in Section 3.2)

As both available processing power and available datasets increased, the idea of *deep learning* arose. Deep learning involves the use of *deep networks*, which are neural networks with an arbitrary amount of layers beyond the standard three common to multilayer perceptrons.

These networks have been shown to have human competitive results over many different problems, especially in computer vision. While deep learning has existed in some form since around 2006, where Hinton et al. [47] generated deep networks by training one layer at a time and composing the trained layers together, interest in modern deep learning, involving training all network layers simultaneously, arose when Krizhevsky et al. [48] won the Imagenet training competition, thus proving deep networks capable of state of the art results.

However, why deep neural networks work so well remains an open question, with several theoretical answers. One such answer states that the large amount of layers correspond to an increase in complexity of the internal representation the network has of the input problem, hence leading to a superior ability to approximate the required function [49]. Another common belief is that deep neural networks learn better abstractions to the data than shallower networks [50]. However, it appears that just because *how* they work is not rigorously understood, it does not mean that they cannot be *made* to work.

3.2 Common Neural Networks

While new kinds of neural networks are being developed every day, much state of the art work has been accomplished by the three networks listed below, and impressive results can be achieved by combining these kinds of networks in various different manners. Hence, understanding the inner workings of these networks gives a good overview of the field.

Before continuing it would be worth defining what a neural network **is**. An artificial neural network is a collection of artificial neurons, connected arbitrarily together. An artificial neuron takes an arbitrary amount of inputs, and produces an output. There exists parameters inside the network that define the effect each input has on a given neuron, and these parameters are changed until the output of the neural network is correct given a certain input. Once this is true for all desired inputs, the network is considered ‘trained’, and can be used to attempt to provide an output for unseen inputs. More technically, a neural network approximates some function that maps the input data to the output data, by projecting the input data into a higher dimensional space, and finding a function that either fits the projected data (in the case of regression problems) or finding a hypersurface that divides the data into an arbitrary amount of different groups (in the case of classification).

3.2.1 Multilayer Perceptrons

Multilayer perceptrons are characterised by having three or more layers of artificial neurons, and are the ‘simplest’ form of neural network. Multilayer perceptrons are capable of solving problems that are not linearly separable, and are universal approximators [43].

Multilayer perceptrons are outperformed in most cases by more domain-specific neural networks, such as using convolutional networks for images or recurrent neural networks for time-series data.

3.2.1.1 Structure

A traditional multilayer perceptron is composed of a set of input neurons, with their values equivalent to the input parameters, a set of output neurons, that idealistically provide a solution to the problem, and any number greater than zero of ‘hidden’ layers in between. It is the addition of the hidden layers that provide the multilayer perceptron with its ability to fit data that is not linearly separable.

Each layer of the perceptron is *fully connected* and *feed-forward*. Fully connected means that each neuron in the previous layer is connected to each neuron in the next layer, and feed-forward means that the neural network contains no loops, and that information never flows backwards inside the network (unlike recurrent neural networks).

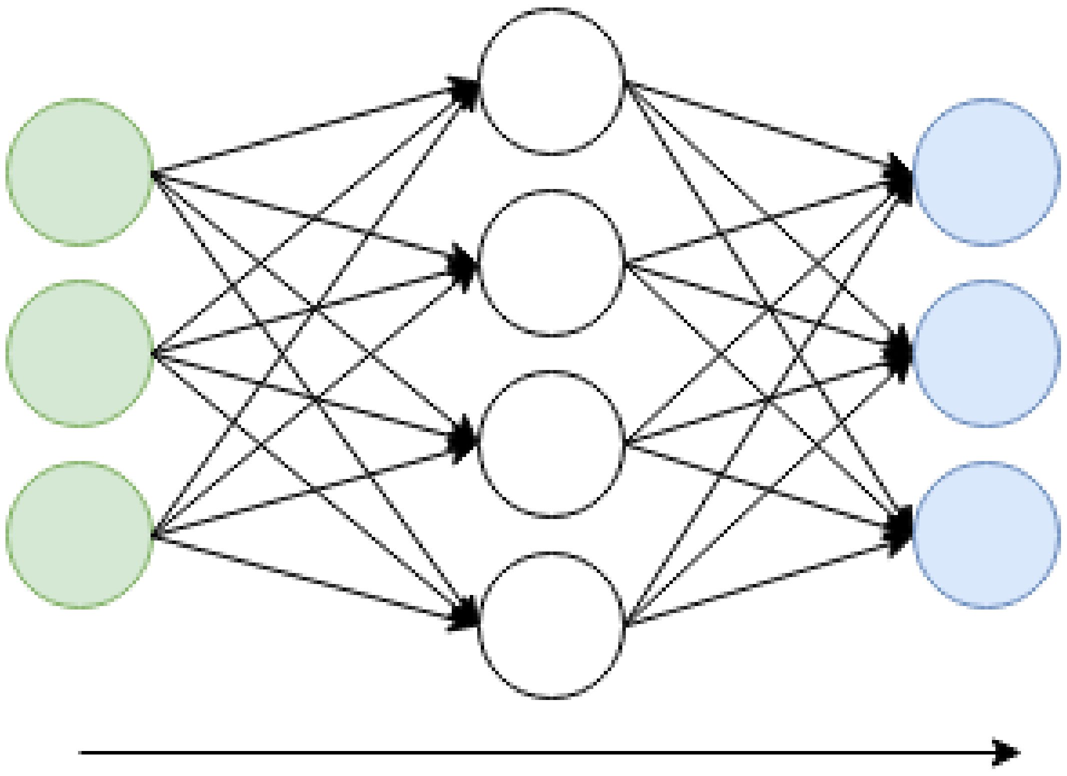


Figure 3.1: Diagram of a simple multilayer perceptron.

Figure 3.1 is the structure of a classic three layer multilayer perceptron. The green circles are the input neurons, and the blue circles the output neurons. There are four neurons in the 'hidden' layer. Information flows through the network in the direction of the arrows.

3.2.2 Recurrent Neural Networks

One of the main problems with traditional neural networks is that the current input is entirely separate from the previous input. This makes sense for discrete data, such as classifying species of animals. It makes less sense, however, for learning functions about data that depends on some previous data.

Recurrent neural networks are a form of neural network where connections between neurons are no longer constrained by being strictly feed-forward. Connections may skip neurons or form cycles between groups of neurons. While there are many different kinds of recurrent neural network, this section will focus on one that has achieved state of the art results in the areas of connected handwriting recognition [51] and automatic translation [52].

3.2.2.1 Structure

Recurrent networks can be thought of as a special kind of network that takes not just the data set input, but also its own previous output, given the input at that time. In a Long Short-Term Memory (LSTM) network, four individual layers make up one LSTM 'module'.

These layers are:

- The Forget Gate
- The Input Gate
- The Output Gate
- The Memory Cell

The memory cell stores some output, the input gate decides which incoming data to store in the memory cell, the forget gate decides which data to remove from the memory cell, and the output gate decides how much the memory cell affects the output of the LSTM cell [53].

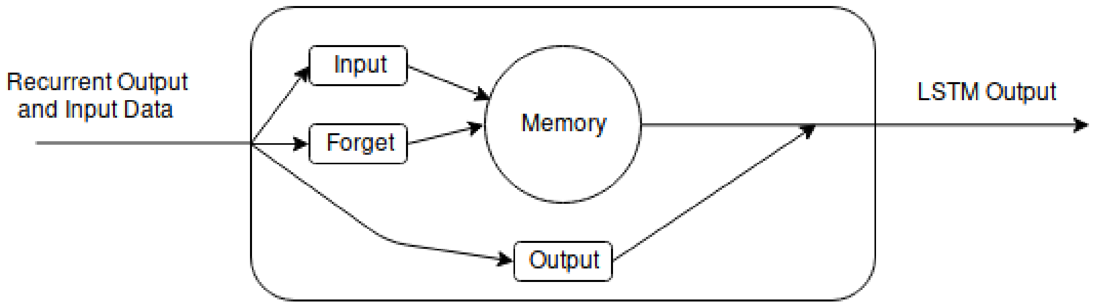


Figure 3.2: Simplified diagram of an LSTM Module.

Figure 3.2 is an extremely simplified representation of an LSTM cell. The previous recurrent state and input data flow into the cell on the left, the input and forget gate control the effect this input has on the memory cell, and the output gate controls the effect the memory cell has on the output.

3.2.3 Convolutional Neural Networks

Convolutional Neural Networks are a specific type multilayer perceptron optimised for image datasets. They are particularly robust to distortions and variance in the spatial locations of features found in images. Unlike multilayer perceptrons, however, CNNs are not *fully connected* throughout the entire network. This means that certain neurons in previous layers do not connect to neurons in later layers. Because CNNs are not fully connected, they also scale better to larger input sizes than multilayer perceptrons. For example, for a 100x100 image, a multilayer perceptron neuron will have 10000 connections, and hence 10000 weights to tune. This lack of connectedness arises because CNNs have a *receptive field*. The receptive field can be thought of as the area that the CNN is currently examining [54]. Hence, neurons will only be connected to their respective receptive field, rather than the entire input. This implies that certain neurons will only excite when their receptive field is over a certain feature, and this provides the key to understanding what causes convolutional neural networks to perform so well.

As certain neurons only excite in response to certain features, *no matter where in the input this feature is located*, each group of neurons learn to identify particular features in the input. Groups of features together may activate groups of neurons in later layers, and so on. For example, some neurons may excite in response to finding curves. Later neurons may excite upon finding those neurons activating together in a certain pattern, and so on, until the network can successfully identify a circle. Convolutional neural networks have been able to classify images and objects in images with an accuracy impossible before their invention. This includes human-competitive identification of hand written digits (error less than 1%, easily achievable on a modern desktop computer), and relatively recently, a convolutional neural network variant achieved an error rate of less than 4% on the Imagenet dataset; a dataset made up of one thousand different classes of images [55].

3.2.3.1 Structure

Modern convolutional neural networks are made up of several different layers arranged in a standard feed-forward architecture. There is no hard rule for the exact organisation of these individual layers, and some may even be omitted if desired, though this is may lead to a loss of accuracy.

These layers are:

- The Convolutional Layer
- The Pooling Layer
- The Non-Linearity Layer
- The Fully Connected Layer
- The Output Layer

The **Convolutional** layer is the layer that gives the convolutional neural network its name, and is, perhaps unsurprisingly, the core component. Convolutional layers are unusual in that they are three dimensional rather than two dimensional. Each input from the receptive field covers a small region of the input image, but extends throughout the depth of the input. This is known as a filter. As this filter is convolved with the input image, for each position the filter is at, a two-dimensional activation map is created. These activation maps are then stacked together to create the three dimensional convolutional layer. All neurons in the same position two dimensionally in the convolutional layer examine the same area of the image, but activate on different features. Three hyperparameters determine the structure of the convolutional layer. The **depth** is how many activation maps are learnt. The **stride** determines how the input filters are moved across the input, and how many neurons overlap onto the same features. Finally, the **padding** determines whether the input volume is padded to maintain the same spatial size after the convolution operator is applied [56].

The **Non-Linearity Layer** introduces a source of non-linearity into the convolutional network. Until this layer, all operations have been linear. Usually, a layer of Rectified Linear Units are used that apply $max(0, x)$ to the output of the convolutional layer. This specific *activation function* is used as it is easier to compute than other non-linear functions. Non-linearity is useful to allow the network to fit to non-linear patterns [57].

The **Pooling** layer implements some form of *pooling*, a mathematical technique that reduces the spatial size of the representation of the input. Commonly used is 2x2 max pooling, which examines each 2x2 region in the representation, and preserves the highest value, creating a new representation that is a quarter of the size of the initial. The theory behind pooling is that it removes unnecessary details in the image, and that important abstractions will be maintained even as detail is lost [58]. Pooling also has the advantage of reducing the number of connections between neurons even further. However, recently pooling is falling out of favour in deep networks, as it was found that it can be replaced with a small convolutional layer with increased stride.

The **Fully Connected Layer** connects to all neurons in the previous layer, and is usually used as the second last layer in the overall network. This layer has access to all the output values of the non-linearised convolutional layer, and hence can identify non-linear patterns in that data, such as combinations of features.

Finally, the **Output Layer** returns the output of the convolutional neural network. Usually, the $softmax$ function is applied at this stage to convert the output of the fully connected layer into a probability distribution of classes (if performing classification). The error in this stage is used to adjust the parameters in the network to lead to better performance, via a *loss function*. For more details on loss functions, see Section 3.3

Additionally, convolutional neural networks use a technique known as **dropout**. Dropout randomly sets a proportion of the activations of a layer to zero, essentially removing them. The purpose of this is to make the neural network more robust, and prevent overfitting to the input data [59].

3.3 Neural Networks in Detail

This section examines the individual components of neural networks in more rigour, in order to paint a clearer picture of why certain hyperparameter choices make more sense than others, especially with regard to loss and activation functions.

3.3.1 Artificial Neurons

Artificial neurons have one or more inputs, each with a weight associated with it. The weight describes how much of an effect the input has on the neuron. In most cases, a bias value is also added to the neuron, which increases the possible range of the output (this bias value is tuned as the weights are tuned).

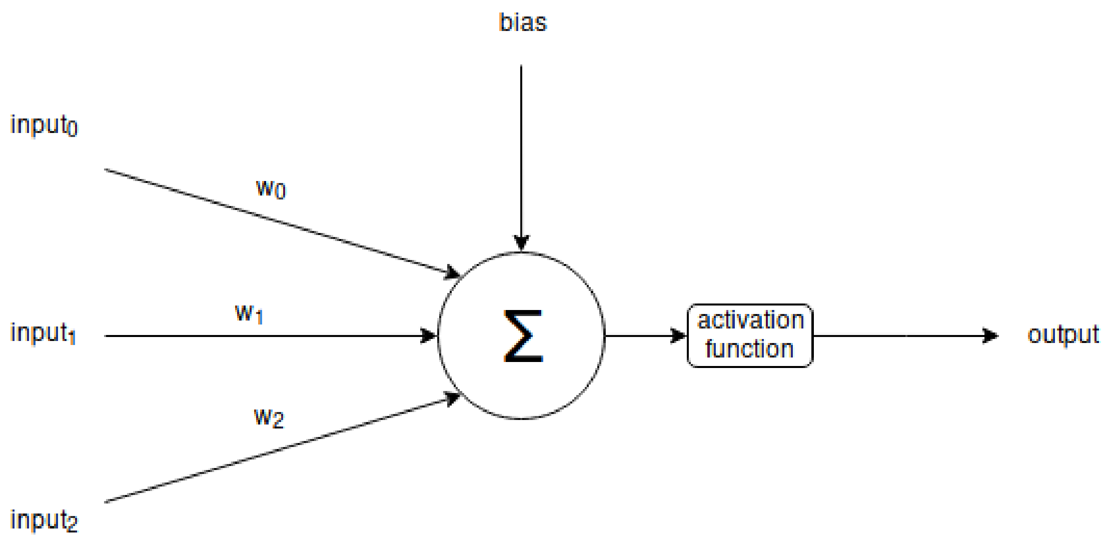


Figure 3.3: An Artificial Neuron

Figure 3.3 is an artificial neuron. The inputs are scaled by the weights and summed together. A bias is added, and then the summed value is passed through an activation function, giving the output of the neuron.

Hence the output of a given artificial neuron can be described as:

$$out = A(\sum_{i=0}^{n} w_i x_i + b)$$

where n is the number of inputs, w_i is the weight of the i_{th} input, and x_i is the i_{th} input, b is the bias, and A is the activation function.

3.3.2 Activation Functions

The activation function converts the linear equation of the neuron into a non-linear output suitable for fitting a function onto non-linear data. In theory, any non-linear function can be used. However, ideally the function should be fast to compute, and for reasons explained later in this chapter, the function must be *differentiable*.

The most commonly used activation functions are the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (RELU). In practice, the rectified linear unit is used most often, because as mentioned above it is much faster to train than others. Additionally, the sigmoid and hyperbolic tangent functions suffer from the *vanishing gradient problem*, where the neural network training becomes increasingly slow [60]. These functions should only be used on the hidden layers of the network, for output, the softmax activation function is often used.

Graphs of the hidden layer activation functions are shown in Figures 3.4, 3.5, and 3.6

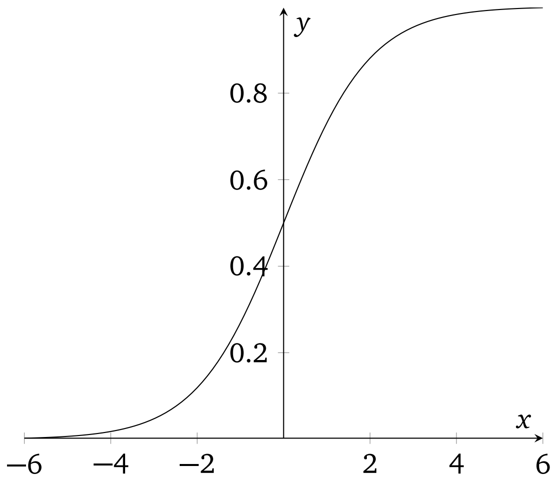


Figure 3.4: Graph of sigmoid function

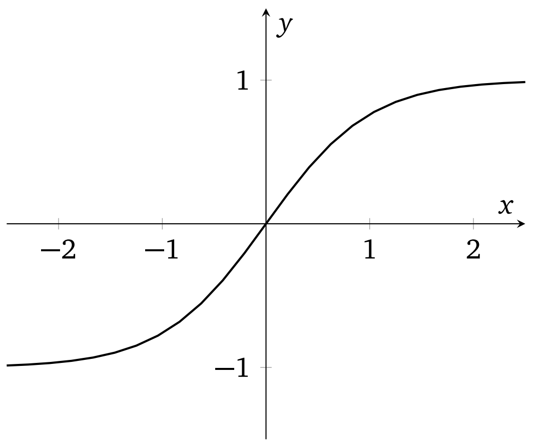


Figure 3.5: Graph of hyperbolic tangent

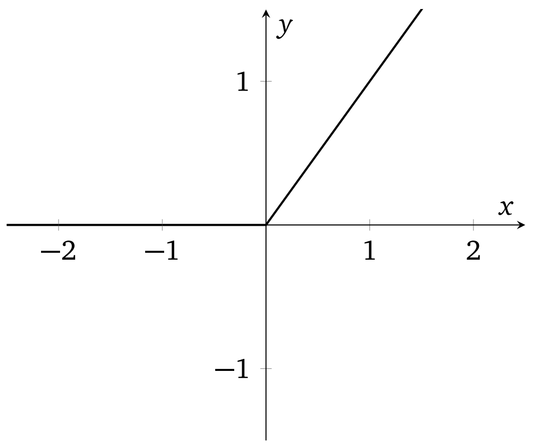


Figure 3.6: Graph of a rectified linear unit

3.3.2.1 Softmax

The softmax function is a function that converts the output of the final layer of the neural network into a probability distribution. This is useful, as it allows the probability of a certain input being of a given class to be computed and output at the same time. Softmax is a generalisation of the sigmoid function, and is particularly useful for multiclass classification problems.

3.3.3 Loss Functions

The loss functions of a neural network calculate the error between the output and the desired output. Like activation functions, many different forms of loss functions exist, as there are many different methods to determine how close two sets of values are. Much like choosing a fitness function for an evolutionary algorithm, there is no direct method to identify the best loss function for a neural network. If the loss function being used provides the desired accuracy, then there is little reason to change it.

One of the most commonly encountered loss functions is **Cross Entropy**

Cross Entropy is used for both binary classification problems and multiclass classification problems and measures the error between the probability distribution of the output, and the probability distribution of the desired output.

$$e = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

where y is the desired distribution and $\hat{y}$ is the predicted distribution. Typically the average of this error for all samples is used as the cost function. For multiclass problems, cross entropy is summed over each class in the distribution.

3.3.4 The Backpropagation Algorithm

Backpropagation is a critical component for training deep neural networks, and the reason that all activation functions used in a deep neural network must be differentiable.

Nearly all neural networks are trained using a method known as *gradient descent*. The aim of a neural network is to find the best combination of weights and biases that eventually provide the best output of the neural network, measured via a loss function. Gradient descent itself refers to minimising the loss function by descending ‘downhill’ in the higher dimensional loss landscape. The lower in the landscape, the smaller the loss, the more tuned the weights and biases, and hence, the greater the accuracy.

However, to use gradient descent, the best gradient must be found from the current position to proceed down. Backpropagation is the algorithm that determines this gradient. Until Rumelhart et al popularised this technique, training of multiple layer neural networks was much less efficient. In order to calculate the effect a weight change may have on the error, previously each individual weight would need to be altered and the effect on the entire network computed.

Backpropagation solves this inefficiency by calculating the partial derivatives of the error function with respect to each weight in the network, for each hidden layer, by working backwards from the output layer. Once each derivative is calculated, the best gradient for the given weights and biases can be found, and the weights and biases are adjusted with respect to that gradient, minimising the loss function each time this algorithm is executed.

