Neural Network Guided Transfer Learning for Genetic Programming

Alexander Newton Wild, BSc (Hons), MRes
School of Computing and Communications
Lancaster University

A thesis submitted for the degree of
Doctor of Philosophy

Thursday 17th August, 2023

I dedicate this thesis to Tess. And to myself.

Neural Network Guided Transfer Learning for Genetic Programming
Alexander Newton Wild, BSc (Hons), MRes.
School of Computing and Communications, Lancaster University
A thesis submitted for the degree of *Doctor of Philosophy.* Thursday 17th August, 2023

Abstract

Programming-by-Example, and code synthesis in general, is a field with many different sub-fields, involving many forms of machine learning and computational logic. With advantages and disadvantages to each, attempts to build effective hybrid solutions would seem to be a promising direction. Transfer Learning (TL) provides a good framework for this, as it allows one of the classic code synthesis techniques, Genetic Programming, to be augmented by past success, to target a particular code synthesis system to the problem domain it is facing. TL allows one type of machine learning algorithm, in this thesis a neural network, to support the core GP process, and combine the strengths of both. This thesis explores the concept of hybrid code synthesis approaches, and then brings the identified strongest elements of each approach together into a single neural network driven Transfer Learning system for Genetic Programming. The TL system operates autonomously, without any human intervention required after the problem set (in example only format) is presented to the system. The thesis first studies how to structure a training corpus for a neural network, across two different experiments, exploring how the constraints placed on a corpus can result in superior training. After this, it studies how GP processes can be guided, to ensure that a hypothetical NN guide would be useful if it could be created and how it can best assist the GP. Finally, it combines the previous studies together into the full end-to-end TL system and tests its performance across two separate problem domains.

Publications

The experiments in Chapters 4.1 and 6 have generated a number of papers, which have included large portions of the text and contributed to the direction of the thesis.

General Program Synthesis Using Guided Corpus Generation and Automatic Refactoring **Wild Alexander & Porter Barry** In Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31 – September 1, 2019, Proceedings
https://doi.org/10.1007/978-3-030-27455-9_7

Code Synthesis in Self-improving Software Systems **Roberto Rodrigues Filho, Alexander Wild and Barry Porter** In International Workshop of Self-Improving System Integration, IEEE, 4-5, 8/08/2019, Conference contribution/Paper
https://doi.org/10.1109/FAS-W.2019.00015

Neurally guided transfer learning for genetic programming **Wild Alexander & Porter Barry** In GECCO '21: Proceedings of the Genetic and Evolutionary Computation Conference Companion July 2021 Pages 267–268
https://doi.org/10.1145/3449726.3459511

Multi-Donor Neural Transfer Learning for Genetic Programming. **Wild, Alexander & Porter, Barry** In: ACM Transactions on Evolutionary Learning and Optimization, 23.08.2022.

Acknowledgements

This work was made possible by the Levenhulme Trust's Funding, via their support of the "The emergent self-aware data centre: autonomous software landscaping at scale" project, RPG-2017-166

Special thanks to Dr Barry Porter, for guiding me through the whole thesis, and the many researchers with whom I've discussed the science, the work and the emotions of going through a PhD. Notably Leandro Soriano Marcolino, Roberto Rodrigues Filho, Wyatt B Lindquist, Christina Bremer, Dr Emily Rowan Winter, William Peter Dance, Tereza Kameníková, Jeneen Haj-Hammou and Thomas Hewitt.

Thanks to my friends and family, most notably Nicky, Sue and Ian Wild, Ellie, Wendy, Evan, Viv and Dave Haines, and Tasneem Solie.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

This thesis targets the problem of 'Programming by Example' Gulwani and Jain (2017) (PbE). Specifically, this entails viewing the desired input and output of a program, and attempting to generate source code which produces that behaviour. For example, if a hypothetical user were to want a program to reverse a list, they could provide a set of examples in which a list is supplied as the input and its reversed form as the output. With a limited number of examples, the PbE system would be required to return a program which could reverse a list, by deducing the user's intent and implementing it.

More broadly, the system could have applications for all manner of creative tasks. It could be possible for a game designer to implement mechanics by simply giving a few examples to an automated code synthesis system, then validating the code (possibly simply by reviewing some example of behaviour, to avoid the designer needing any programming experience). Alternatively, it could design entirely novel re-implementations of existing programs, by being given examples of that program's behaviour. Such alternative implementations could have useful properties, such as potentially being more efficient under certain constraints or avoiding certain bugs. This approach would be an alternative approach to Genetic Improvement, which would not re-use the original system's code.