3.4 Limitations of Neural Networks

After reading this chapter, the reader may be left with the impression that neural networks are the be-all and end-all of machine learning. This is not the case. It's true that neural networks have achieved excellent results in areas that were previously intractable, and that they are likely to continue setting records for at least the foreseeable future, but neural networks suffer from problems of their own [61] [62].

3.4.1 Blackbox Nature

There still remains no good all-encompassing model that explains why neural networks work so well. Input data is passed in, and an accurate prediction may come out, but exactly which connections in the network are responsible for this prediction, and why, remains a mystery. Inspecting the inner workings of neural network remains difficult, as can debugging a specific problem within the network. In the case of convolutional networks, the individual filters and activation maps can be extracted, but in the later layers of the network these look like so much random noise to a human.

3.4.2 Hyperparameter Tuning

How deep should the network be? How many neurons per layer? Which stride to choose for the convolutional layer? Which filter size? There is no formal method for choosing these, or any other hyperparameters for a given network, aside from human intuition. Certain guidelines can be taken from previous work in the field, but even this may not be applicable to the new problem to be solved.

3.4.3 Available Datasets

Neural networks can only learn what is available to be learnt, and in general this is from datasets with tens of thousands of data points, often labelled time-consumingly by a human. If the dataset of sufficient size does not exist, it must either be (usually, at great expense) created, or a new technique must be found. Exploring a method of using neural networks with less data was a part of this research project.

3.4.4 Computational Power

Deep neural networks, of the kind that achieve record breaking results in classification competitions, often have hundreds of layers. Such networks are usually trained using expensive GPUs, and even with their thousands of parallel processors, training time can still exceed a month. This renders deep neural networks with thousands of classes impractical for problems that require fast solutions. This is due to the amount of connections, each with associated weights, and neurons, each with an associated bias, all of which must be tuned via a learning algorithm. A state of the art deep network could have nearly two million parameters to tune [55].

3.5 Conclusion

In this chapter, an overview of neural networks has been presented, starting with the history of the first formalisation of an artificial neuron, all the way through the modern convolutional and recurrent neural networks. Additionally, the inner workings of a neural network has been explored, and a look at the current limitations neural networks face. A specific focus was placed on explaining the workings of convolutional neural networks, as this is the network that this research project has been examining for the purposes of integration with a genetic programming system, which is the topic of the next chapter.

CHAPTER 4

Combining Evolutionary Algorithms and Neural Networks

For the first few years after their conceptualisation, neural networks and evolutionary algorithms existed as quite separate entities. However, it did not take long for the two different machine learning algorithms to cross paths. While research into combining these two methods is relatively rare compared to the bulk of research published about either technique, it does exist. On the other hand, nearly all research in this field is involved with attempting to use evolutionary algorithms to optimise either the structure, or the hyperparameters, or both, of neural networks. There appears to have been no prior research into making neural networks available as components for an evolutionary algorithm to use.

4.1 Existing Work

Research into combining evolutionary algorithms and neural networks saw a brief period of interest around the late 1980s, continuing to the latter half of the 1990s. After that, it seems the scientific community lost interest in this technique, possibly due to neural networks falling out of favour due to the lack of processing power required for good results. Again, much like neural networks, there has been a resurgence, as researchers begin to search for esoteric methods of training, in the hopes of finding a better method than backpropagation [63]. Brameier et al. [23] did attempt to use a *linear* neural network to combine and weight the outputs of genetically evolved programs as early as 2001, but this combination never appears to have gained much traction; perhaps because Brameier found that a linear neural network did not give an advantage over other linear weighting techniques. Today, the idea of a linear neural network seems strange, and researchers appear to strive to add as much stable non-linearity to neural networks as possible.

Recent results have shown that not only can neural networks be *trained* by a genetic algorithm, the performance achieved with such a network surpasses modern state of the art networks at the specific task it was trained to do. Modern neuroevolution appears to be centred around evolving the parameters of neural networks, which, interestingly, is where this technique began.

4.1.1 Neuroevolution

The field of neuroevolution began in 1989, when Montana et al [1] published his paper on training neural networks with genetic algorithms. Montana et al encoded the weights and biases of the neural network as the chromosomes of individuals in the genetic algorithm, and tested the performance of the neural network on a set of tasks. This testing provides the fitness function for the genetic algorithm. Montana et al found that it was possible to use a genetic algorithm to train the network, and also found that a genetic algorithm ran for 10000 evaluations outperformed backpropagation ran for 5000. Montana et al also notes that such a training scheme can be used to train neural networks with discontinuous activation functions, a task that backpropagation cannot perform, as discontinuous functions are not differentiable. (See Chapter 3)

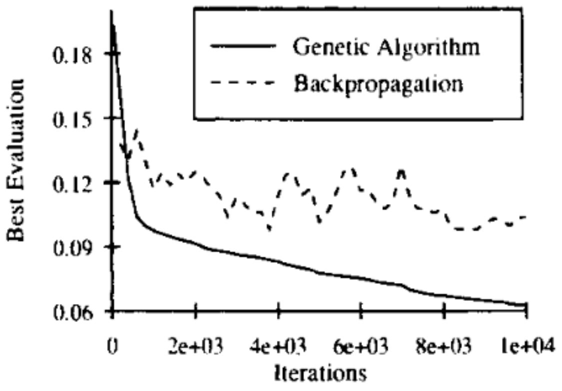


Figure 4.1: Genetic algorithms vs backpropagation. [1, Fig. 8]

Only slightly later, in the same year, Miller et al [2] explores the problem of evolving neural network architectures using a genetic algorithm. Miller notes that designing a good neural network is a problem difficult for humans to accomplish well, and that it's often impossible to tell whether a good solution has been chosen over a great solution that was missed. Miller proposes the use of genetic algorithms to explore the possible search space of neural network architectures, as normal search methods and random methods are likely to fail due to the extremely large (and high computational complexity) spaces that encode the possible architectures. The fitness of each genetic individual is assessed via the training of the neural architecture, followed by its evaluation on a test problem. Miller finds that it is possible to evolve these networks, but such architectures tended to be convoluted, asymmetrical, and dissimilar to human methods of architecture design. Nonetheless, Miller et al achieved the goal of separating neural networks from human biases. Generated architectures and human designed architectures were not rigorously compared.

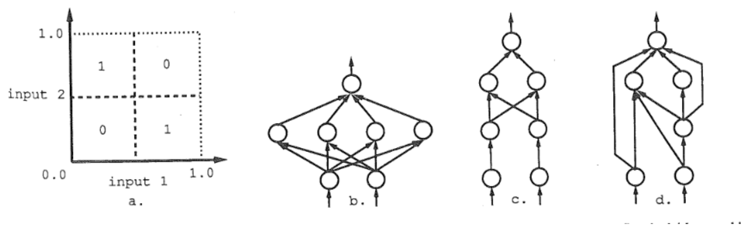


Figure 4.2: Evolved Architectures. [2, Fig. 3]

In 1990, Whitley et al [64] explores both the evolution of weights in neural networks as well as the evolution of novel architectures, noting that genetic algorithms make excellent tools for creating neural networks as they make no assumptions about what the output should look like, as long as it functions well. Whitley found that optimising the connection weights in an artificial neural network causes the network to converge (find a good output) faster than backpropagation on the hardware at the time. Whitley also found genetic algorithms capable of refining existing neural network architectures by removing unnecessary connections, leading to new network topologies. The refined networks learned faster with a smaller amount of connections than the unrefined network. Both the results of Whitley et al. match the findings of Montana et al. and Miller et al, however only very small (less than 50 connections) networks were experimented with.

Several years later Stanley et al [65] applies neuroevolution to the problem of reinforcement learning, finding that evolved neural network architectures learn significantly faster than fixed topology networks where only the weights are evolved, when attempting to find solutions to the pole-balancing problem. Like Whitley et al, the trained network was small compared to modern networks. In 2004, Oullette et al. [66] found that a fixed convolutional neural network trained via a genetic algorithm is more accurate at detecting cracks in images than the same network trained via backpropagation. Being a convolutional network, the number of connections is far greater than a standard multilayer perceptron, indicating this technique can work for larger networks.

Interest in this area then appears to wane, until in 2016 Ijjina et al [67] finds that using a genetic algorithm to set the initial weights of filters in a deep convolutional network minimises the final accuracy of the network, with a seven layer neural network. Such a network would have thousands of connections. Finally, in 2018, Such et al [68] uses a genetic algorithm to evolve the weights of a deep neural network with over four million parameters. Genetic algorithms turn out to be a competitive alternative to using backpropagation in deep networks, and in certain cases the evolved networks perform better than the standard networks. These results are despite the extremely simple genetic algorithm used by Such. The key to efficient training of a deep network is efficient compression of the trainable parameters into a representation the genetic algorithm can work with. Such found a representation method independent of the size of the network; the networks represented in the research of Such et al. were able to be compressed to a representation between 8000 and 50000 times smaller than the network itself. Such notes however, such a compression is not generalised, and is only usable with neural networks intended to be trained by a genetic algorithm. Because of this compression, other neuroevolution techniques are now able to be applied not just to small artificial neural networks, but also to very deep networks.

4.1.2 Conclusion

Neuroevolution has had a long and interesting history. After experiencing a golden age in the 1990s, once the complexities of the problems began to outstrip the available computational resources and techniques of the time, interest in this technique fell off. Only very recently, with vastly superior computing power, and advanced compression techniques, does neuroevolution seem set to make a comeback into the world of machine learning, until now dominated by extremely deep networks trained with backpropagation.

However, while research has been conducted into using genetic algorithms to evolve and design neural networks, it seems no research has been conducted into combining already trained neural networks with genetic algorithms. Such a combination may be able to use existing neural networks in manners that a human machine learning architect may not think of, to solve problems without having to 're-invent the wheel'. After all, there are many neural networks out there, but their actual use still depends upon human choice, and human biases. The next section discusses the problem, and the main aim of this research project.

4.2 This Research Project

4.2.1 The Problem

The human visual system is a excellent example of a learning system. It can learn to identify objects in thousands of poses and against thousands of backgrounds, after only seeing such an object one time. It can generalise well, for example, by guessing that any object with a flat base and four legs is a chair, even if having never seen that specific chair before. And it is not just humans that have such a system; nearly every living being on earth has some form of sight. The difficulty lies in copying this ability onto a machine, such that we can bring machine vision closer to our own.

Some progress has been made in using genetic programming to evolve vision systems. Such a technique is relatively generic, capable of using a small amount of training data and genetic programming to build classifiers that when executed on an image are capable of identifying and classifying objects in that image. However, the 'toolkit' used by this system; the components that each classifier is built out of, are relatively primitive compared to modern day vision techniques, and hence accuracy suffers.

Convolutional neural networks are excellent at classification problems, but require lots of training data for an accurate result. A network capable of classifying both types of cars and types of planes requires thousands of images of both. The more classes, the more training data and the more complex the neural network required.

Is there a way to combine these two techniques such that the drawbacks of both are minimised, and more effective object classifiers can be evolved by using genetic programming?

4.2.2 A Possible Solution

Convolutional neural networks work well at image classification, but in order to actually apply them to a problem, human decision is still involved. By integrating trained neural networks with a genetic programming system, it may be possible to evolve image classifiers that make use of one or more neural networks, alongside other techniques, to classify regions in images with greater accuracy than a genetic system alone. This is somewhat conceptually similar to the approach employed by Oeschle et al, which itself was inspired by haar cascades. This removes the human element in deciding whether a trained neural network should be applied to a problem; if the neural network is beneficial for solving the problem, the evolved classifier will automatically include it in the final output. The general idea is that a classification problem of many different classes could be solved by some evolved combination of neural networks, automatically, with very little human input, with each network being able to identify one or more of the classes.

Such a combination could several advantages of each technique individually. Primarily, the genetic algorithm would no longer have to evolve classifiers for objects that an integrated neural network can identify more accurately. This should increase the final accuracy of the evolved classifier. Another advantage is that the complexity of the included neural networks can be reduced. Whereas before to identify planes or cars, a network that can identify both must be used, with this method, two separate networks could be used. The genetic algorithm would then choose which neural network to use depending on the problem. For example, if the task is to identify planes, the neural network that can identify cars would not be used. This means that rather than one large, complicated network, many smaller networks can be used. These may require both less training time, and less training data, for a similar result, as the task of ‘plane or not’ is simpler than ‘plane or car or neither’. If this is possible, then the need to train extremely deep, large neural networks diminishes in real-world situations, as the genetic system would be able to simply use the best network for the problem at hand.

4.2.3 Aims and Objectives

The central aim of this research is to determine whether it is possible to integrate neural networks into a genetic programming system capable of evolving image classifier, such that the inclusion increases the classification capability of the system.

A secondary goal is to determine if such a system is capable of combining less complex neural networks to achieve a similar result to including one large, complex neural network.

Along the way to integrating neural networks, various other aims arise:

(1.) Design an efficient fitness function that describes the problem well.

(2.) Determine if the filters extracted from various layers of a convolutional neural network can be used in the networks place, likely reducing the required processing time.

(3.) Implement high-level feature descriptors, in order to determine whether, if it is possible to integrate neural networks, they offer a tangible benefit over such descriptors.

4.2.4 Challenges

As mentioned in Chapter 1, various challenges exist that will affect this research. The main problem is certainly the difficulty in designing a good fitness function. The fitness function defines the behaviour of the genetic algorithm. A poor fitness function will give strange, and even useless, results, and render the evolved classifiers useless. Hence, it is no surprise that a large amount of time was devoted to designing good fitness functions as the project progressed, and several were implemented over the course of this research.

As the amount of possible tools the genetic algorithm can choose from increases, so does the search space, and hence, the time required to find a good solution. This is a challenge beyond the scope of this thesis to address. However, the author is confident in stating that a neural network's classification accuracy means that including several different neural networks, while increasing the search space, certainly increases it less than providing the system with the tools to evolve classifiers with a similar accuracy. However, it is possible that introducing such networks may increase the complexity of the search space, and render it more difficult for a genetic algorithm to traverse, especially considering the point below.

Combining two forms of machine learning has the side effect that any error in one will compound with error in another, potentially meaning that poorly designed neural networks can skew the genetic algorithm and result in poor performing classifiers. However, the nature of a genetic algorithm means that individuals that use poor neural networks will die out, and the final classifier should not utilise any neural networks if none of them provide an increase in fitness. However, the algorithm may also become stuck in a local minimum due to such a network, and hence evolve a less than optimal classifier.

Neural networks are often seen as a black box, as the inner workings are difficult or even impossible for a human to decipher. Genetic programming does not necessarily suffer from this to such an extent, as even in the worst cases the program evolved can still be inspected. However, the reasons that a genetic algorithm made one choice over another are not always be apparent, and because the evolved programs would also be making use of neural networks, the operation of a classifier evolved by this program may become more opaque.

Finally, this technique is unlikely to improve the performance of the evolved classifier if the task that it is being trained on is not a task that any of the neural networks in the system can assist with. This means the systems accuracy becomes dependent on which neural networks are available to it at that time.

4.3 Conclusion

In this section an overview of research into combining evolutionary algorithms and neural networks has been presented, as well as a description of the new method of combining these techniques explored in this research project, and in general this section concludes the review of the techniques relevant to this project. By now, the reader hopefully has a cursory understanding of evolutionary algorithms, neural networks, their strengths and weaknesses, and the benefits of combining these two techniques.

Now that the problem has been defined and a solution identified, the next chapter explains the actual system implemented to attempt to tackle this problem, and the difficulties encountered in the technical implementation of this theory.

CHAPTER 5

The Implemented System

5.1 An Introduction to GPT

Genetic Programming, Typed (GPT) is the name given to the core implementation of a typed genetic programming system. GPT, combined with an appropriate function set, terminal set, and fitness function, is capable of evolving vision systems that can classify regions of images. GPT is inspired by the work of Oeschle et al, in the sense that it is a typed genetic programming system for evolving vision systems [31]. This makes GPT a good base for further experimentation in this area. It was originally created by researchers in the Department of Computer Science and Electronic Engineering at the University of Essex. Throughout the course of this research project, GPT has been heavily modified.

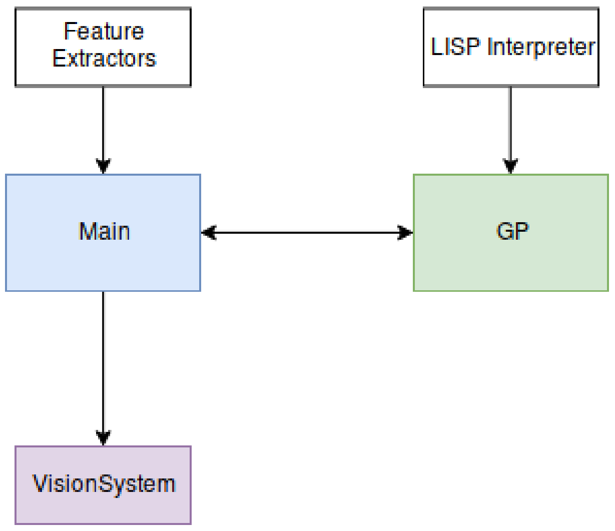


Figure 5.1: High Level View of GPT

GPT is composed of two main components, and several ancillary ones. These components are the appropriately named 'Main' component, which is responsible for setting up the terminal sets, functional sets, fitness functions, and controlling the evolution of the population of individuals, and the 'GP' component, which is an implementation of a typed genetic programming algorithm. The functional set contains all functions that the GP system may choose to implement in the evolving program. These functions must take inputs, and return an output. The terminal set also contains functions, but these functions take no input, and only return an output. Hence, they can only be included as the 'leaves' of the growing program tree. A full list of all available functions and terminals that the developed system may choose from is available in Appendix B.

The ancillary components are:

- A component that represents a trained (evolved) vision system
- A program that can run such vision systems
- A set of feature extractors
- A LISP Interpreter (third party)

For general processing, OpenCV 2 and Numerical Python are used for computer vision processing and mathematical operations respectively.

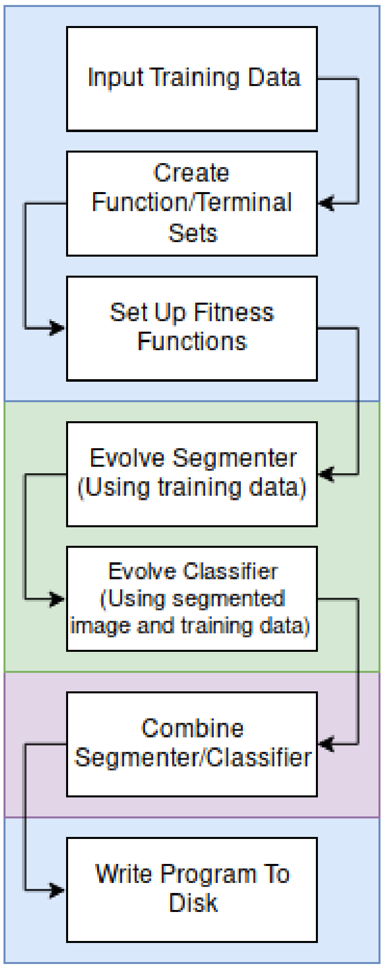

Input Training Data
Create Function/Terminal Sets
Set Up Fitness Functions
Evolve Segmenter (Using training data)
Evolve Classifier (Using segmented image and training data)
Combine Segmenter/Classifier
Write Program To Disk

Figure 5.2: GPT program flow, each components responsibilities colour coded as per Fig. 5.1

5.1.1 The Main Component

The Main component is the executable part of the GPT system, that when ran, and given suitable training data, starts the process of evolving a vision system. In this part of the system, fitness functions for both segmentation and classification are defined, input arguments are parsed, and the functional and terminal sets are initialised.

The evolution of a vision system is a two step process. Initially, a program capable of segmenting regions of interest from other areas of the image is evolved, using a fitness function suited to this task. Once this is complete, another, separate program is evolved that can classify segmented regions. These two separate programs are then combined, and the final step of the Main component is to output a combined system that integrates these two components.

This component is in control of starting, stopping, and evaluating the output of the GP Component that runs the genetic algorithm, and as such can be thought of as a 'supervisor' for the other components that make up GPT. The Main component also sets the population size, the max program depth, and the number of generations to evolve the programs for, among other less important parameters. The user is in control of this program, and specifies when to begin training. Once the Main component has collated the training data, functional sets, and terminal sets, it initiates the training of the genetic algorithm.

5.1.1.1 Training Data

The training data used by the GPT system consists of a set of images, a set of masks for these images, and a metadata file that relates each mask to its appropriate image. The masks are generated using a separate program that allows a person to highlight in regions of the image in different colours (an example of one of these masked images is shown in Fig 5.3. Each painted region represents both a Region of Interest (ROI), and a named class. Multiple regions may have the same class. These masks and images are then associated using a metadata file. These masks and images are then used by the fitness functions to determine how correct the output of a given evolved program is. To fully train the system, this entire set of images and masks is provided to the system.

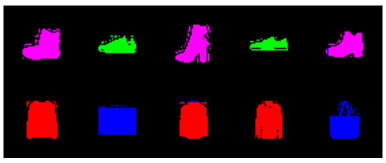


Figure 5.3: Example of a painted classification mask.

5.1.1.2 Fitness Functions

Each evolved vision program in this version of GPT outputs an image, and each input training image has an associated mask that identifies the ground truth. This means that fitness functions can be written that compare the masks and the output image to determine how close the output of the evolved program is to the ground truth when a training image is used as input. The error between output and truth can then be used to determine the error in the output and hence the fitness of the program. Two fitness functions are used, one for segmentation, and one for classification. The classification fitness function is based upon Oechle et al. Dynamic Range Selection 2 (DRS2) algorithm.

The **segmentation** fitness function is simple and effective, and as such has not been modified over the course of this work. This function examines the output of evolved program, and compares it to the training mask. Labelled regions in the training mask have a non-zero value, and other areas have a zero value. Hence, the fitness function can compare the amount and position of non-zero values in the output image to the training mask. Pixels that are non-zero where they should be zero, and vice-versa, add to the error. This forces the genetic algorithm to evolve a program that attempts to find regions of interest and give them a non-zero pixel value, while giving unimportant regions a zero value. The output of this process results in a black and white image, with regions of interest white and background black, an example shown in Fig. 5.4.

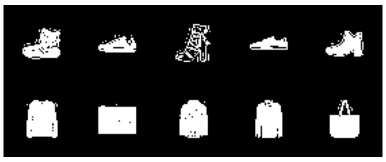
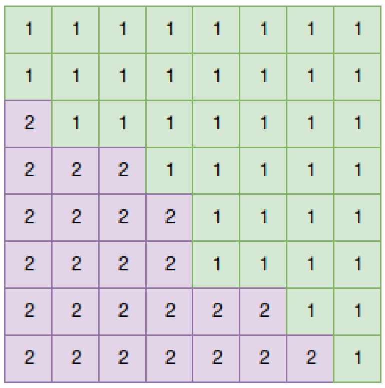


Figure 5.4: Example of a black and white segmented image.

The **classification** fitness function is far more complicated, and is an implementation of the DRS2 algorithm described in Chapter 2, Section 2.6. For this implementation, each slot in the class array is filled in using the training mask regions, and then filled in using the output of the evolved program, and then compared. This is a time consuming algorithm, and often did not lead to good results. One of the focuses of this research is creating a more suitable and faster fitness function for classification.



1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1
2	2	2	1	1	1	1	1
2	2	2	2	1	1	1	1
2	2	2	2	1	1	1	1
2	2	2	2	2	2	1	1
2	2	2	2	2	2	2	1

Figure 5.5: Example of pixel-wise confusion, the square has some pixels predicted as class 1, and some as class 2.

This DRS2 function is prone to making pixel-wise errors for each region, assigning half the region to one class, and half to another, despite many generations of training. These pixel-wise errors occur because the DRS2 fitness function directs the genetic algorithm to evolve programs that consider the image on a pixel by pixel basis, with no consideration for the extraction of entire regions as one object or identifying each region as a potential entire class based on the segmentation procedure.

All fitness functions in GPT are framed as minimisation problems. A fitness of zero is considered perfect.

5.1.2 The GP Component

The GP Component is an implementation of a typed genetic programming algorithm, as described in Chapter 2. Upon starting the evolution of a program, the GP Component initialises a set number of random programs, written in the LISP programming language, and built from the functional and terminal sets passed to it by the Main component. These random programs are guaranteed to be executable (this is a key caveat of Typed Genetic Programming) but the actual program is unlikely to do anything useful. There is no set of pre-built starting programs.

The programs are built to a set minimum and maximum depth. Both subtree crossover and subtree mutation is implemented. Tournament selection is used for selection purposes, with a size of two. No duplicate individuals are allowed to be generated to ensure maximum diversity, and this constraint is enforced throughout the evolution process. Around 1% of each generation's population survives unchanged to be placed into the population of the next generation.

As this is a typed genetic programming implementation, only terminals and functions with the corresponding types can be paired together during the crossover, mutation, and initial creation of the genetic population. This decreases the possible search space by removing invalid combinations of components, increasing the likelihood a solution is found in a suitable time frame.

5.1.3 Ancillary Components

5.1.3.1 Functional Set, Terminal Set, and Feature Extractors

The functional and terminal sets are gathered into dictionary structures in the Main component, but the actual definitions of these functions and terminals may be defined elsewhere, such as in feature extractor files. Each function available is actually a python function defined in a python file. Functions may take one or more arguments, and must return a value. Terminals must only return a value.

Functions are added to the function set using the following definition:

```
funcs["getNxNMean"] = ["IMG", FeatureExtractor.getNxNMean, ["IMG", "INT"]]
```

In this example, the functional set is represented by the *funcs* dictionary. *getNxNMean* is the name of the function. On the right hand side of the variable assignment, the list structure encodes the output type of the function, the reference to the actual function itself, and finally the input type for each parameter of the function.

Terminals are defined similarly, but only specify a value, or function that returns a value, that values type, and no input parameter types (as there should not be any).

```
terms["100i"] = ["INT", 100]
```

There are many different functions available to GPT. A description of the more advanced components is available in Section 5.2.

5.1.3.2 Output Vision System

The VisionSystem component invoked by the Main component using the evolved segmenter and classifier, along with the functional and terminal sets, and the DRS2 slots. With this information, the VisionSystem component can execute a complete segmentation and classification procedure on any given input image. This component, once set up with the appropriate data, is then converted into a binary file. This facilitates transference of the trained vision system to another computer or allow long-term storage for later re-use. This VisionSystem is the output of a training run, where the GP system is trained on a set of images. This component is used for inference of the same kind of images as the GP system was trained on. The system does not need to be retrained for every image passed for inference, but as this is a learned classification system, only images similar to the training data can be expected to be classified.

5.1.3.3 Vision System Runtime (VSR)

This component loads the binary VisionSystem file, and converts it back into a representation that can be executed on an image.

5.1.3.4 LISP Interpreter

This is a third party LISP interpreter that can evaluate LISP programs inside a python runtime. It is used by the GP Component and the VSR component to create and run the evolved LISP programs respectively.

5.2 Genetic Components for Computer Vision

In order for programs that can segment and classify images to be evolved, some genetic components must exist that facilitate such a process. For example, if the only functions available were mathematical operations, then evolving a segmenter would be impossible without an additional function that can read an image and specify a pixel to run those mathematical operations on.

As mentioned above, the initial feature extractors available to GPT were primitive, working on a pixel by pixel basis. These include operators such as the mean, sum, or variance of an *N* by *N* area of pixels and returning an image with those values defined for some region of pixels. These tend to be selected by the DRS2 fitness function, as that function also works on a pixel-by-pixel basis.

In an effort to overcome this, several more advanced feature extractors were implemented, such as canny edge detection, template generation and matching, and notably, classifying regions based on similar shapes. This function was the first step towards more region-of-interest based classification rather than pixel-wise classification. (Canny edge detection and the automatic filling of canny-detected regions was used occasionally by the segmentation program.)

This function utilises contour selection to find regions in the image, and then applies a shape-matching functionality to each region. Similar shapes are set to have identical pixel values in the output image. This initial, simple approach to feature extraction was, however, never selected by the DRS2 fitness function. In order to push the genetic algorithm towards using its available computer vision functions to identify regions, rather than pixels, a new fitness function would have to be designed that rewards this behaviour.

Over the course of this project, several new feature descriptors capable of describing entire regions of images were implemented, described in section 5.6.

5.3 Cost Functions

Two cost functions were developed to improve upon the accuracy and efficiency of DRS2, based around the idea that the output of an evolved classifier is an image with labelled pixels. Both these fitness functions strive to ensure that regions of the image that have the same class have the same pixel value in the output image.

5.3.1 High Level Features Cost Function

The first developed fitness function with this idea was *High Level Features (HLF)* . This fitness function uses contour detection to find the regions of interest and classes in the training image and the output image of the evolved classifier. A contour is a representation of the edge of a region.

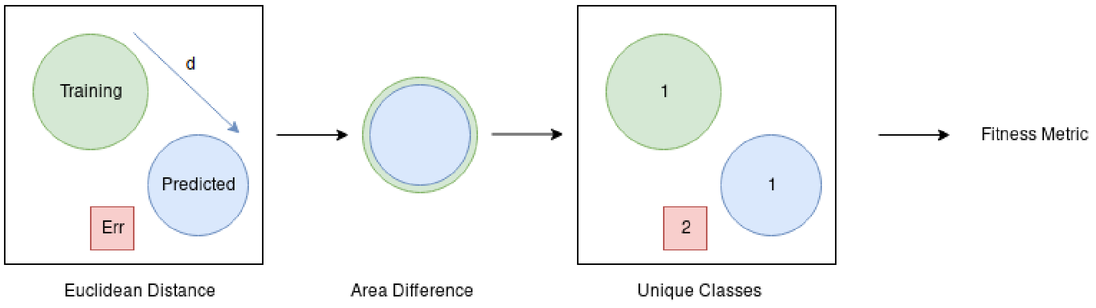


Figure 5.6: High level features cost function algorithm

With contours extracted, the Euclidean distance between the centroids of the training contour and the output contour can be found. In a perfectly accurate output, the distance between predicted contours and training contours would be zero, hence this metric can be used as one metric for error. An assumption is made that the closest predicted contour to the training contour is the matching class for that training contour. Fig. 5.6 shows this process. In this diagram, the algorithm has taken the 'Predicted' circle to be the closest contour to the 'Training' circle, and hence found the square (labelled 'Err') to be erroneous.

Then the number of unique pixel values in the image is counted. As this fitness function is attempting to push the genetic algorithm to evolve a program that labels regions with the same class the same pixel value, and different classes, different pixel values, the amount of unique pixel values (excluding the background) should be equal to the amount of classes. In Fig. 5.6, the algorithm successfully labelled the predicted circle with the right label as the training circle, while labelling the differently-classed square with a different label.

Finally, the area of the matched contours are compared. If the region of interest has been labelled correctly, and fully, by the evolved classifier, than the area should be the same between the training contour and the predicted contours. All these error values are summed and returned as the final fitness.

5.3.2 Cost Regions

This fitness function is a result of an effort to simplify and improve the performance of the above. While the High Level Features fitness function was successful at guiding the evolved classifiers towards classifying regions, the implementation was complex, involved expensive mathematical and image operations, and required several additional data structures to hold intermediary information.

Cost Regions attempts to fix this by extracting the core idea of the previous fitness function, that the evolved classifier should return labelled regions in an output image, and simplifying the implementation and error metrics. This is accomplished by making use of the class masks from the training data. Each class mask represents a single class, and when applied to an image, masks out all regions in that image that are not of the same class as the class mask. There may be as many classes in the class mask as has been labelled in the training data. For example, if there are five classes in the training image, the class mask would correspondingly have five classes, each with an annotation that says which class that area of the training image represents.

Each class mask from the training data for the given training image is applied to the output image. Then, the variance of the remaining pixels is calculated. If all the pixels have the same value, then the variance will be minimal. The most common pixel value is extracted, and assumed to be the value that the GP system is guiding the classifier towards for the given regions.

The most common pixel values are stored. As each class should be assigned a different pixel value, each image should end up with the correct number of unique pixel values. For this cost function, the intra-class variance of pixel values is being minimised, and the inter-class pixel values maximised, to be unique from each other.

This fitness function can be described as follows:

$$L = k\,|N_c - P_c| + \sum_{i=0}^{P_c} Var(C_i)$$

Where N_c is the number of classes, P_c is the number of predicted classes, $Var(C_i)$ is the variance inside the predicted class (which we sum over all predicted classes), and k is a constant that elevates the importance of having the correct number of classes. L is the final fitness. This function is minimised. To give a perfect example, if there are 5 classes, 5 classes are predicted, and there is no variance inside the predicted class (i.e every pixel is labelled correctly), then the output of the loss function is 0, and hence, reveals a perfect program for the given training data.

5.3.3 Comparing Cost Functions

To evaluate the different cost functions, the time for each generation's fitness calculation was measured, for the first ten generations of the same problem. This means that on average, the fitness calculation was ran two hundred times per generation, ten times, for a total of two thousand executions. This measurement was repeated five times for each cost function. An average of each time for each generation was computed and then graphed, shown in Fig. 5.7

Figure 5.7: Cost Functions Compared

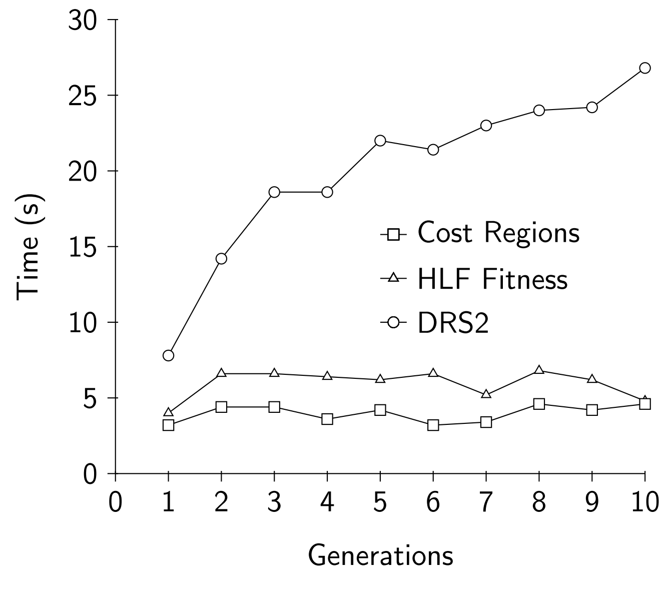


Table 5.1: Average Execution Times

	DRS2	HLF	Cost Regions
Average Execution Time (s)	20.06	5.94	3.98

Table 5.1 shows the average execution time for all runs for all generations for the three cost functions. It's clear that from this and Fig. 5.7 in terms of time spent executing the cost function, *Cost Regions* outperforms *Higher Level Features*, and also has the benefit of being easier to understand and implement. However, both *Cost Regions* and *Higher Level Features* significantly outperform the *DRS2* fitness function.

5.4 Extracting Structure from Neural Network Filters

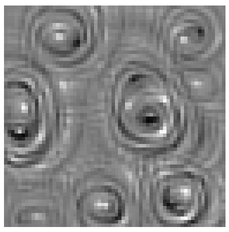
One of the reasons that convolutional neural networks work so well at image identification is their ability to learn representations of features in images, which are stored as filters in the network. These filters are passed over the image, and activate when a certain structure in the image aligns with the filter. These filters are in fact what lies along the channel dimension of a convolutional block. For example, if the third convolutional block operates on input that is 64 x 3 x 3, the first dimension represents the 64 filters that make up that single convolutional layer.

If these are the key to image recognition, rather than integrating entire neural networks to a GP vision system, one would only have to provide a set of filters appropriate to the problem at hand, and determine how closely the structure of the filter matches the structure of the input image. However, these filters are stored as arrays of neuron activation levels, rather than images that can be directly compared. To perform structural comparison, an image is needed that represents the filter.

The key to obtaining this image is that a certain filter strongly activates when it passes over the feature that it has been trained to identify. All that needs to be done to extract a representation of a filter is to generate an image that causes the filter to activate.

Gradient ascent on the pixels of the generated image is used, with the error being the inverse activation of the filter. This allows the movement of the values of the pixels in the input image to gradually move towards values that activate the filter. Starting from a randomly generated image, the final result is an image of a structure that activates the filter. Two filters showing structure that maximised some filter in the VGG16 Neural Network [69] are shown in Fig. 5.8. VGG16 is a deep neural network with 16 layers that is available pre-trained for image classification purposes.

To determine if extracted filters are useful for structural analysis, a test dataset was composed of 64 extracted filters representations from a convolutional neural network trained to identify a set of five shapes, one of which was a square. Another test dataset was composed of a set of 64 randomly generated images of the same size.


(a) Filter 1

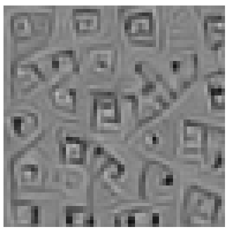

(b) Filter 2

Figure 5.8: Two interesting extracted filters from the VGG16 Neural Network

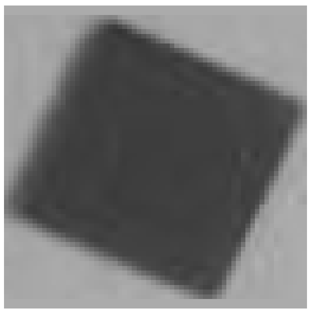


Figure 5.9: The test shape

An image of a square (Fig. 5.9) was then compared using a structural similarity index (SSIM) algorithm against both the set of extracted filters and the set of random images. The SSIM algorithm returns a value of 1.0 for an exact match. All similarity scores for each data set were stored, and an average taken for each one. Table 5.2 shows the results of running this process three times with three sets of random images.

Table 5.2: Neural Filters vs Random Images for Square SSIM

	Neural Filters	Random Images
Structural Similarity (Mean)	0.06	0.007
	0.06	0.006
	0.06	0.009

(a) A Square (b) Corresponding Filter

Figure 5.10: Square shape, with its highest SSIM filter

The random images were an order of magnitude less structurally relevant than the neural filters. The filter with the highest SSIM for the square is shown in Fig. 5.10, and a different shape shown in Fig. 5.11

(a) A Lemon (b) Corresponding Filter

Figure 5.11: A lemon shape, with its highest SSIM filter

This preliminary investigation indicates that neural filters can be used to identify different shapes. For the purposes of this research into using filters alone, these filters were extracted from the second-to-last convolutional layer of a seven layer neural network. The network itself, *in this case*, was never used for end-to-end inference by the GP system. The next section provides a method of implementing this functionality into a GP system. Unfortunately it was found that in practice these neural filters do not convey a suitable advantage to the genetic algorithm, and this line of research was in fact a dead-end. See Section 6.3 for details on the results. These neural filters are **not** the same neural networks that were eventually implemented - these are simply parts of a neural network that were cut out from the network for use by themselves. The eventual implemented system makes use of whole, end-to-end neural networks, **not** neural filters.

5.5 Classifying Regions with Neural Network Filters

In order to allow a genetic algorithm to use extracted filters for classification, it must be packaged into a genetic component that GPT may select during evolution. This component should look at the regions of interest in the image, utilise the extracted neural filters to identify similar regions, and mark such regions as similar. This component was named 'NeuroCompare'.

NeuroCompare works as described:

1. Extract all regions of interest in the image. The GP system has already segmented the image and located regions of interest. This generated segmentation mask is used to select each region in the image.
2. Compare each region with each filter using SSIM, and store result as a vector.
3. Determine the angle between all vectors.
4. Group similarly angled vectors together.
5. Determine which regions of interest have vectors in the same group.
6. Mark similar regions the same pixel value.

The comparison between each filter builds an Q dimensional vector for each region of interest, where Q is the number of filters. Different dimensions in the vector will be larger depending on which filters had a higher SSIM value. Similar shapes should have similar vectors, as identical filters should have similar SSIM values for similar shapes. This means that the angle between the two vectors in Q dimensional space should be close for similar shapes, as they should have similar vectors.

The angle between each vector can then be computed, and vectors that have a small angle between them can be grouped together. Ideally, each class with K regions of interest in the image will correspond to a group of K vectors with close angles between them, while regions of interest with different classes will have a larger angle between their vectors. However, deciding exactly how close or far the angle should be before a different class is assigned is a difficult job.

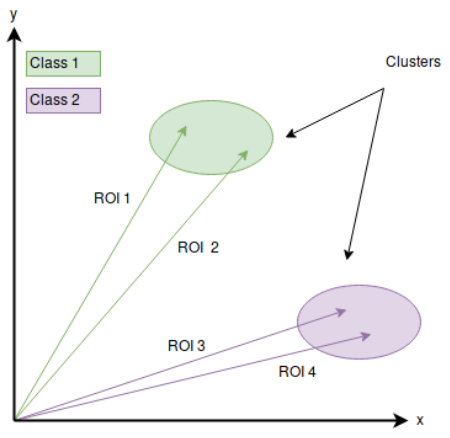


Figure 5.12: 2D NeuroCompare Algorithm

To solve this problem automatically, K-means clustering was used to group the set of vectors into N groups, where N is the number of training classes. Each cluster was examined to determine which regions of interest it contained, and regions of interest in the same cluster were marked with the same pixel value. Thus, an image with labelled regions of interest is returned. Fig. 5.12 shows a simplified two dimensional example of this algorithm. This K-means clustering simply groups together similar angles, and decides where the separation boundary between groups of similar angles should lie. This leads to an overall operator that takes an image and it's segmentation mask, and outputs another image that has each region labelled with pixel values, where the same predicted class have the same output pixel values.

5.6 Higher Level Feature Extraction

In order to move GPT towards classifying regions of images rather than individual pixels, there must also be standard computer vision operators capable of describing regions to serve as a baseline to compare to the neural networks intended to be implemented.

Eleven basic descriptors for regions of interest were implemented. These descriptors are:

- Area
- Perimeter

- Number of Holes
- Perimeter
- Roundness
- Aspect Ratio
- Rectangularity
- Rotated Rectangularity
- Joints
- Solidity
- Average Intensity

Most of these are self explanatory, however some require further explanation.