PbE is broader than these particular examples, and does not inherently depend on a user giving the system a precise programmatic task. It simply requires a set of examples of behaviour, and attempts to build a program to exhibit that behaviour. The development of approaches towards PbE may also be applicable in the field of Inverse Reinforcement Learning, where an agent attempts to build a model of another agent's behaviour; or it could form the core of approaches which build models of more abstract systems such as economic behaviours. Both of these fields would benefit from approaches which build code models, as code is symbolic and highly abstract, giving a clear idea of the behaviours of the system as opposed to its physical appearances.

This work, however, assumes the existence of a human 'user' of the system, as this is a typical deployment scenario of a PbE system, but attempts to maintain generality of the system such that it is not constrained to operate only alongside a human user.

This hypothetical scenario assumes a human without programming experience has a set of data processing tasks they wish to perform, perhaps in a spreadsheet type application, as the PbE system FlashFill and BlinkFill were designed for Microsoft Excel (initially Excel 2013) Singh (2016). They have a set of inputs they wish to transform into outputs, and manually transform the first 10 inputs into the desired output form. The system would then infer the desired behaviour and produce a function to replicate it, to process all remaining inputs. This would allow a non-programmer to rapidly process large quantities of data using a machine tool.

This particular problem specification implies conceptually simple functions, which may translate into shorter functions in terms of lines of code, if the language provides a compact syntax and potentially includes useful libraries. Despite years of study, the programs able to be synthesised by current PbE systems in both industrial applications and the literature remain short. While a physicist programmer attempting to replicate a physical system's behaviour may require hundreds of lines of code, PbE currently operates in spaces of programs of tens of lines at most.

This suggests that one possible direction for the field is that by resolving problems of tractably short length, but allowing high-level abstractions and functional encapsulation, it may allow progress along the path towards full human-level programming-by-example. This thesis aims to add to the literature in a way which advances the field along these lines.

1.2 Key Issues and Challenges

A number of factors make this problem a difficult one. The level of problems for which any synthesis technique can succeed rarely matches that able to be solved by a beginner programmer.

- **Problem space size.** In almost all works which will be touched on by the Literature section of this work, extreme constraints on the languages used are imposed. Restrictions such as very low number of variables permitted, low line counts, or limited operators present in the language reduce the number of programs which can be considered by the synthesis systems. This work similarly limits itself, roughly matching the sizes possible for the majority of the genetic programming literature, but exceeding certain exhaustive neural network searchers such as Balog et al. (2017) and Zohar and Wolf (2018a).

Despite the restrictions, the number of programs remains large. This particular work deals in program space sizes of up to $5 * 10^{119}$. As will be discussed, search spaces of this size render many techniques intractable.

- **Poor navigability of program space as classically represented**. A conventional programming language is formed of a sequence of tokens, and even a single changed token can alter the behaviour of the program radically. Altering an addition operator to a subtraction operator may retain a degree of high-level similarity between the outputs of two programs, but altering a loop to a conditional operator can cause the program's output to change completely. As such, programs which are adjacent to one another in program space do not necessarily share functional properties in a way which can be usefully exploited.

  This renders many search techniques far more difficult. Genetic methods rely on following changes in the fitness landscape, which requires to some degree an average slope to exist and to be sufficiently strong relative to this 'noise' to drive the population in the correct direction. Neural methods, such as Williams (1992), depend on a neural network to be able to embed a fitness landscape, which depends on it being sufficiently predictable for the network to learn, as well as low-information enough to fit within the network's capacity.

- **Patterns in the input-output which are useful to finding a solution can be symbolic and high-level**. When observing a program's behaviour in input-output form, a human is able to pick out certain very high level characteristics. Simpler ones might be 'all output values are even', or 'all outputs are prime', but may become as complex as requiring knowledge of the differences between UK and US date format. A human may find it easy to recognise these characteristics, but for a machine learning algorithm to recognise these characteristic it could require large degrees of training, and some may be nearly impossible, as they lack the human perspective which the data-manipulation problem may require.

1.3 Research Questions

This thesis attempts to study and answer a number of dimensions relating to programming by example, focusing on Genetic Programming supported by Neural Networks.

- What is the best way to generate artificial training data for a neural network for this particular field?
- Which specific form of hints, suggestions or assistance could a GP process be provided with to improve its performance?

- What performance gains could be achieved by an end-to-end system, which automatically trains NNs and uses them to guide a GP without need for human guidance?

1.4 Objectives

The objective is to provide an advance in the field of code synthesis, improving the performance compared to existing approaches. The aim is that by finding a way to integrate the advances from multiple sub-fields a successful hybrid could be produced. It will attempt to combine genetic programming with neural network algorithms, using the network to recognise high-level features of the problem and guide the GP appropriately.