The *rectangularity* and *rotated rectangularity* descriptors output the rectangular bounding box of the region of interest. In the rotated version, the bounding box is rotated to better fit the region.

The *joints* descriptor approximates a polygon to the region of interest, and counts how many corners this approximation requires.

The *solidity* descriptor determines how convex or concave the region of interest is.

5.7 Separating Feature Extraction and Classification

Now that regions can be described through the above descriptors, a method of storing and evaluating these descriptors is required. This is accomplished through a 'feature vector' variable, that the evolved classifier can decide to use. All the region of interest descriptors require a feature vector type as one of their input parameters. This feature vector stores the output of the descriptors, and is passed throughout the evolved program as various descriptors are used. A set of descriptors that output a feature vector that describes the region can be thought of as a feature extractor.

Previously, both feature extraction and classification was performed purely on a pixel-by-pixel basis. Now, each region can have an evolved feature extractor associated with it that describes it using the available genetic components. However, this feature vector must now be evaluated to determine which class that region belongs to. The easiest method to accomplish this is to guide the evolved classifier towards outputting a single value if a certain region of interest belongs to a certain class.

To implement this, a variety of Boolean genetic components were added to GPT. These are:

- Equality
- Greater Than
- Less Than
- AND
- OR
- XOR
- NOT

The equality, greater than, and less than operators returns a Boolean from the compared integers or floats, while AND, OR, XOR, and NOT operator on Booleans exclusively. This allows comparison between elements of the feature vector, along with the initial integer and float terminals for hardcoded values such as 1, or 2.0, or 10.

For example, a region that is a triangle may end up with a feature vector that describes it has having three joints. The associated Boolean classifier for that class may decide that a region is a member of the triangle if it has a number of joints equal to three. Both the classifier and the feature extractor are evolved together, simultaneously, in the same program, to avoid having to run another evolution step. However, evolving a program that can return a True or False value depending if a region belongs to a given class, for every region, at once, is a complicated task that would require a large program depth and many branching statements. For example, to classify four regions of interest, each which is a member of a different class, the program would have to build four separate feature extractors, and then evolve a classifier that can separate out four feature vectors into a class.

Both the classifier and the feature extractor are evolved simultaneously. However, evolving a program that can return a True or False value for every region in the image at the same time is a complex problem.

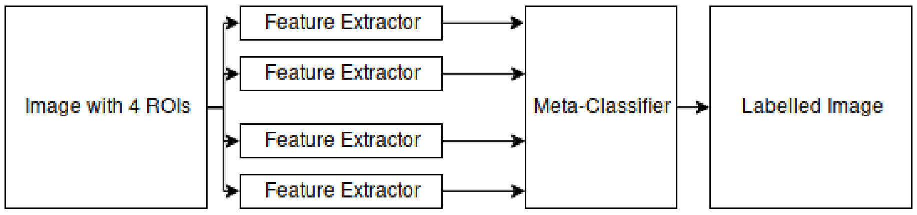


Figure 5.13: A Complicated Multiclass Classifier

It would be far simpler to separate out the training data, and simply evolve four different programs that can determine whether a certain region belongs to a given class. This is *binary decomposition*, explored in the next section.

5.8 Binary Decomposition

When humans are confronted with a problem, rarely do they tackle the entirety of the problem at once. It is natural to wish to break the problem down into smaller, more manageable chunks, solve those, and then integrate them into a working solution, than to solve the entire problem in one go. Binary decomposition is this idea formalised. In the area of evolving vision systems, rather than evolving every feature extractor and one large classifier, the problem is instead broken down into a per-class basis, turning one problem into many 'one vs all' problems.

This technique has already been applied to the area of evolved vision systems [31], where it succeeded in decreasing the time necessary to find a solution. Binary decomposition is a natural method of approaching a problem, and is relatively common among the machine learning community. It has also been successfully applied to genetic programming [70] and image classification [71]. Teredesai et al also found binary decomposition useful for evolving image classifiers, noting however that with low training data class confusion increases [72].

Each class in the training data has a feature vector and Boolean classifier evolved for it separately. The only job of each evolved program is to determine whether its input region is a member of the class that the program was evolved to solve. If it is, return a true value, if it is not, return false. For the purposes of training data, all data that has the class the classifier is being evolved to solve is seen as positive, and all other data is seen as negative.

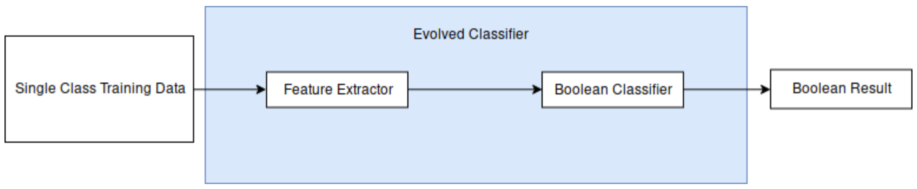


Figure 5.14: A Single Class Classifier

This greatly simplifies the task for the genetic algorithm. Once a classifier has been developed for all classes, these classifiers can be chained together, such that all classifiers return false *except* the classifier that matches the class being identified, which, if trained successfully, will return true.

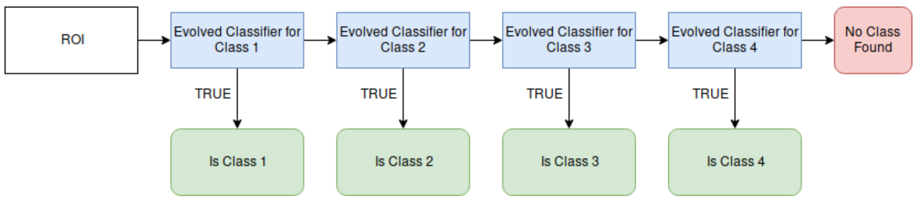

ROI
Evolved Classifier for Class 1
Evolved Classifier for Class 2
Evolved Classifier for Class 3
Evolved Classifier for Class 4
No Class Found
TRUE
TRUE
TRUE
TRUE
Is Class 1
Is Class 2
Is Class 3
Is Class 4

Figure 5.15: Binary Decomposition Classification Flow

This is the final step that was required to move towards region based classification of images. However, the output of each classifier is no longer a classified image, but a boolean value. Additionally, now N classifiers must be evolved, where N is the number of classes. This requires a new, and final, cost function.

5.8.1 A Final Cost Function

Thanks to the binary nature of the new classifiers, the fitness function can be greatly simplified. As each classifier has a specific class it is solving for, the fitness of that classifier can be determined by discovering if it returns True for all members of its class, and False for all others. The error for this fitness function is a simple count of how many items should have been classified True, and were classified False, and how many should have been False but ended up as True.

However, due to binary decomposition framing each problem as a one vs others problem, with multiple class problems, the amount of data is skewed towards 'False'. This is because most of the training data will not be True for any given class. For example, with four classes of 250 training images each, for the classifier for any given class, 250 images, when classified, should return true, and 750 should return false.

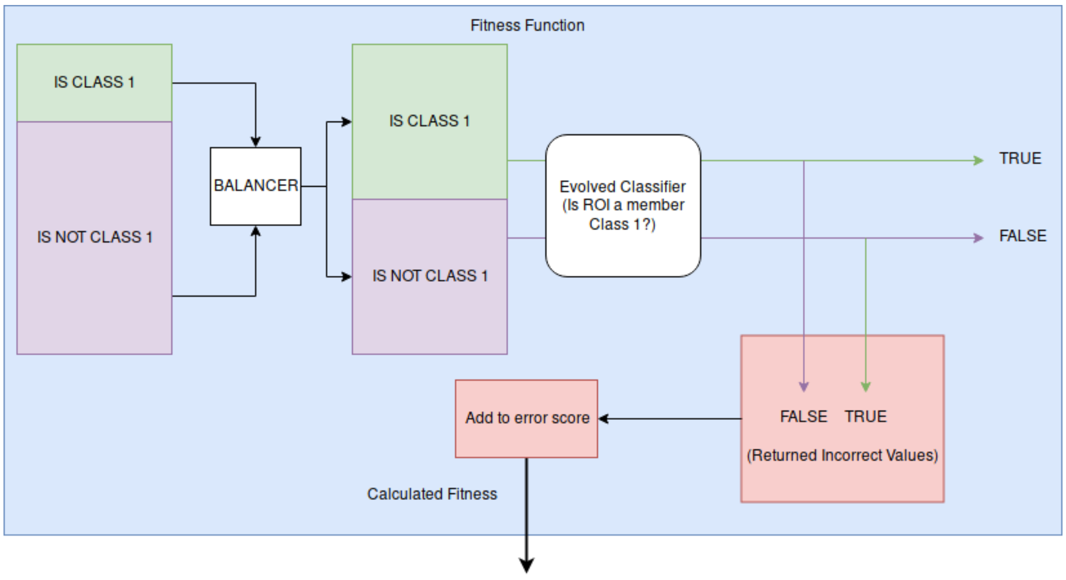


Figure 5.16: An example fitness function for a given class

To avoid the GP system becoming stuck in a local optimum by always returning false, the amount of true data for a given class is artificially increased by a scaling factor to balance the amount of true and false data. Fig. 5.16 shows the operation of the fitness function for an arbitrary class. The green and purple blocks represent classes (and hence labelled regions of interest) in the training data. This fitness function is significantly faster than previous functions as it has very few calculations to perform. The final architecture, showing how programs are trained and inference conducted is shown in Fig. 5.17.

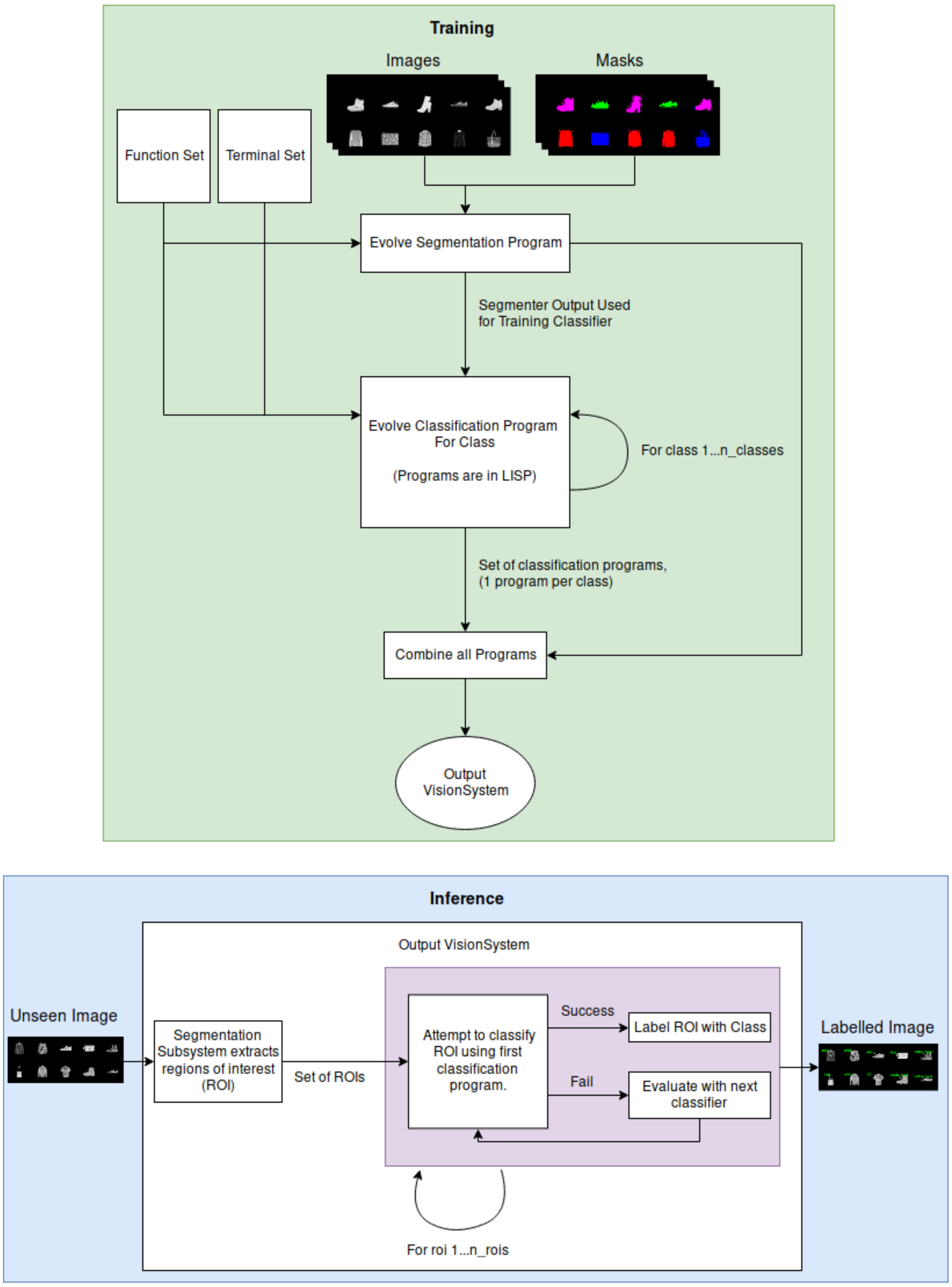

Training
Images
Masks
Function Set
Terminal Set
Evolve Segmentation Program
Segmenter Output Used for Training Classifier
Evolve Classification Program For Class
(Programs are in LISP)
For class 1...n_classes
Set of classification programs, (1 program per class)
Combine all Programs
Output VisionSystem
Inference
Output VisionSystem
Unseen Image
Segmentation Subsystem extracts regions of interest (ROI)
Set of ROIs
Attempt to classify ROI using first classification program.
Success
Label ROI with Class
Fail
Evaluate with next classifier
For roi 1...n_rois
Labelled Image

Figure 5.17: Overview of Data Processing Pipeline

5.9 Integrating Neural Networks

Now that GPT has been modified to support high level feature extraction, the next step is to integrate convolutional neural networks with the genetic algorithm. For example, rather than reinventing the wheel, and developing a complex classifier to identify the number '8' in an input image, the genetic algorithm instead makes use of a trained neural network for classification purposes instead. This has the benefit of making use of convolutional neural network's excellent image classification accuracy, yet also allow for a classifier to be evolved conventionally if no appropriate neural network exists for an arbitrary class in the input image.

Two genetic components based on neural networks are integrated. One type of neural component represents a multiclass neural network, trained over an entire data set. The other kind of neural component is a set of neural networks which have had the principal of binary decomposition applied to them. These 'decomposed' neural networks are trained to identify one element of the dataset from any others.

The purpose of the multiclass neural component is to determine whether a genetic programming system will make use of a neural network where it is appropriate to do so, and the purpose of the decomposed components is to determine whether GPT will assemble multiple neural networks to solve different problems from the same dataset.

5.9.1 Network Structure

All neural networks trained for the purposes of direct integration in this project have the same structure. They are convolutional neural networks, written in Python, making use of a machine learning toolkit known as Keras, built on a Tensorflow backend.

Tensorflow is an example of a graph computation system, that encodes all operations and variables as elements in a computational graph. As tensors move through the graph, they are modified dependent on the graph node they arrive at. This makes tensorflow excellent for neural network calculations, as a neural network can be easily described as a collection of tensors of weights, biases, inputs, and outputs.

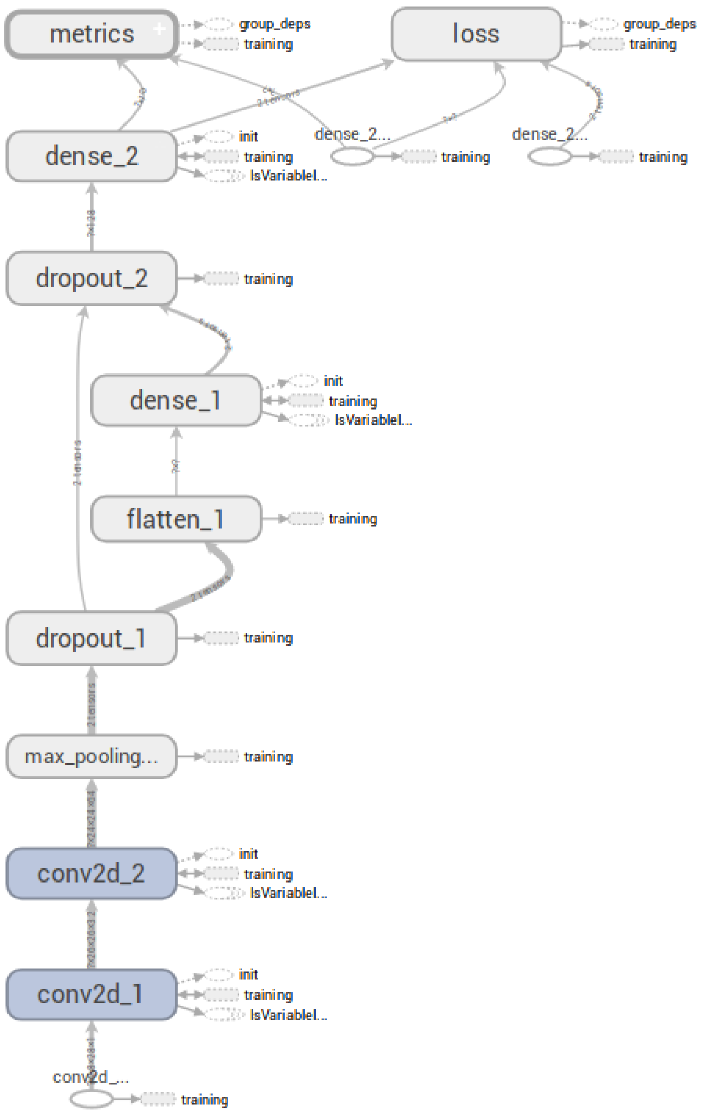


Figure 5.18: Tensorflow Graph of Neural Network Structure

Fig. 5.18 is the structure of the neural network. It is composed of two convolutional layers, a max pooling layer for representation reduction purposes, two dropout layers to help prevent overfitting, and a 128 neuron fully connected layer that is connected to a softmax layer. This network is deep enough to provide good accuracy on the datasets it was trained on, while not taking a long time to train.

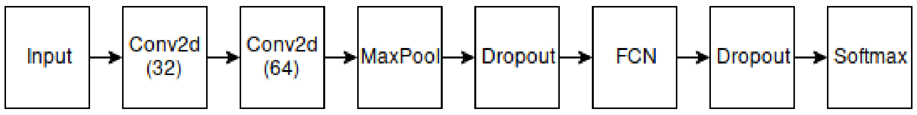


Figure 5.19: Neural Network Structure

The first convolutional layer has 32 filters, and the second has 64. Both layers have a filter size of 3x3, a stride of 1, and zero padding. All layers that require non-linearisation use rectified linear units, with the exception of the final layer that uses softmax to output class probabilities. The pooling used is 2x2 max pooling. The first dropout function uses a dropout probability of 0.25, while the second has a 0.5 probability of dropping neurons. Fig. 5.19 shows the simplified structure of the neural networks used. The networks are trained for 12 epochs on their respective datasets (in a single epoch the neural networks sees all the training data once).

5.9.2 Decomposed Neural Networks

The decomposed neural networks are networks trained on the individual parts of the same dataset as the multiclass neural network. Much like a binary decomposed GP classifier, these networks, when trained, can only determine whether an image is part of one specific class, or if it is not.

The advantage of this is that these networks require much less training data to train, and train far faster. It is easier to train a network to determine if an image is a member of one class than it is to train a network to be able to classify every class in the training set. Because the networks need less training data, they are much faster to train, and hence an accurate neural network can be achieved much faster than training a multiclass neural network, even taking into consideration that N networks must be trained, where N is the number of classes in the dataset.

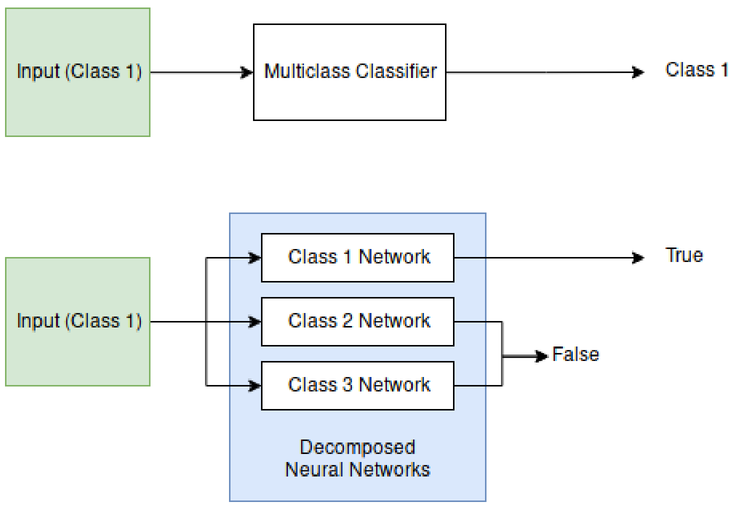


Figure 5.20: Multiclass vs Decomposed Networks

The disadvantage in these lies in that they are less useful to a human as now the appropriate decomposed network must be chosen manually for the task rather than using a network that solves every task. By integrating these with a genetic algorithm, this choosing can be automated, and the correct network can be chosen for the task at hand by the evolved classifier. Fig. 5.20 shows the difference in the network types.

Figure 5.21: Validation Accuracy of Decomposed Networks vs Multiclass

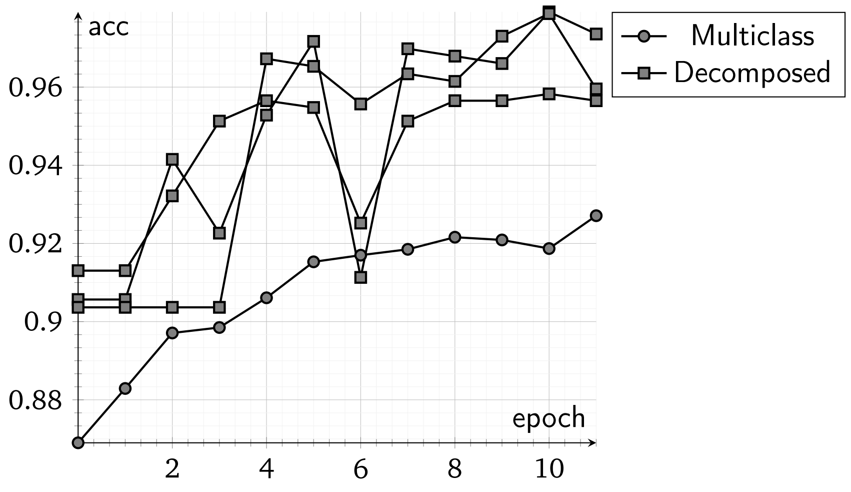


Figure 5.22: Accuracy vs Time of Decomposed Networks vs Multiclass

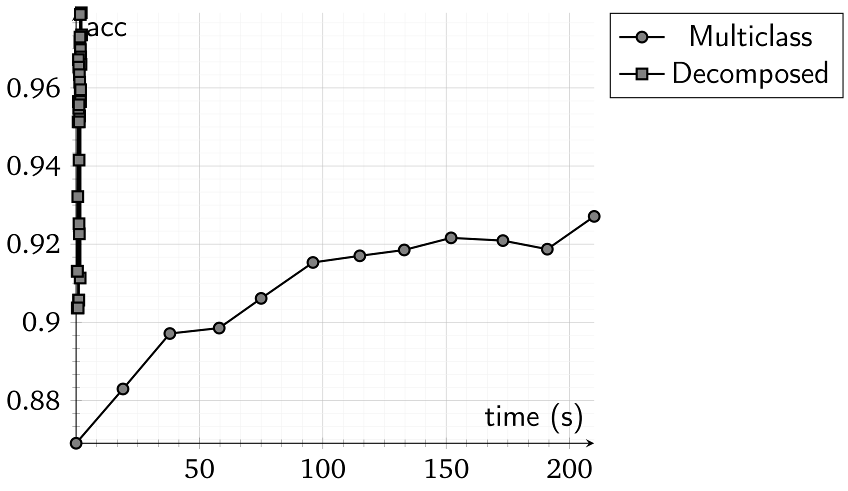


Fig. 5.21 shows the accuracy of these neural networks on a validation set of one of the datasets used in this research. All four networks have similar accuracy, however, as Fig. 5.22 shows, the decomposed networks train significantly faster than the multiclass network, to the point where the data points are insignificant on the graph compared to the multiclass training time.

5.9.3 Integration

Each network is a self-contained module with each module handling executing the network given an input, and returning the output of the network back to the genetic algorithm. Said output is stored into the feature vector of the classifier. These functions are grouped, and added to the functional set of the GP system. Any evolving classifier can now make use of any convolutional network, as long as a function mapping input and output between the GP system and the neural network exists.

Typewise, a neural network genetic component can only connect to other genetic components that return a feature vector type or a region of interest (image). Two components may feed each type into one neural component. The return type of the neural network component is another feature vector, with data added to represent the output of the network. An example program tree that could be generated by the final system is shown in Fig 5.23. (A more complex example can be found in Appendix A).

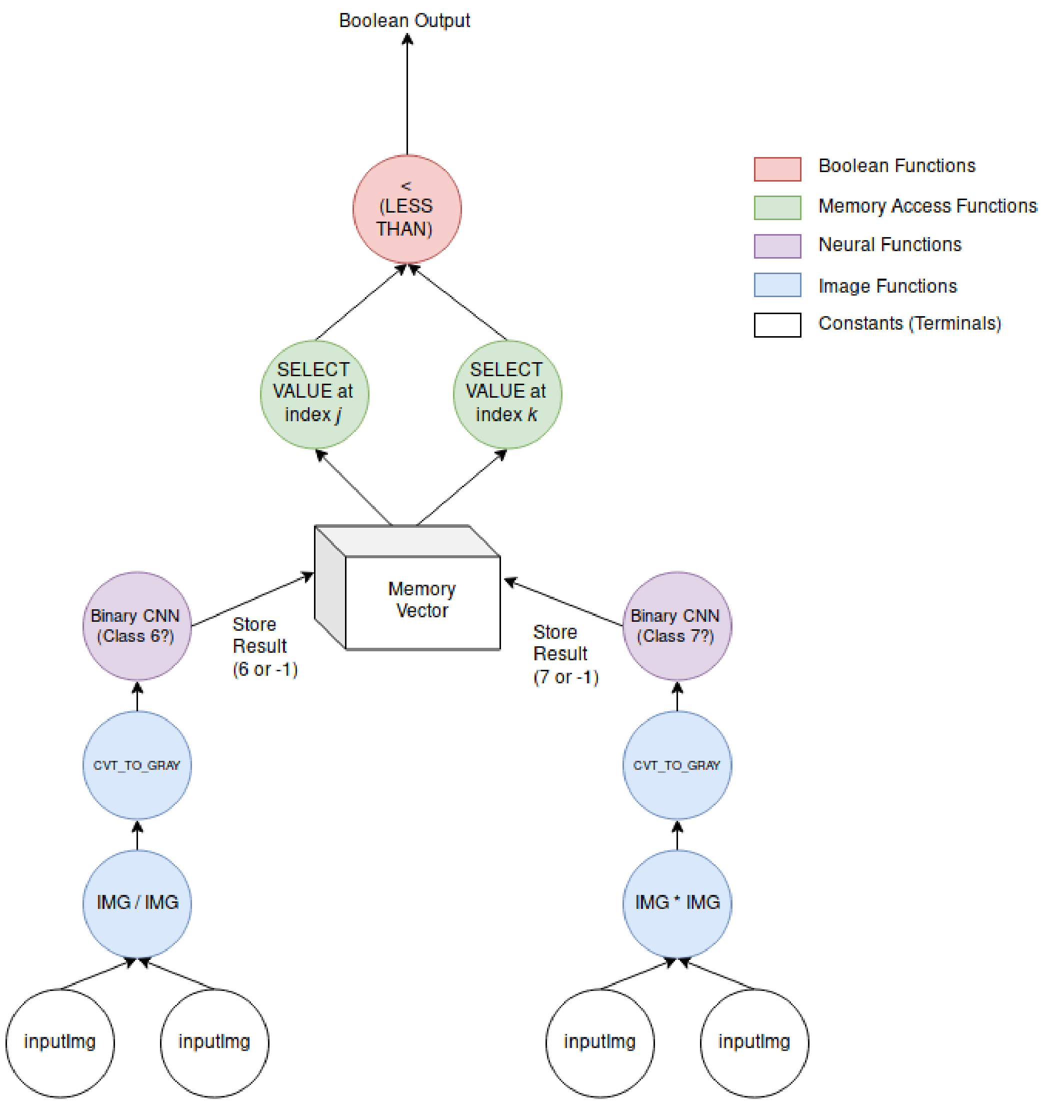


Figure 5.23: Example Generated Program Tree

As can be seen in Fig 5.23, multiple neural networks may lie on separate branches of the program tree, and that the output of one, two, or potentially more neural networks may be combined to determine the final classification. The example shown was a generated attempt to solve an MNIST classification problem.

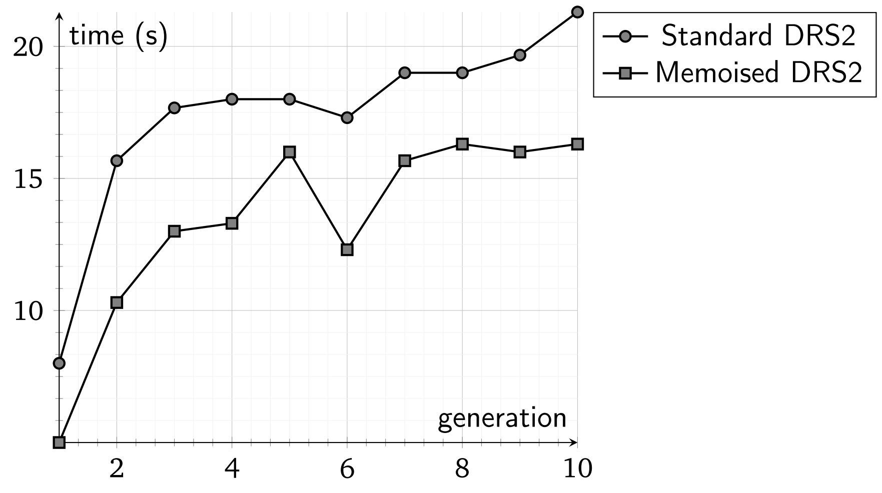
5.10 Performance and Optimisation

Training a genetic algorithm can take a long time. For the problems in this research, training times of eight hours or longer are not uncommon. Two different optimisation techniques were explored to reduce this time and optimise performance as much as possible.

5.10.1 Memoisation

Memoisation is an optimisation technique that stores the results of functions, essentially trading time complexity for space complexity. When the function is called with the same parameters as the stored result, then the stored result is returned rather than running the function again. This means that computationally intensive function calls can be returned faster if the function is deterministic and the input parameters are the same.

Figure 5.24: Effect of memoisation on DRS2



Memoisation was implemented for many of the genetic functional components. Results were stored in memory in a dictionary structure. Every *N* calls, the memoisation cache was cleaned, where *N* is a tunable parameter based on memory size. Fig. 5.24 shows the effect that applying memoisation has on DRS2. Three runs of ten generations were evaluated for both memoised and non-memoised versions of DRS2. The average execution time for each generation was then computed. As shown, memoised DRS2 shows a mild improvement over standard DRS2.

5.10.2 Multiprocessing

The main computational time sink in GPT is the evaluation of the fitness function for a specific individual. However, the fitness function itself is not dependent on any other values aside from the output of the evolved individual. The individual is also self contained. These properties mean that fitness function evaluation in GPT lends itself well to multiprocessing.

By evaluating each individual's fitness separately on a different processing core of the evaluating machine, the time to evaluate the cost of an entire generation falls dramatically. If two cores are used, the processing time can be expected to halve! With three cores, the processing time would be around a third of the time with one core.

Figure 5.25: Processing on 3 cores vs 1 core (Memoised)

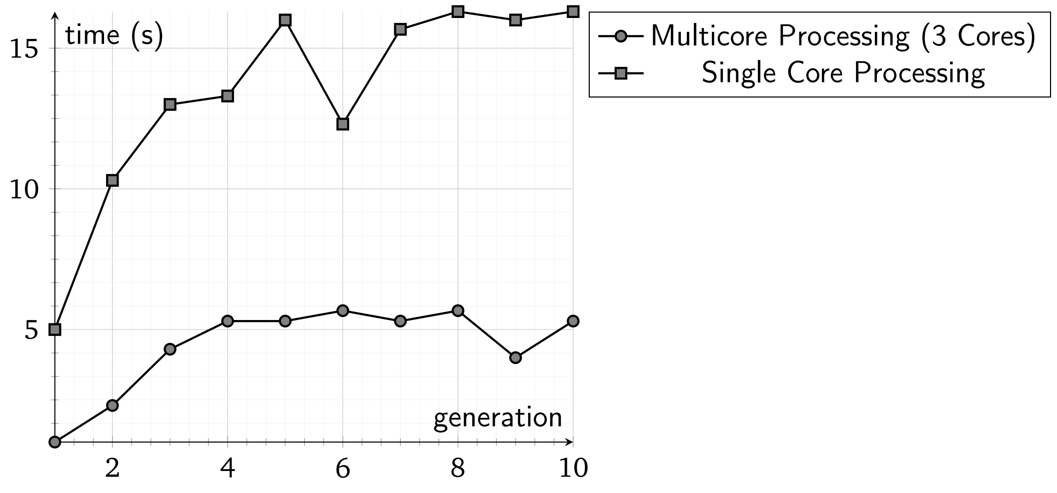


Fig. 5.25 shows the effect of running evolution of the same problem on a system with three available cores versus a single available core. The multicore system evaluates each generation significantly faster than the single core system. Memoisation was active for both systems. As before, three runs of each problem was performed, and the average execution time taken for each generation.

CHAPTER 6

Results and Analysis

This chapter discusses the datasets used for testing the Standard GPT and Neural GP system, and examines the results from all of the variants of GP components that were developed over the course of this project. Baselines for each dataset are discussed, including both simplistic statistical models and the result of a GP system that does not make use of any neural components. Both multiclass neural network components and decomposed neural network components are tested and the results analysed in this chapter. Nearly all results examine the recall and precision over each class and specifically, include the F1-Score of each class. This score is the harmonic mean between recall and precision, and for a multiclass classifier, provides a more accurate method of evaluating which classifier and class performed 'best', compared to a simple dataset wide accuracy measure (though in certain cases, this measure is revealed to give the reader a more general idea of performance).

This chapter begins with an overview of the datasets used and how training and validation data was generated, then progresses to determining how well the original GPT system works alone, then when combined with neural filters, and finally the baselines are discussed and the effect of neural network components on the GP system evaluated.

6.1 Overview of Datasets Used

Three datasets were used over the course of this project to determine the effectiveness of the various techniques developed. These datasets consisted of:

- A set of shapes
- The MNIST Dataset
- The Fashion MNIST dataset

The shapes were used to test the extracted neural filters and to provide a baseline for the initial GPT system. The MNIST and Fashion MNIST datasets were used to verify how well the neural network components integrated into GPT. The MNIST and Fashion MNIST datasets were used to test neural integration as they are much more complex datasets than geometric shapes, making the GP system more likely to evolve to use pre-trained neural networks to avoid having to evolve complex conventional classifiers. Identifying geometric shapes is an easier problem, suitable for testing how well extracted neural filters work, and demonstrating the per-pixel confusion DRS2 is prone to. Fashion MNIST and MNIST also have the advantage of being slightly closer to real-world problems than simple shape analysis.

6.1.1 Datasets Description

6.1.1.1 Shapes

This dataset is a set of nine images, each image consisting of ten shapes. The shapes are:

- Square
- Star
- Sunburst
- Lemon
- Flower

Each shape appears, on average, twice in each image. The dataset is in colour, but each class (of which there are five) is defined by the type of shape it contains. An example training image from the set is shown in Fig. 6.1 The colour attribute simply offers another potential classification grouping (eg, ‘all red shapes’ or ‘all blue shapes’). These classification groupings were not used, in favour of the shape grouping (‘all squares’). Thus, the purpose when classifying this dataset is to ensure all elements of the same class are detected as *being* the same class.

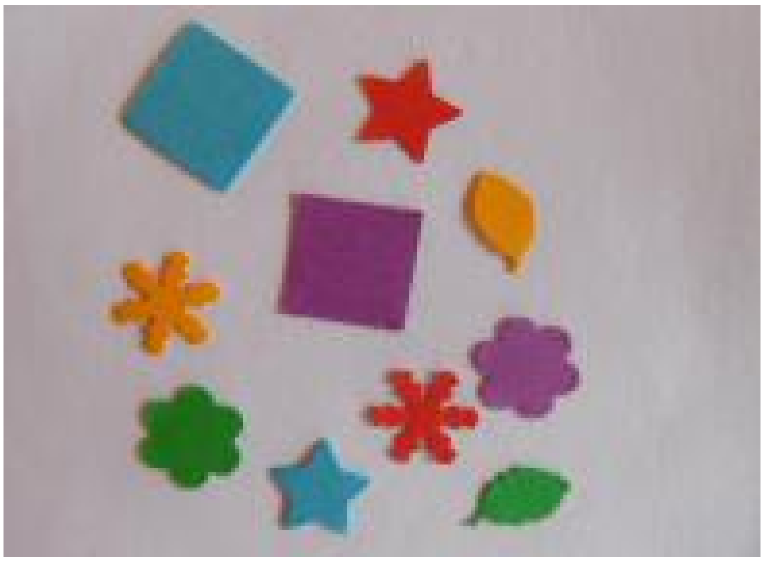


Figure 6.1: Shape Dataset

6.1.1.2 MNIST and Fashion MNIST

MNIST [73] is a dataset composed of 70000 28x28 pixel images. There are 60000 images in the training set, and 10000 images in the validation set. Each image is a hand drawn digit, ranging from 0 - 9, white against a black background. MNIST was originally created by Yann LeCunn.

Fashion MNIST [74] is a dataset built in a similar manner to MNIST, only rather than 70000 greyscale images of numbers, it is composed of 70000 greyscale images of various clothing items. As with MNIST, there are 60000 training images and 10000 validation images, and there are ten different classes of clothing. As this image is greyscale, and made up of complex clothing items, classification on this dataset is appropriately more difficult. The items in the dataset are:

- T-Shirts
- Trousers

- Pullovers
- Dresses
- Coats
- Sandals
- Shirts
- Trainers
- Bags
- Ankle Boots

Fashion MNIST was created by Zalando Research.

6.1.2 Dataset Generation

GPT is designed to examine all classes in an image at once. For example, by providing GPT with several labelled images similar to Fig. 6.1, it could be trained to classify those shapes in similar images. However, the images in the MNIST and Fashion MNIST datasets are individual images of the members of each class, as shown in Fig. 6.2. Building an image out of the entire MNIST or Fashion MNIST datasets would result in an impossibly large training image. It makes sense to choose a few images from the various classes of MNIST and Fashion MNIST, and train on those.

(a) MNIST Image (b) Fashion MNIST Image

Figure 6.2: Individual images from MNIST and Fashion MNIST

In order to convert these datasets into a format that GPT can use, a selection of the MNIST and Fashion MNIST datasets are combined into larger images, that can then be used to train the GPT system. These images are each made up of four classes from the MNIST and Fashion MNIST datasets, and a total of ten items. The amount of each class in each training image is random.

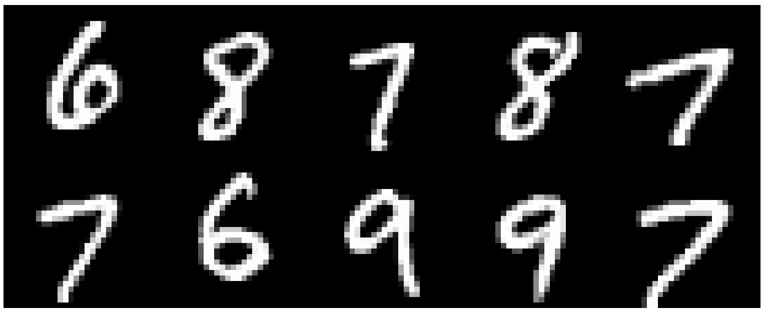


Figure 6.3: MNIST Training Image

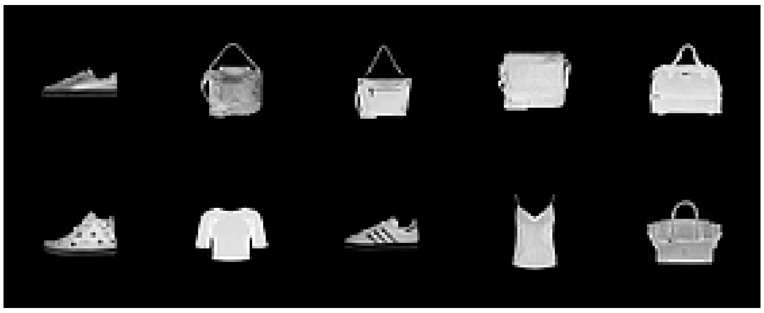


Figure 6.4: Fashion MNIST Training Image

The built MNIST training set is composed of four images such as Fig. 6.3 and the validation set is another four images of similar structure, and the Fashion MNIST training set is six training images similar to Fig. 6.4, and four validation images. This may appear low, but one advantage of evolved vision systems compared to other techniques is that they are often able to make use of low amounts of training data and still obtain reasonable results [31], at least compared to neural networks.

Labelling these training images is usually accomplished by hand, with the user using a 'painting' function that sets pixels to be of a certain class. This painted image becomes a mask for the training set. Manually labelling these datasets would be a time intensive task. However, both MNIST and Fashion MNIST consist of light images, on a black background. In fact, every black pixel in the image is not a member of a region of interest. This means that when composing the training images, by combining individual 28x28 images from the MNIST or Fashion MNIST dataset, a corresponding mask file for each class can be automatically built, with no human intervention.