1.5 Thesis Contributions

- A review of the literature of the sub-fields of program synthesis which focuses on their commonalities and potential overlaps
- An evaluation of how neural network training can be best structured to maximise its ability to predict programmatic structures. This lead to an understanding of which types of features need to be present in a training corpus for optimal results. This then lead to to the introduction of a novel system able to guide creation of a corpus which exceeds a uniformly generated one
- An evaluation of how Genetic Programming performance changes as the program space is constrained. This lead to an understanding into which parts of the solution, if given to a Genetic Programming algorithm, would best guide its search and by how much they could increase performance.
- An end-to-end Transfer Learning system which combines neural networks and Genetic Programming to exceed the baseline implementation of both.

1.6 Thesis Structure

The remainder of this thesis is structured as follows: Chapter 2 provides an overview of the field of program synthesis, and a motivation for the direction taken by this research. It covers three broad topics, logical based solvers, neural code synthesis and genetic programming, along with others falling outside of these three sub-fields. It also touches on how Transfer Learning has been approached in the field of Genetic

Programming. It provides an overview of the rough common architectures these approaches share, and also a set of milestone and key papers in the literature.

Chapter 3 discusses and explains the proposed architecture. Based on the areas identified as promising in the background section, a transfer learning system was developed to improve the performance of a Genetic Programming (GP) algorithm. It employs a set of neural networks to deploy code fragments which have been extracted from prior successes to boost the performance of this GP algorithm. This is intended to leverage the high-level behaviour recognition capabilities of neural networks without losing the high performance of GP, which has shown good success on general purpose Turing Complete languages.

Chapters 4, 5 and 6 then covers the experiments which demonstrate the sub-systems involved, and answer the research questions posed in this introductory chapter. The experiments follow a logical progression from the start of the algorithmic process to the end. The first aspect investigated regards the data to provide to a neural network to optimise its training and effectiveness of similar programs to the ones it has been trained on, based on a set of human-useful programs defined by the user. This investigation takes place over two experiments. The first demonstrating how constraints on how the training corpus for a neural network benefit its eventual performance on human-useful programs in a general Turing-Complete programming language. The second then investigates how using the output of GP processes to form corpora can boost the performance of neural networks. Once the mechanism by which the neural network should be trained has been established, mechanisms by which the GP can be guided are investigated, to identify candidate targets for neural network guidance. This is done in both a systematic and human-guided way, across two separate experiments, to determine which forms of assistance best improve performance. After that, an experiment is run to evaluate if a neural network is able to deploy this assistance correctly, and evaluating its performance while doing so. After this is done, an end-to-end system is tested, extracting fragments from solved programs to deploy into newly faced problems. Finally, after the single-guidance-point architecture has been demonstrated to be functional, a full end-to-end Transfer Learning system for Genetic Programming is deployed and evaluated, able to draw from arbitrary numbers of previously solved programs, and demonstrates a high degree of improvement off the same Genetic Programming algorithm without Transfer Learning assistance.

Chapter 7 then ties the work together by evaluating how the research questions were answered, and discussing the work as a whole, and future directions which could be taken by later works which build on this.

Chapter 2

Background Literature

2.1 Introduction to Background

This section attempts to cover the broad field of programming-by-example (Gulwani and Jain (2017)). This is a field of computer science (not always necessarily Machine Learning), which attempts to generate source code to match the behaviour of a function described by a set of observed input-output mappings. This goal has been described as 'The Holy Grail of Computer Science' (Gulwani et al. (2017)) and has been studied since 1950 (Turing (1950)) yet remains elusive. This long history has, however, provided a wide range of approaches, each with their own advantages and failings. This work considers them to roughly fall into the groupings of Genetic Algorithms, the oldest yet currently most successful in terms of complexity of generated programs; the work leveraging Deep Neural Networks which have seen wide-ranging success in the fields of Machine Learning and Artificial Intelligence in recent years; and the somewhat restricted yet sometimes extremely effective deductive reasoning systems. This work then attempts to bring these together into a single direction which is felt to be promising and understudied.

2.2 Genetic Programming

Genetic programming (GP), as well as being the oldest field in code synthesis, is also, as will be discussed below, one of the most successful in terms of general performance. While it may be exceeded in certain domains by other techniques, it shows good success across a wide range of problem categories. This section discusses GP firstly in terms of its current state of the art, then in terms of architectures of interest to this thesis, then ends with a section of works which closely resemble GP but do not technically fulfil the criteria to be classified as a GP process. These are included into due to the similarities between GP and them, in terms of how program space is navigated on a high level.

2.2.1 Overview of Genetic Algorithms

Genetic Algorithms are one of the oldest concepts in computer science, first proposed by Alan Turing in 1950 in his paper “Computing Machinery and Intelligence", which discusses the ‘Imitation Game’, the famous Turing-test for intelligence, and proposes a form