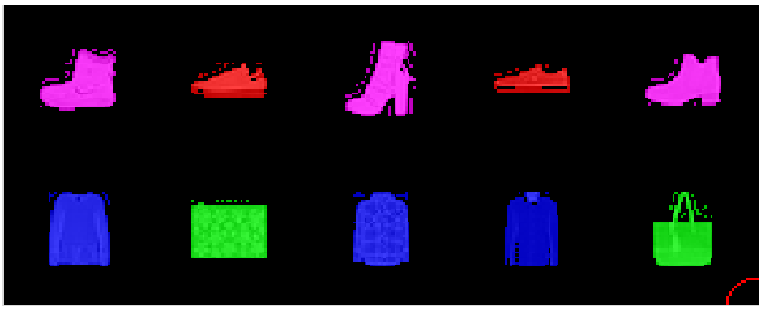


Figure 6.5: Fashion MNIST Coloured Mask

Fig. 6.5 is a coloured representation of the training mask. While not a perfect outline of each class, the small inconsistencies seem to make little difference in practice. Each different colour represents one class. For both MNIST and Fashion MNIST, to reduce training time, images with only four classes from each dataset are used.

6.2 GPT

For comparison, the results from a shape classifier evolved by the original GPT system, using the DRS2 cost function are shown below in Fig. 6.6. For reference, the square class is coloured in red, star in green, hexagon in blue, lemon in pink, and sunburst in cyan. These colours are intended to convey classes that have been detected as the same. For example, all squares were classified as red, and all sunbursts as cyan. These images are all outputs of the classification system.

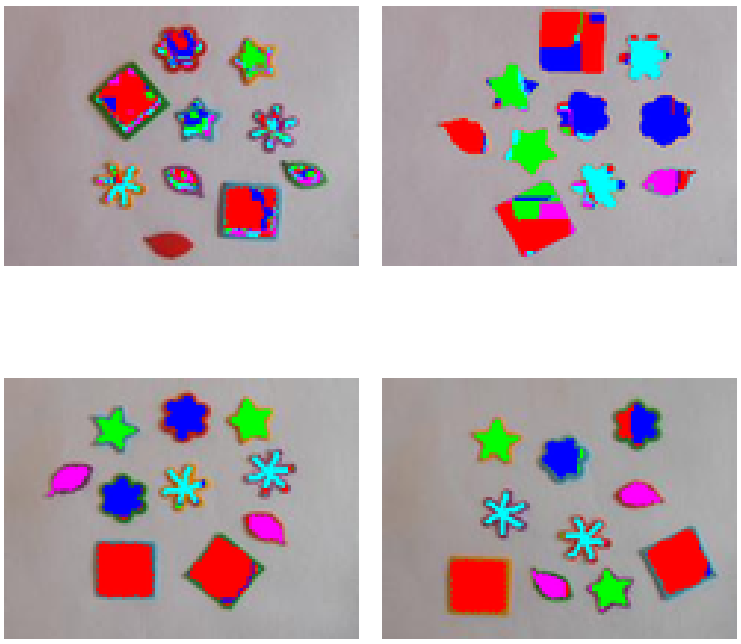


Figure 6.6: DRS2 Classified Shapes Dataset

As can be seen in the images, while DRS2 is capable of producing decently classified images, there is also evidence of extreme per-pixel confusion, that render the result difficult to interpret. Each different colour in the image is intended to be a separate class. For example, in these images, the square class should be highlighted red. However, due to the pixel confusion, it is hard to determine exactly which class certain shapes have been classified as; in poor-performing images, the shapes often become a mosaic of several different classes.

Table 6.1: DRS2 Shape Confusion

	Square	Star	Hexagon	Lemon	Sunburst
Confusion (%)	18.2	45.1	42.6	45.4	17.4

To determine the confusion in each class, the number of correctly classified pixels for each class were compared with the number of incorrectly classified pixels, for each class, for each validation image. The results are shown in Table 6.1

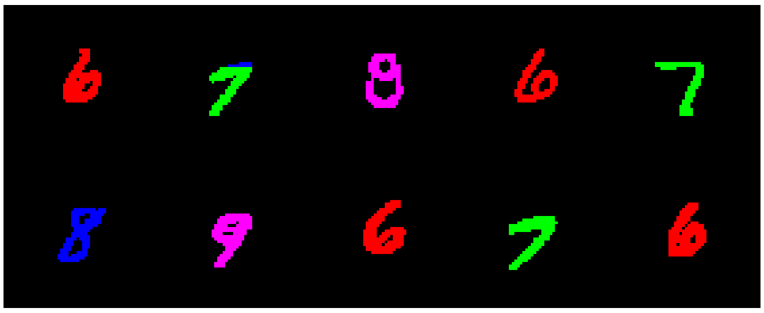


Figure 6.7: MNIST Classified Training Example

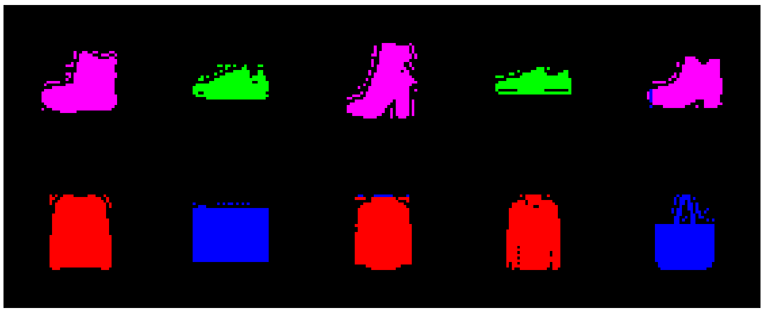


Figure 6.8: Fashion MNIST Classified Training Example

For the sake of completion, GPT with DRS2 was also tested on the MNIST and Fashion MNIST dataset. The results for the training data were good, with low amounts of per-pixel and per-class confusion (shown in Fig. 6.7 and 6.8). However, on the validation set, both evolved classifiers failed to identify any items as belonging to the same class, with a class confusion rate of 100%. This indicates that for each image the evolved classifier simply learned the positions of the training data, rather than extracting any features from them. As it stands, the initial GPT system appears to lack the expressiveness to classify those datasets.

6.3 Neural Filters

To generate the neural filters, a three layer convolutional neural network was trained. This is similar in structure to the neural networks trained for the purposes of integration, with the exception that this network has one more convolutional layer. This additional layer generates filter images with clearer evident structure (comparison shown in Fig. 6.9)

(a) Filter from third convolutional layer (b) Filter from first convolutional layer

Figure 6.9: Two filters from different layers of the network.

This network was trained on the shapes dataset, and then the extracted filters made available to the 'neuroCompare' component. The GP system was then trained on the same dataset, to determine whether the filters would be deemed useful, and if the component is used, how accurate the evolved program is. Three training runs were performed. In all instances, the neuroCompare operator was selected and used.

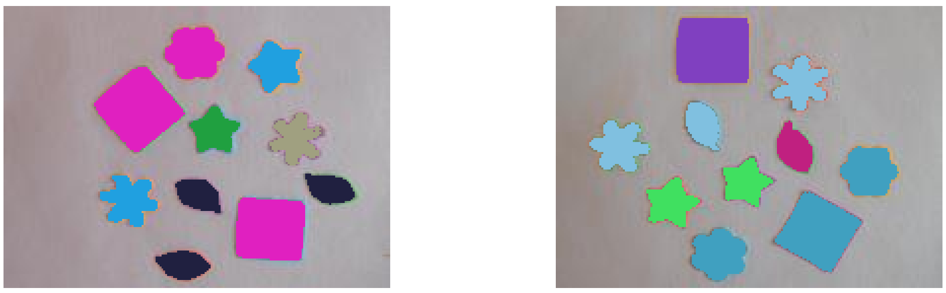

(a) Example 1 (b) Example 1

Figure 6.10: Sample classified images.

Fig. 6.10 is the output of an evolved program that utilises neural filters for classification purposes. Unlike DRS2, there is no evidence of pixel confusion, indicating that the GP system is examining entire regions rather than each individual pixel. However, while the filters were selected, the general accuracy of the evolved classifier is low, with a high degree of class confusion. This indicates that the extracted filters do not have enough structure to perfectly identify every class from each other.

Table 6.2: Results from 3 Evolved Classifiers

Classifier	1			2			3		
Class	Recall	Precision	F1-Score	Recall	Precision	F1-Score	Recall	Precision	F1-Score
Square	0.88	0.54	0.67	0.88	0.21	0.34	0.63	0.71	0.67
Lemon	0.50	0.31	0.38	0.00	0.00	N/A	0.50	0.57	0.53
Star	0.75	0.67	0.71	0.25	1.00	0.40	0.50	0.80	0.62
Hexagon	0.25	0.40	0.31	0.25	0.67	0.36	1.00	0.53	0.70
Sunburst	0.75	0.75	0.75	0.13	0.50	0.20	0.63	1.00	0.77

Table 6.2 displays the precision, recall and F1 score for each class, classified by separate programs evolved for 250 generations each. All evolved programs use the neuroCompare component for classification purposes. As the data shows, the F1 scores are generally low for most classes. The most consistently classified class was clearly the Square shape, that in general had the highest recall and precision compared to the other classes. This is likely due to the simple geometric shape generating well defined structural similarity vectors that are unlikely to be confused with other classes by the clustering algorithm.

The lemon class also proved extremely difficult to classify, and one evolved program failed to recognise it at all. This is likely due to its nondescript shape, and rotation-variant tail that may not have been captured well by the neural filters. However, considering that a random classifier would, on average, have an F1-Score of 0.2 for each class, the results are better than random chance. Despite this, certain classes did have classification results close to this value. This suggests that not enough structure was extracted to confidently classify every class.

It appears that this shortcut method for neural integration is not a suitable way to combine neural networks and genetic programming. If this method performed poorly on this relatively simplistic toy dataset, it is unlikely to perform well on a much more complex one. The next sections move onto investigating more complex datasets, and evaluating the performance of evolved vision systems with integrated neural networks.

6.4 Baseline and No Neural Components

To progress onto more real world data, such as the MNIST and Fashion MNIST datasets, it is useful to generate some simplistic baselines for comparison purposes.

These classifiers are described below:

- The ZeroR Classifier
- Random Chance
- Geometric descriptors as described in Section 5.6

Each dataset has five potential classes that could be returned by the evolved classifiers. Four classes which are members of the dataset, and a ‘fail’ pseudoclass the indicates the class could not be classified. For a random classifier, this gives each class a 1 in 4 chance of being classified correctly (as the fail class can be excluded). Table 6.3 show the results of a random classifier on the MNIST datasets, and table 6.4 show the results on the Fashion MNIST datasets.

Table 6.3: MNIST Random Classifier

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
Six	0.25	0.25	0.25	0.25	0.25	0.23	0.24	0.25
Seven	0.25	0.275	0.26		0.25	0.27	0.26	
Eight	0.25	0.175	0.21		0.25	0.25	0.25	
Nine	0.25	0.3	0.27		0.25	0.25	0.25	

Table 6.4: Fashion MNIST Random Classifier

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
Shirt	0.25	0.275	0.26	0.25	0.25	0.25	0.25	0.25
Trainer	0.25	0.3	0.27		0.25	0.25	0.25	
Bag	0.25	0.325	0.28		0.25	0.25	0.25	
Ankle Boot	0.25	0.1	0.14		0.25	0.26	0.25	

However, random chance is often not the best performing simple classifier. To obtain a better result without having to perform any machine learning at all, the ZeroR classifier can be used. ZeroR finds the most common class, and classifies every example as that class. While this renders obtaining precision and f1-scores for every class impossible, general accuracy of the entire classifier and for the most common class can be calculated. This is shown in table 6.5 for MNIST, and table 6.6 for Fashion MNIST.

Table 6.5: MNIST ZeroR Classifier

	Most Common Class	Precision	Recall	F1	Total Acc Over Dataset
Built Dataset	9	0.25	1	0.4	0.25
1000 Images	7	0.3	1	0.46	0.3

Table 6.6: Fashion MNIST ZeroR Classifier

	Most Common Class	Precision	Recall	F1	Total Acc Over Dataset
Built Dataset	Bag	0.25	1	0.4	0.25
1000 Images	Ankle Boot	0.26	1	0.41	0.255

Beyond these statistical methods, three classifiers were evolved for both datasets, and the classifiers with the best fitness were selected to serve as baselines. The classifiers were evolved for 750 generations, and use no neural components, only standard descriptors. It is expected that the evolved classifiers using conventional components perform better than the simple classifiers above on the standard built datasets that are similar to the training data. The 1000 image datasets consist of many more examples, which may not have had any similar images in the training set, which may hinder the classifiers. The results are shown in tables 6.7 and 6.8.

Table 6.7: MNIST, No Neural Components

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
6	0.10	0.10	0.10	0.18	0.19	0.18	0.18	0.22
7	0.55	0.55	0.50		0.66	0.23	0.34	
8	0.00	0.00	N/A		0.00	N/A	N/A	
9	0.00	0.00	N/A		0.00	N/A	N/A	

Table 6.8: Fashion MNIST, No Neural Components

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
Shirt	0.27	0.5	0.35	0.33	0	0	N/A	0.16
Trainer	0.33	0.8	0.47		0.33	0.85	0.48	
Bag	0.46	0.67	0.55		0.3	0.64	0.41	
Ankle Boot	0	0	N/A		0	0	N/A	

In general, for the Fashion MNIST datasets, where the GP system was able to evolve a suitable classifier, the evolved program classifies the items from both the built dataset and the general 1000 image dataset better than either the random or ZeroR classify. The F1 scores of successful classifiers are better than either the random or ZeroR classifier. However, in one case in the built dataset, and two cases in the 1000 image dataset, the evolved classifier failed to classify any items of the ankle boot and/or shirt class. This is likely due to both a weak evolved classifier for the Ankle Boot, combined with the geometric operators being unable to extract enough information purely from the shape and internal features of the regions of interest. As expected, when ran against the built validation set, the classifier outperforms both the random and ZeroR classifiers by around 7%.

For the MNIST datasets however, accuracy is worse than both the random classifier and the ZeroR classifier. This is likely due to the program classifying many of the classes as belonging to one single [4] class. The data shows class 7 appears to be classified more accurately than the other classes. There are also more cases of failed classifiers in the MNIST dataset compared to the Fashion MNIST dataset. This is likely because geometric components fail to capture the intricacy of handwritten digits; there is not enough variation in the dimensions the conventional components examine to separate the classes.

This section concludes having created three baseline datasets that future evolved programs using neural components can be compared against. Two of the baselines are simple statistical measures, and one baseline allows direct comparison between a conventional vision GP system, and a GP system with access to neural networks. Idealistically, neural networks should convey an advantage to the GP system, and classification ability should improve. The results from attempting this are examined in the next section.

6.5 Multiclass Neural Networks as Components

Each neural network component is one of two types. The first type, as described above, is a multiclass neural network, trained on either the entire MNIST or Fashion MNIST dataset. The second type is a neural network trained only to separate one MNIST or Fashion MNIST class from all others. This section concerns the effectiveness of multiclass neural networks components, that is, neural networks that have been trained to classify all classes in the dataset rather than just a single one.

To evaluate the efficacy of evolved programs that contain multiclass neural networks, two methods are used. First, the program is ran against the validation set of its corresponding dataset (as similar to that shown in Fig. 6.4). This dataset is referred to as the 'built validation set'. This indicates how well the evolved program would classify against data similar to the training data. After this, the program is then ran against 1000 images from the MNIST or Fashion MNIST validation set. This indicates how well the evolved program can classify against a more general dataset. Three runs of the genetic algorithm resulted in three programs, each of which was tested separately against the appropriate validation dataset. Each program was evolved for 750 generations.

6.5.1 MNIST

6.5.1.1 Built Validation Set

This section discusses the results against the built validation set. Precision, recall, and F1-Scores are calculated for each class, and shown in table 6.9. Then, a combined confusion matrix of all runs is shown in table 6.10.

Table 6.9: MNIST Results

Run	1			2			3		
Class	Recall	Precision	F1-Score	Recall	Precision	F1-Score	Recall	Precision	F1-Score
Six	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	N/A
Seven	0.36	1.00	0.53	0.18	1.00	0.31	0.73	0.89	0.80
Eight	1.00	0.88	0.93	1.00	0.88	0.93	0.43	0.38	0.40
Nine	0.67	1.00	0.80	0.58	1.00	0.74	0.50	0.27	0.35

Table 6.10: MNIST Combined Confusion Matrix

	Six	Seven	Eight	Nine
Six	20	0	0	0
Seven	0	14	0	2
Eight	1	0	17	6
Nine	9	3	4	21
Fail	0	16	0	7
Total	30	33	21	36
Precision	1.00	0.88	0.71	0.57
Recall	0.67	0.42	0.81	0.58
F1-Score	0.80	0.57	0.76	0.58

6.5.1.2 1000 Images from MNIST Validation Set

The MNIST evolved classifiers were then tested against 1000 relevant images from the MNIST validation dataset. The results for each run are shown in table 6.11 and a combined confusion matrix in table 6.12.

Table 6.11: MNIST Results

Run	1			2			3		
Class	Recall	Precision	F1-Score	Recall	Precision	F1-Score	Recall	Precision	F1-Score
Six	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	N/A
Seven	0.00	0.00	N/A	0.00	0.00	N/A	0.96	1.00	0.98
Eight	1.00	0.97	0.98	1.00	0.97	0.98	0.70	0.34	0.46
Nine	0.97	0.99	0.98	0.97	0.99	0.98	0.19	0.22	0.20

Table 6.12: MNIST Combined Confusion Matrix Results

	Six	Seven	Eight	Nine
Six	450	0	0	0
Seven	0	260	1	0
Eight	140	2	671	218
Nine	91	12	73	516
Fail	0	566	0	0
Total	681	840	745	734
Precision	1.00	1.00	0.65	0.75
Recall	0.66	0.31	0.90	0.70
F1-Score	0.80	0.47	0.76	0.72

6.5.1.3 MNIST Analysis

The results from the MNIST dataset tests are encouraging. Overall accuracy against the built dataset is 63%, and against the thousand image validation set is 60%. Both these results are considerably better than the baseline classifiers, notably the conventional component classifiers. The close overall accuracy between these datasets indicate that the evolved classifiers generalise relatively well, without a large degree of overfitting to the training set. However, overall accuracy does not tell the entire story when applied to multiclass classifiers. It tends to cover up issues with the classification of individual classes within the classifier.

Hence, it makes more sense to examine the individual F1 scores and confusion matrices associated with each dataset. Over the runs against the built dataset, performance of each class is reasonable. In nearly every case it is higher than the baseline classifiers. Some classes have a perfect F1-Score, indicating both perfect retrievable of relevant items, as well as perfect classification. However, this score is likely due to the low amount of data in the built validation set. Only one class failed to be correctly classified at all, the 'six' class in the third run. Examining the confusion matrix, it appears the evolved classifiers had the most trouble classifying sevens and nines. These digits are often drawn in variants very different from each other, which could explain the higher rate of failure compared to the other classes. For example, an eight is usually written the same way by most people, yet sevens can have increased variety due to stroke length and strikethroughs.

These class confusions hold broadly true for the thousand image dataset as well, with the ‘seven’ class performing the worst, followed by the ‘nine’ class. The nine class has the most confusion with the eight class, which makes sense; both these classes contain circular structures and holes (notice that the ‘six’ class is also confused with eights and nines). Examining table 6.11 it is obvious that the classifiers evolved in run one and run two have identical performance on the dataset. The high accuracy is indicative that in both cases the GP system realised that the neural networks provided the best method for classification. Interestingly, despite this identical result over the 1000 image MNIST dataset, over the built dataset, results vary. This is likely due to differences in methods evolved to handle the training data itself. Run 3 performed the worst, yet was the only evolved classifier that could classify the ‘seven’ class well, a fact reflected in the built dataset tests. Despite both runs 1 and 2 failing to evolved a ‘seven’ classifier that worked over the thousand image dataset, over the built validation set, both classifiers had reasonable success on the ‘seven’ class, indicating some overfitting [4] towards the training data.

It appears that over the MNIST datasets, the evolved classifiers were capable of using the digit classifying neural network, yet how well each classifier handles the built vs 1000 image dataset reveals variances in how each classifier evolved to adapt to the training data. However, despite some classes performing relatively poorly, in general, results are superior to the baseline, and most were able to use the available network for classification purposes. The MNIST dataset has been tackled by an evolved vision system before, as described by Oeschle et al, who achieved a higher accuracy over the dataset than the neural system achieved [31]. However, the evolved vision system in that case was trained on the entire MNIST dataset, rather than a set of forty digits, and its overall per class accuracy is lower than the per class accuracy of the best performing neural classifiers.

6.5.2 Fashion MNIST

6.5.2.1 Built Validation Set

First, the Fashion MNIST built validation data was used as the test dataset. For each run, the precision, recall and F1-Score for each class are calculated, and shown in table 6.13. These runs were then combined in order to generate a confusion matrix for each class, shown in Table. 6.14.

Table 6.13: Fashion MNIST Results

Run	1			2			3		
Class	Recall	Precision	F1-Score	Recall	Precision	F1-Score	Recall	Precision	F1-Score
Shirt	0.45	0.42	0.43	0.82	0.56	0.67	0.73	0.67	0.70
Trainer	0	0	N/A	0.08	1.00	0.07	0.00	0.00	N/A
Bag	0.85	0.79	0.81	0.69	0.69	0.69	1.00	0.81	0.90
Ankle Boot	1	0.33	0.5	0.75	0.33	0.46	1.00	0.40	0.57

Table 6.14: Combined Confusion Matrix

	Shirt	Trainer	Bag	Ankle Boot
Shirt	22	18	0	0
Trainer	0	1	0	0
Bag	6	3	33	1
Ankle Boot	3	14	3	11
Fail	2	0	3	0
Total	33	36	39	12
Precision	0.55	1.00	0.77	0.35
Recall	0.67	0.03	0.85	0.92
F1-Score	0.60	0.05	0.80	0.51

6.5.2.2 1000 Images from Fashion MNIST Validation Set

Tables 6.15 and 6.16 show the results of the classifiers ran against 1000 relevant images extracted from the 10000 image Fashion MNIST Validation Set.

Table 6.15: Fashion MNIST 1000 Image Results

Run	1			2			3		
Class	Recall	Precision	F1-Score	Recall	Precision	F1-Score	Recall	Precision	F1-Score
Shirt	0.17	0.16	0.16	0.98	0.50	0.66	0.48	0.41	0.44
Trainer	0.00	0.00	N/A	0.00	0.00	N/A	0.00	0.00	N/A
Bag	0.88	0.57	0.69	0.88	0.49	0.63	0.91	0.63	0.74
Ankle Boot	0.51	0.64	0.57	0.09	0.54	0.15	0.52	0.61	0.56

Table 6.16: Combined Confusion Matrix

	Shirt	Trainer	Bag	Ankle Boot
Shirt	350	472	16	24
Trainer	19	0	14	4
Bag	147	16	685	242
Ankle Boot	73	171	17	400
Fail	146	85	24	95
Total	735	744	756	765
Precision	0.41	0.00	0.63	0.61
Recall	0.48	0.00	0.91	0.52
F1-Score	0.44	N/A	0.74	0.56

6.5.2.3 Fashion MNIST Analysis

The GP system did not perform as well over the Fashion MNIST dataset as it did over the standard MNIST dataset. Accuracy over the built validation dataset is 0.56, and over the 1000 validation images, is 0.48. This is superior to the baseline classifiers, including the conventional component classifier. As mentioned above, accuracy is not necessarily the best measure of a multiclass classifier. It is more accurate to state that the majority of the classifiers have an F1 score superior to their equivalent classifier in the baseline models.

It appears that overall performance was degraded by the inability for the GP system to evolve a suitable classifier for the 'trainer' class. This specific class suffered from confusion with the 'ankle boot' and 'shirt' classes. Both the trainer class and the shirt class are highly textured, as they include graphics or coloured marks on the respective item of clothing, which could lead to classification difficulty. Meanwhile, the ankle boot and trainer classes are naturally confused, as they are both similarly structured items of clothing. The author also occasionally had difficulty differentiating the two classes, so a machine experiencing the same difficulty is not necessarily a huge surprise. This class confusion appears in the thousand image dataset as well.

Comparing the results from the two datasets reveals that performance across them both is similar, with all F1 scores for each dataset within 15% of each other, and most within 10%, though better scores were observed over the built validation set than the 1000 image validation set. This indicates that overtraining to images similar to the training images occurred, with the built validation set accuracy 8% higher than over the thousand image dataset. This is not overly surprising, considering the very low amount of training data that was used to train the GP system. It is a considerable research interest to reduce overfitting in machine learning algorithms, beyond the scope of this thesis. Despite this, for only four training images, results are reasonable, and the neural networks clearly provided a benefit over the conventional components.

Accuracy is lower than expected compared to testing the neural network alone, which indicates some evolved adaptions to the training data hindered how well the neural network can classify each class. The GP system performs more poorly on the Fashion MNIST dataset than the standard MNIST dataset. This may be due to the amount of generations the programs were evolved for; Fashion MNIST is a more complex problem than MNIST, and in most cases other techniques perform worse for Fashion MNIST compared to standard MNIST [74], hence a longer evolution time may have resulted in the GP system making better use of the neural network. The overfitting problem may be solved with the use of additional training images, although the amount used was certainly enough for the GP system to determine that a trained neural network component could be used to increase fitness on the assigned problem, and it may be a case of increasing the *variation* of the training data, rather than the amount.

6.6 Decomposed Neural Networks

The effect of decomposed neural network components available to the GP system is evaluated in this section, and the results shown in tables 6.17 and 6.18. Despite the datasets consisting of four extracted classes, neural networks trained on the entire ten class datasets were exposed to the GP system. This results in ten neural networks, each that can classify a given class in the corresponding dataset. The most immediate effect this has on the GP system is that it increases training time due to the increased amount of possible component choices to make, however, it reduces the probability the genetic algorithm will select a good neural network by chance. The decomposed networks did provide effective when used, notably over the MNIST dataset. To generate the decomposed network components, each network was trained to separate one class from all others. For the MNIST dataset, 500 training images were used, and for the more complex Fashion MNIST dataset, 5000 training images were used. This is a fraction of the total training set for both datasets.

Table 6.17: MNIST Decomposed Neural Components

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
Six	1.00	1.00	1.00	0.73	0.00	0.00	N/A	0.48
Seven	0.71	0.45	0.56		0.90	0.99	0.94	
Eight	1.00	1.00	1.00		0.94	1.00	0.97	
Nine	0.58	1.00	0.74		0.00	0.00	N/A	

Table 6.18: Fashion MNIST Decomposed Neural Components

Dataset	Built Validation Set				1000 Image Validation Set			
Class	Recall	Precision	F1	Total Accuracy	Recall	Precision	F1	Total Accuracy
Shirt	0.18	1.00	0.31	0.33	0.03	0.88	0.06	0.23
Trainer	0.00	0.00	N/A		0.00	0.00	N/A	
Bag	0.85	0.65	0.73		0.88	0.89	0.89	
Ankle Boot	0.00	0.00	N/A		0.00	0	N/A	

6.6.1 Analysis

The results from the MNIST datasets and the Fashion MNIST datasets are quite different, when using programs that make use of decomposed neural networks. Each network available to the classifier could only separate one class from all others. In all cases, the GP system did choose to make use of one (or more) of these decomposed networks at least once in the evolved programs. However, results clearly varied.

The MNIST classifier over the built validation set performed well, beating all baseline classifiers by a significant margin. Over the thousand image validation set, results are less promising. While the overall accuracy still improves over the baselines, two classifiers failed to classify any items in the dataset. However, the exact same classifiers performed well over the built validation set! These contrasting results between these two MNIST datasets reveal clear evidence of overfitting to the training set. It is possible some feature exists in the training data that the evolved classifier takes advantage of to classify items in the built validation set that does not exist in the more standard MNIST dataset. However, this is inverted for the 'seven' classifier, which performed much better over the standard dataset compared to the built validation set. This could well be for the same reasons, in that some feature existed for the seven class that hindered classification in the built validation set. Despite these results, the very high F1 scores for the Seven and Eight class in the thousand image validation set reveal good utilisation of the neural networks those classifiers had access to; that degree of accuracy is very similar to the accuracy of the neural network classifiers alone.

The Fashion MNIST classifier performs worse than the MNIST classifier. Each dataset had two classes that failed to classify at all, across both datasets, indicating that the classifier itself failed to evolve correctly. Like the MNIST data, the built validation set performs better than the general thousand image dataset, again, likely due to overfitting. There is a 25% difference in F1 scores between the different 'shirt' class classifiers. However, for the 'bag' class, the GP system has successfully evolved a classifier with a 0.73 and 0.89 F1 score respectively, which indicates that at least for that class, a decomposed network was used successfully. Despite this, overall accuracy remains low, only beating the conventional classifier by 5% over the thousand image validation set. The difficulty the GP system had in evolving classifiers for this problem is likely due to the more complex dataset, combined with the increased difficulty caused by additional components that must be considered during evolution.

This data indicates that a GP system can take advantage of decomposed networks, and combine them into usable classifiers, and there were no instances where a neural network provided no advantage. However, the system is prone to developing 'dead' classifiers. This could potentially be alleviated with longer training times, however, for this project, only one classifier was evolved for each dataset. This is due to the long training times generating a decomposed classifier takes. On the hardware available, each decomposed classifier was trained for over a week. However, the neural networks themselves were able to be trained on significantly less data and for a shorter period of time than the multiclass network.

6.7 Conclusion

From the MNIST and Fashion MNIST multiclass experiments it is fair to say that integration of a neural network increased the performance of the evolved classifier versus baselines. For decomposed networks, individual classifiers outperform the baseline, and some classes had accuracy close to an evolved classifier that used more complex multiclass neural networks. In the next and final chapter, the author presents a conclusion to this research, evaluates the advantages and disadvantages of this method, and examines if the original research questions have been answered.

CHAPTER 7

Conclusion

As humans, our brain has the ability to take a brief look at very few examples of an item, and then, somehow, use those few examples to generalise to hundreds of items. Humans are capable of using information we already know to identify objects in new images, even if perhaps we have never seen that image before.

In this research, the author has attempted to tackle the problem of transferring,in a simplified format, this ability to a machine, such that a vision system can evolve to use knowledge that already exists in order to classify items that it has not seen before. In the same manner that an end-user human trying to build a vision system for multiple classes may attempt it by picking and choosing trained neural networks manually, the author attempted capture this behaviour, by using genetic programming to combine neural networks, such that the evolving vision system can pick and choose relevant neural networks to assist in its classification of data unseen by both the genetic algorithm, and the neural network itself.

It was the author's hypothesis that the genetic programming system would evolve classifiers that use relevant neural networks to boost its own ability to classify items, to a greater degree than using conventional region descriptors such as area, perimeter, or number of edges can accomplish in a generic vision system, with the limited amount of training data used.

Two main research questions were posed in order to determine if neural networks can be used by a genetic algorithm that evolves vision systems successfully. These were:

1. **Can neural networks be integrated into a genetic algorithm that evolves object**

classifiers in a manner that increases the accuracy of the evolved system?

2. **And if so, can we use much simpler networks in the confidence that the genetic algorithm can assemble them for accurate classification of many regions in one image?**

The first question was examined through the integration of multiclass neural networks, with a brief detour into attempting to utilise extracted neural filters for the same purpose. If neural networks can be integrated, then the evolved vision systems gains the benefit of prior domain knowledge, and no longer has to spend time attempting to construct complex classifiers when tackling joined classification problems it can already solve individually. If one network can be integrated, this is indicative that multiple networks can be implemented and selected for. The next question asks whether multiple neural networks can be integrated, and, if so, those networks may be simple compared to a multiclass neural network, hence reducing necessary training time, and training data, for the networks themselves.

In general, the overall theme of of this project was thus: neural networks are known to be very good at image classification. Can this ability be harnessed by a genetic programming system such that it can learn to automatically use this classification ability for its own ends?

7.1 Assessment

Over this year of research, the author has heavily modified an existing vision system evolver, integrating neural components, testing extracted neural filters, adding high level feature descriptors, rewriting fitness functions to support these changes, and speeding up processing via implementation of multiprocessing. The results, presented in the previous chapter, evaluate the final system over two forms of two different datasets. In this section, the developed approach is critically evaluated. The author notes that there appears to be no previous work in attempting to combine genetic algorithms and neural networks in this manner, and that, because of this, there was no guarantee that such a combination would prove workable. However, it is the author's findings that such a combination, while perhaps not competitive compared to more focused machine learning techniques (the author certainly does not expect this current system to be capable of classifying thousands of classes with over 90% accuracy, as some deep neural networks demonstrate), it is certainly possible. And this is the general idea behind this research project; advanced techniques for classification certainly already exist, but combining multiple variants of these techniques to solve different problems still remains a problem that requires human attention. This approach attempts to transfer this problem to a machine.

7.1.1 Multiclass Neural Networks

It is the author's opinion that the first research question has been answered. The genetic algorithm was able to evolve programs that made use of multiclass neural networks in order to classify both MNIST digits and Fashion MNIST Clothing items. The results generated were clearly superior to the results generated by baseline statistical classifiers and, more importantly, the baseline conventional component classifier. However, despite this, certain classes did still sometimes fail to classify any items. This is more obvious when examining the more complex Fashion MNIST dataset results. Additionally, accuracy was lower than may have been accomplished by running the neural network component alone against the dataset, likely due to the classifier over-training to fit the training dataset, and hence, adapting to it in a manner that hindered the neural networks effectiveness. Despite this, evidence is clear that neural networks can be used by a system that uses genetic programming to evolve vision systems.

7.1.2 Decomposed Neural Networks

While for some classes, the system successfully chose the correct decomposed neural networks to use, and hence demonstrates that this is certainly possible, there was a higher rate of failed or confused classifications than experienced with the multiclass neural networks. Thus one can not be completely confident that the system can assemble these decomposed neural networks to obtain close results to a multiclass neural network. However, where such networks were used, accuracy remained reasonable. It is the authors reasoning that here the amount of generations used to evolve the classifier was too low, and that assembling multiple binary neural networks is a more complex problem than the utilisation of multiclass networks. It does appear that the system is capable of assembling these simpler networks, and that, where used, they do provide a boost in accuracy. Hence it appears that the second question can be considered answered as well, though certainly the criteria of ‘confidence’ does not appear to have been reasonably met.

7.1.3 Advantages and Drawbacks

The main advantage of this method is that the training data used to choose the neural networks can be very low. In this research, only four images are used for each dataset, giving a total of forty individual items. For the classifiers that evolved to utilise neural networks correctly for a given class, that class was usually classified well. This is because the genetic algorithm learns to use information it already knows to solve the given problem, requiring only a small amount of ‘guiding data’.

Another advantage is that decomposed binary classification networks can be used successfully for classification purposes, even if the evolved system did suffer from a higher rate of failed classifiers due to the complexity of the dataset combined with insufficient training times. These binary networks were trained using a fraction of the total dataset of the multiclass neural networks, and evolved to a similar accuracy in a much shorter period of time. This is demonstrated several times, where a classifier using a decomposed network had excellent accuracy over its class.

Another potential advantage of this method is that it allows the inclusion of already trained neural networks, which theoretically means that a very large amount of prior domain knowledge can be included. While it is possible to repurpose a convolutional neural network via transfer learning by taking an existing network and retraining it over a new dataset, this only results in a new network trained on that specific problem. With the developed system, providing access to multiple networks each suited to a different problem allows the genetic algorithm to choose which set of prior knowledge it uses to solve the current problem, which may include solutions that require the use of multiple networks. This kind of system offers a more ‘off-the-shelf’ approach to vision problems, at least assuming suitable pre-trained networks exist (one potential disadvantage of this system is that it *does* need such networks to exist. Luckily, new pretrained models appear frequently). This is especially useful as neural models become increasingly difficult to train due to complexity. It is certainly possible to lack the resources to train a neural network while still having the resources required to conduct inference using it.

However, the main disadvantage this method incurs is very long training times for the classifiers, and this is one of the main reasons that only a subset of each dataset was used to evolved every program. Training times for the multiclass classifiers with four classes took one to two days, and for the decomposed classifiers with four classes took over a week. This training was performed on a machine with 24 individual processing cores, but no GPU. Should a GPU be present, the neural network evaluation time would be expected to decrease.

Another drawback of this approach is the potential that some classes may never be classified correctly. Several times, across both datasets, and both multiclass and decomposed networks, a class failed to be classified at all, due to the genetic algorithm never finding a suitable combination of genetic components to create a classifier. Although, most of these failures occurred over the more generic thousand image datasets, with the exact same class performing better over the built validation sets. This is indicative of overfitting, and could potentially be reduced with more initial training data or a longer training time. It should be noted that while it is desired that such developed vision systems function as well as the human visual system, in practice, the human visual system is trained for many years, continuously, with a vast amount of high-quality input data. The authors system is on average, trained for hours, and in total, even including the neural networks, views a very, very small subset of the training data the average human will see in their lifetime.

7.2 Future Research

While this research, over the course of the year, has certainly answered the two main research questions the author set out to investigate, this by no means suggests that *all* questions in this area have been solved. With little other apparent research in this area, many different research paths branch from this starting investigation.

The simplest questions lay around the tuning of the genetic programming algorithm itself. Machine learning algorithms are slaves to the parameters that guide them, and the parameters in this project were chosen as they ‘worked well’. There was simply not time to evaluate the effect of every parameter. A full investigation into the effect of population size, number of generations, and the probability of crossover or mutation could shine a light on which parameters tend to lead to the best evolved classifier in the quickest time.

Beyond this would involve experimenting with the neural components themselves, such as implementing new ones. The author used components trained specifically for this project, but a wide array of pretrained models exist in the wild, often deep networks with thousands of classes. What effect would including one of these have on the ability of the GP system to generalise? What effect would including ten of them have? There is not necessarily a need to include only convolutional networks, either. The author chose to use them as they have been shown to work well with images, and that was the current problem domain. However, if one was attempting to read important information from street signs, perhaps the addition of a recurrent neural network would perform better at text classification.

It could be argued that the author's neural GP system for evolving vision systems is a kind of transfer learning, as the knowledge of each neural network becomes available to the genetic programming algorithm to use and build upon. However, in the world of deep learning, transfer learning involves obtaining a pretrained model of a deep neural network, and retraining the last few layers over a new dataset, in the hope that the knowledge of all previous layers of the network will be used by the last layers to classify new data. This could be integrated into the GP system itself, by creating a component that trains the last layers of a pretrained neural network on the current training set automatically. If enough malleable pretrained networks were included, the best one to use for the task at hand could be selected and included in the final classifier.

7.3 Closing Remarks

It seems that from this one investigation, with its modest goal to answer two questions, many other questions arise, and many modifications and improvements are left to be discovered. The author never intended for this developed system to be complete, only to discover if it was possible to combine neural networks and genetic programming in this manner; which it certainly seems to be. With the apparent absence of research into this specific area, this was somewhat of a stab into the dark, but the results appear indicative that there is yet more to accomplish in order to build a system that is capable of evolving a program that pieces together neural networks in the same manner that human might. There is even more to discover if the end result is a generic system capable of stacking together networks such that the end result might be close to our own organic stacks of networks; our brain.

It is likely that to accomplish this lofty goal, improvements in both neural networks and genetic programming algorithms will be required, not to mention the additional increase in processing power to run such a system.

Machine learning itself is a very broad field with very difficult goals, and sometimes it is easy to lose track of the feats already accomplished. The best deep neural networks can classify MNIST and Fashion MNIST with around 90% accuracy, and it was this ability that this project attempted to convey to a generic genetic programming system. It is often desired that neural networks and other machine learning techniques be ‘human competitive’ over a dataset, which occasionally seems like a tall thing to ask of a technique that has not benefited from millions of years of evolution. Despite this, progress in machine learning appears to be ever accelerating, and the author doubts it will be long before it becomes even more integrated into our lives than it already has.

— FIN —

Appendices

APPENDIX A

Diagrams

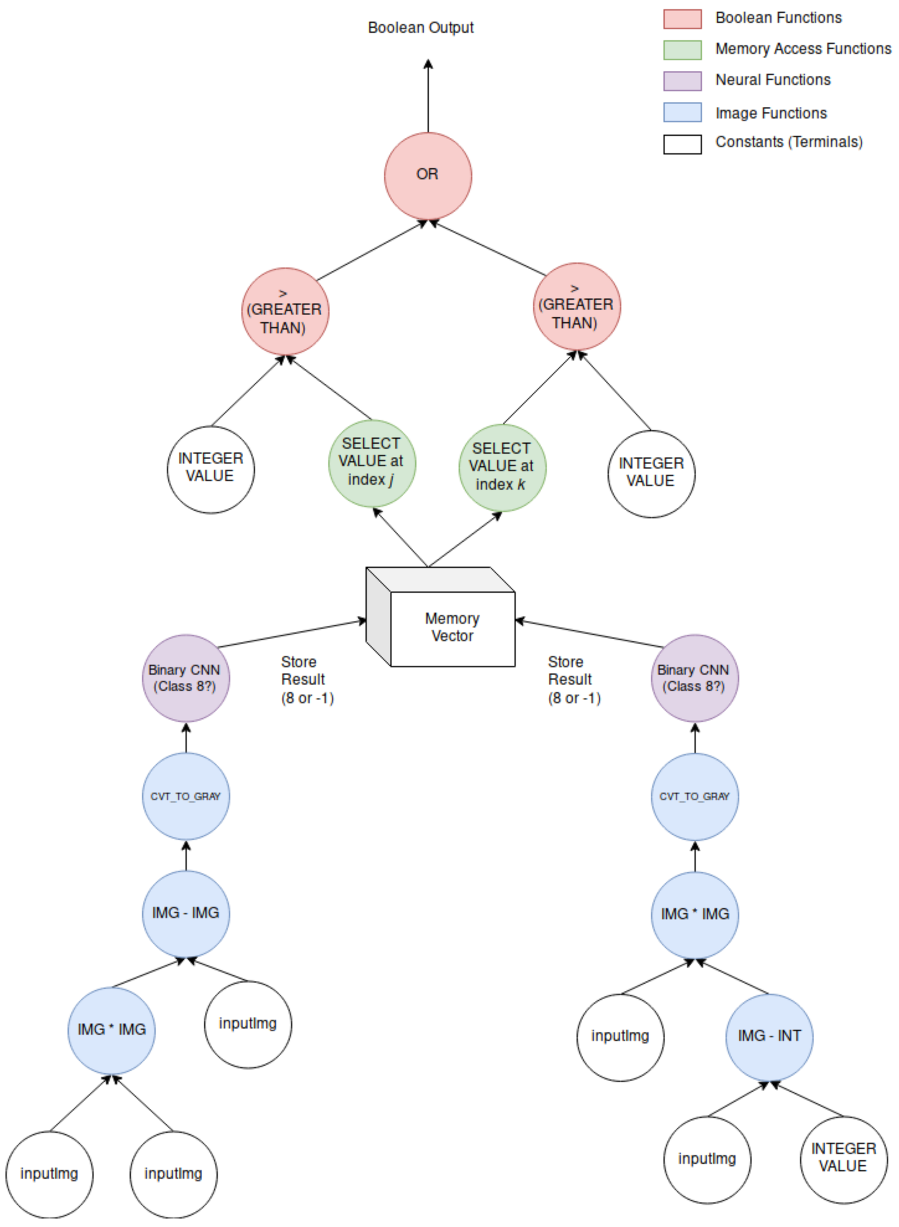


Figure A.1: Complex example of a generated program tree.

APPENDIX B

Functions and Terminals

B.1 Math Functions

Function Name	Input Type	Output Type	Description
I ->F	INT	FLOAT	Casting
F ->I	FLOAT	INT	Casting
I + I	INT	INT	Integer Add
I - I	INT	INT	Integer Subtract
I / I	INT	INT	Integer Divide
I * I	INT	INT	Integer Multiply
F + F	FLOAT	FLOAT	Float Add
F - F	FLOAT	FLOAT	Float Subtract
F / F	FLOAT	FLOAT	Float Divide
F * F	FLOAT	FLOAT	Float Multiply
F ^F	FLOAT	FLOAT	Float Power

B.2 Boolean Functions

Function Name	Input Type	Output Type	Description
I==I	INT	BOOL	Integer Comparison
F==F	FLOAT	BOOL	Float Comparison
I <I	INT	BOOL	Integer Less Than
I >I	INT	BOOL	Integer Greater Than
F <F	FLOAT	BOOL	Float Less Than
F >F	FLOAT	BOOL	Float Greater Than
AND	BOOL	BOOL	Boolean AND
OR	BOOL	BOOL	Boolean OR
XOR	BOOL	BOOL	Boolean XOR
NOT	BOOL	BOOL	Boolean NOT
IF	[BOOL, BOOL]	BOOL	IF Function

B.3 Basic Image Functions

B.3.1 Simple & Numeric

Function Name	Input Type	Output Type	Description
CVT_TO_GRAY	IMG	IMG1	Convert a colour image to grayscale (Single channel).
IMG (+ - / *) INT	[IMG, INT]	IMG	Four functions that take an image and an integer and perform the corresponding operation on the image.
IMG (+ - / *) IMG	[IMG, IMG]	IMG	As above but for two colour images.
IMG1 (+ - / *) IMG1	[IMG1, IMG1]	IMG1	As above but for two single channel images.
avg3to1	IMG	IMG1	Average a three channel image into a one channel image.
max3to1	IMG	IMG1	Return the maximum channel.
min3to1	IMG	IMG1	Return the minimum channel.
range3to1	IMG	IMG1	Return max channel minus the minimum channel.
chan1	IMG	IMG1	Return the first channel.
chan2	IMG	IMG1	Return the second channel.
chan3	IMG	IMG1	Return the third channel.

B.3.2 Thresholding

Function Name	Input Type	Output Type	Description
getOtsu	IMG1	IMG1	Otsu threshold the input.
getThresh3	[IMG, INT]	IMG1	Threshold a colour image by INT input.
getThresh1	[IMG1, INT]	IMG1	Threshold a single channel image by INT input.
invert	IMG1	IMG1	Invert a single channel image.

B.3.3 Morphological

Function Name	Input Type	Output Type	Description
getRectKernel	[INT, INT]	KERNEL	Get a rectangular kernel of INT by INT
getEllipticalKernel	[INT, INT]	KERNEL	Get an elliptical kernel of INT by INT
getCrossKernel	[INT, INT]	KERNEL	Get a cross kernel of INT by INT
erode	[IMG1, KERNEL, INT]	IMG1	Erode an image using a kernel. INT refers to number of iterations.
dilate	[IMG1, KERNEL, INT]	IMG1	Dilate an image
blur	[IMG1, KERNEL, INT]	IMG1	Gaussian blur an image.
medianBlur	[IMG1, INT]	IMG1	Median blur an image.

B.4 Advanced Image Functions

Function Name	Input Type	Output Type	Description
AREA	[IMG1, FEATURE_VEC]	FEATURE_VEC	Calculate the area of a region. Takes a vector, and returns a vector with the result appended to it.
PERIMETER	[IMG1, FEATURE_VEC]	FEATURE_VEC	Get the perimeter of a region.
ASPECT_RATIO	[IMG1, FEATURE_VEC]	FEATURE_VEC	Get the aspect ratio of a bounded region.
ROUNDNESS	[IMG1, FEATURE_VEC]	FEATURE_VEC	Calculate the roundness of a region, where a result of 1 indicates perfect roundness, i.e a circle.
RECTANGULARITY	[IMG1, FEATURE_VEC]	FEATURE_VEC	Fit a bounding rectangle to a region.
ROTATED_RECT	[IMG1, FEATURE_VEC]	FEATURE_VEC	Fit a rotated bounding rectangle.
JOINTS	[IMG1, FEATURE_VEC]	FEATURE_VEC	Counts the corners in the regions shape.
SOLIDITY	[IMG1, FEATURE_VEC]	FEATURE_VEC	Determines how filled in the region is.
HOLES	[IMG1, FEATURE_VEC]	FEATURE_VEC	Counts the holes in the region.
HU_MOMENTS	[IMG1, FEATURE_VEC]	FEATURE_VEC	Calculates the hu moments for the region.
SELECT_FROM_VEC	[FEATURE_VEC, INT]	FLOAT	Select the value at the index INT from a FEATURE_VEC

B.5 Neural Functions

Function Name	Input Type	Output Type	Description
NEURO_MNIST_ALL	[IMG1, FEATURE_VEC]	FEATURE_VEC	A function that interfaces with a neural network that can classify all MNIST classes.
NEURO_MNIST_[0 - 9]	[IMG1, FEATURE_VEC]	FEATURE_VEC	Ten functions, each that use a neural network to classify a single class of MNIST.
NEURO_FASHION_MNIST_ALL	[IMG1, FEATURE_VEC]	FEATURE_VEC	A function that interfaces with a neural network that can classify all FASHION_MNIST classes.
NEURO_FASHION_MNIST_[0 - 9]	[IMG1, FEATURE_VEC]	FEATURE_VEC	Ten functions, each that use a neural network to classify a single class of FASHION_MNIST.

B.6 Terminals

Terminal Name	Type	Value
0i	INT	0
1i	INT	1
5i	INT	5
25i, 50i, ..., 200i	INT	25, 50, ..., 200
2i, 4i, ..., 256i	INT	Powers of 2 from 2 - 256
0f, 1f, ..., 10f	FLOAT	0.0, 1.0, ..., 10.0
16f, 32f, ..., 256f	FLOAT	Powers of 2 from 16.0 - 256.0
25f, 50f, ..., 200f	FLOAT	25.0, 50.0, ..., 200.0
TRUE	BOOL	True
FALSE	BOOL	False
INPUT_IMG	IMG	Placeholder for image input

Bibliography

[1] D. J. Montana and L. Davis, “Training Feedforward Neural Networks Using Genetic Algorithms,” in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI’89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 762–767. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623755.1623876

[2] G. F. Miller, P. M. Todd, and S. U. Hegde, “Designing Neural Networks Using Genetic Algorithms,” in *Proceedings of the Third International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 379–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=93126.94034

[3] P. Yimyam and Adrian F. Clark, “Agricultural Produce Grading by Computer Vision Based on Genetic Programming,” Ph.D. dissertation, University of Essex, 2014.

[4] Cameron P. Kyle-Davidson and Adrian F. Clark, “Neural Network Components for Evolved Vision Systems,” 10th Computer Science and Electronic Engineering Conference, University of Essex, Sep. 2018.

[5] A. M. Turing, “I.âĂŤCOMPUTING MACHINERY AND INTELLIGENCE,” *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950. [Online]. Available: https://academic.oup.com/mind/article/LIX/236/433/986238

[6] D. B. Fogel and L. J. Fogel, “An introduction to evolutionary programming,” in *Artificial Evolution*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sep. 1995, pp. 21–33. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-61108-8_28

[7] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, ser. Complex adaptive systems. Cambridge, Mass: MIT Press, 1992.

[8] E. E. Altshuler and D. S. Linden, “Wire-antenna designs using genetic algorithms,” *IEEE Antennas and Propagation Magazine*, vol. 39, no. 2, pp. 33–43, Apr. 1997.

[9] G. Hornby, A. Globus, D. Linden, and J. Lohn, “Automated Antenna Design with Evolutionary Algorithms,” in *Space 2006*. San Jose, California: American Institute of Aeronautics and Astronautics, Sep. 2006. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2006-7242

[10] M. W. Cohen, M. Aga, and T. Weinberg, “Genetic Algorithm Software System for Analog Circuit Design,” *Procedia CIRP*, vol. 36, pp. 17–22, Jan. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2212827115000360

[11] M. A. Lewis, A. H. Fagg, and G. A. Bekey, “GENETIC ALGORITHMS FOR GAIT SYNTHESIS IN A HEXAPOD ROBOT,” in *World Scientific Series in Robotics and Intelligent Systems*. WORLD SCIENTIFIC, Jan. 1994, vol. 11, pp. 317–331. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/9789814354301_0011

[12] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. [Morrisville, NC: Lulu Press], 2008, oCLC: 837998350.

[13] T. Blickle and L. Thiele, “A Comparison of Selection Schemes used in Genetic Algorithms,” p. 67, 1995.

[14] P. J. B. Hancock, “An empirical comparison of selection methods in evolutionary algorithms,” in *Evolutionary Computing*, ser. Lecture Notes in Computer Science, T. C. Fogarty, Ed. Springer Berlin Heidelberg, 1994, pp. 80–94.

[15] G. Syswerda, “Uniform crossover in genetic algorithms,” in *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 2–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=645512.657265

[16] M. Srinivas and L. Patnaik, “Adaptive probabilities of crossover and mutation in genetic algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 24,

no. 4, pp. 656–667, Apr. 1994. [Online]. Available: http://ieeexplore.ieee.org/document/286385/

[17] N. Hansen, D. V. Arnold, and A. Auger, "Evolution Strategies," in *Springer Handbook of Computational Intelligence*. Springer, Berlin, Heidelberg, 2015, pp. 871–898. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-662-43505-2_44

[18] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *arXiv:1703.03864 [cs, stat]*, Mar. 2017, arXiv: 1703.03864. [Online]. Available: http://arxiv.org/abs/1703.03864

[19] J. McCall, "Genetic algorithms for modelling and optimisation," *Journal of Computational and Applied Mathematics*, vol. 184, no. 1, pp. 205–222, Dec. 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0377042705000774

[20] L. J. Eshelman and J. D. Schaffer, "Real-Coded Genetic Algorithms and Interval-Schemata," in *Foundations of Genetic Algorithms*, ser. Foundations of Genetic Algorithms, L. D. Whitley, Ed. Elsevier, Jan. 1993, vol. 2, pp. 187–202. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780080948324500180

[21] J. R. Koza, "Human-competitive results produced by genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3, pp. 251–284, Sep. 2010. [Online]. Available: https://doi.org/10.1007/s10710-010-9112-3

[22] M. Brameier, "On Linear Genetic Programming," p. 278, 2004.

[23] M. Brameier and W. Banzhaf, "Evolving Teams of Predictors with Linear Genetic Programming," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 381–407, Dec. 2001. [Online]. Available: https://doi.org/10.1023/A:1012978805372

[24] ——, "A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining," *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, vol. 5, no. 1, p. 10, 2001.

[25] J. F. Miller, Ed., *Cartesian Genetic Programming*, ser. Natural Computing Series. Berlin Heidelberg: Springer-Verlag, 2011. [Online]. Available: //www.springer.com/gb/book/9783642173097

[26] D. G. Wilson, S. Cussat-Blanc, H. Luga, and J. F. Miller, "Evolving Simple Programs for Playing Atari Games," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '18. New York, NY, USA: ACM, 2018, pp. 229–236. [Online]. Available: http://doi.acm.org/10.1145/3205455.3205578

[27] S. Luke and L. Spector, "A Comparison of Crossover and Mutation in Genetic Programming," p. 9, 1997.

[28] D. J. Montana, "Strongly Typed Genetic Programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, Jun. 1995. [Online]. Available: http://dx.doi.org/10.1162/evco.1995.3.2.199

[29] T. Haynes, R. L. Wainwright, S. Sen, and D. A. Schoenefeld, "Strongly Typed Genetic Programming in Evolving Cooperation Strategies," in *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 271–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=645514.657915

[30] L. D. Chambers, *The Practical Handbook of Genetic Algorithms : Applications, Second Edition*. Chapman and Hall/CRC, Dec. 2000. [Online]. Available: https://www.taylorfrancis.com/books/9781420035568

[31] Olly Oechsle, "Towards the Automatic Construction of Machine Vision Systems using Genetic Programming," Ph.D. dissertation, University of Essex, School of Computer Science and Electronic Engineering, 2009.

[32] W. A. Tackett, "Genetic Programming for Feature Discovery and Image Discrimination," in *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 303–311. [Online]. Available: http://dl.acm.org/citation.cfm?id=645513.657740

[33] R. Poli, "Genetic Programming for Image Analysis," in *Proceedings of the 1st Annual Conference on Genetic Programming*. Cambridge, MA, USA: MIT Press, 1996, pp. 363–368. [Online]. Available: http://dl.acm.org/citation.cfm?id=1595536.1595587

[34] A. Song and V. Ciesielski, "Texture analysis by genetic programming," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, vol. 2, Jun. 2004, pp. 2092–2099 Vol.2.

[35] T. Loveard and V. Ciesielski, “Representing classification problems in genetic programming,” in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, vol. 2. Seoul, South Korea: IEEE, 2001, pp. 1070–1077. [Online]. Available: http://ieeexplore.ieee.org/document/934310/

[36] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. [Online]. Available: http://www.nature.com/articles/nature16961

[37] N. Watters, A. Tacchetti, T. Weber, R. Pascanu, P. Battaglia, and D. Zoran, “Visual Interaction Networks,” *arXiv:1706.01433 [cs]*, Jun. 2017, arXiv: 1706.01433. [Online]. Available: http://arxiv.org/abs/1706.01433

[38] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “WaveNet: A Generative Model for Raw Audio,” *arXiv:1609.03499 [cs]*, Sep. 2016, arXiv: 1609.03499. [Online]. Available: http://arxiv.org/abs/1609.03499

[39] Warren S. McCulloch and Walter H. Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity,” 1943.

[40] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*, ser. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. [Online]. Available: https://books.google.co.uk/books?id=P_XGPgAACAAJ

[41] M. L. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Mit Press, 1972, google-Books-ID: Ow1OAQAAIAAJ.

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1,” D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362. [Online]. Available: http://dl.acm.org/citation.cfm?id=104279.104293

[43] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0893608089900208

[44] M. Sabourin and A. Mitiche, “Original Contribution: Optical Character Recognition by a Neural Network,” *Neural Netw.*, vol. 5, no. 5, pp. 843–852, Sep. 1992. [Online]. Available: http://dx.doi.org/10.1016/S0893-6080(05)80144-3

[45] M. D. Garris, J. L. Blue, G. T. Candela, G. T. C, P. J. Grother, S. A. Janet, and C. L. Wilson, *NIST Form-Based Handprint Recognition System (Release 2.0)*, 1997.

[46] B. K. Verma, “Handwritten Hindi character recognition using multilayer perceptron and radial basis function neural networks,” in *Proceedings of ICNN’95 - International Conference on Neural Networks*, vol. 4, Nov. 1995, pp. 2111–2115 vol.4.

[47] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[49] C. Baral, O. Fuentes, and V. Kreinovich, “Why Deep Neural Networks: A Possible Theoretical Explanation,” *Departmental Technical Reports (CS)*, Jun. 2015. [Online]. Available: https://digitalcommons.utep.edu/cs_techrep/975

[50] R. Ilin, T. Watson, and R. Kozma, “Abstraction hierarchy in deep learning neural networks,” in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 768–774.

[51] A. Graves, M. Liwicki, S. FernÃąndez, R. Bertolami, H. Bunke, and J. Schmidhuber, “A Novel Connectionist System for Unconstrained Handwriting Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 855–868, May 2009.

[52] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *arXiv:1609.08144 [cs]*, Sep. 2016, arXiv: 1609.08144. [Online]. Available: http://arxiv.org/abs/1609.08144

[53] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[54] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object Recognition with Gradient-Based Learning," in *Shape, Contour and Grouping in Computer Vision*, ser. Lecture Notes in Computer Science, D. A. Forsyth, J. L. Mundy, V. di GesÃž, and R. Cipolla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 319–345. [Online]. Available: https://doi.org/10.1007/3-540-46805-6_19

[55] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385

[56] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, "Gradient-Based Learning Applied to Document Recognition," p. 46, 1998.

[57] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. USA: Omnipress, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425

[58] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *arXiv:1412.6806 [cs]*, Dec. 2014, arXiv: 1412.6806. [Online]. Available: http://arxiv.org/abs/1412.6806

[59] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *J. Mach.*

Learn. Res., vol. 15, no. 1, pp. 1929–1958, Jan. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2627435.2670313

[60] S. Hochreiter, “The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions,” *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, no. 2, pp. 107–116, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1142/S0218488598000094

[61] J. V. Tu, “Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes,” *Journal of Clinical Epidemiology*, vol. 49, no. 11, pp. 1225–1231, Nov. 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0895435696000029

[62] C. Dumitru and V. Maria, “Advantages and Disadvantages of Using Neural Networks for Predictions,” *Ovidius University Annals, Economic Sciences Series*, vol. XIII, no. 1, pp. 444–449, 2013. [Online]. Available: https://ideas.repec.org/a/ovi/oviste/vxiiy2012i1p444-449.html

[63] L. Rios and N. Sahinidis, “Derivative-free optimization: a review of algorithms and comparison of software implementations,” *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013. [Online]. Available: https://ideas.repec.org/a/spr/jglopt/v56y2013i3p1247-1293.html

[64] D. Whitley, T. Starkweather, and C. Bogart, “Genetic algorithms and neural networks: optimizing connections and connectivity,” *Parallel Computing*, vol. 14, no. 3, pp. 347–361, Aug. 1990. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/016781919090086O

[65] K. O. Stanley and R. Miikkulainen, “Efficient Reinforcement Learning Through Evolving Neural Network Topologies,” in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO’02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 569–577. [Online]. Available: http://dl.acm.org/citation.cfm?id=2955491.2955578

[66] R. Oullette, M. Browne, and K. Hirasawa, “Genetic algorithm optimization of a convolutional neural network for autonomous crack detection,” in *Proceedings of the 2004*

Congress on Evolutionary Computation (IEEE Cat. No.04TH8753), vol. 1, Jun. 2004, pp. 516–521 Vol.1.

[67] E. P. Ijjina and K. M. Chalavadi, "Human action recognition using genetic algorithms and convolutional neural networks," *Pattern Recognition*, vol. 59, pp. 199–212, Nov. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320316000169

[68] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," *arXiv:1712.06567 [cs]*, Dec. 2017, arXiv: 1712.06567. [Online]. Available: http://arxiv.org/abs/1712.06567

[69] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Sep. 2014, arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556

[70] P. K. Spivak, "Discovery of Optical Character Recognition Algorithms using Genetic Programming," in *Genetic Algorithms and Genetic Programming at Stanford 2002*. Stanford Bookstore, 2002, pp. 223–232.

[71] W. Smart and M. Zhang, "Using Genetic Programming for Multiclass Classification by Simultaneously Solving Component Binary Classification Problems," in *Genetic Programming*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3447, pp. 227–239. [Online]. Available: http://link.springer.com/10.1007/978-3-540-31989-4_20

[72] A. M. Teredesai and V. Govindaraju, "Issues in evolving GP based classifiers for a pattern recognition task," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, vol. 1, Jun. 2004, pp. 509–515 Vol.1.

[73] Yann Lecun, Corinna Cortes, and Chris Burges, "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges," 1999. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[74] Zalando Research, “A MNIST-like fashion product database.” Sep. 2018, original-date: 2017-08-25T12:05:15Z. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist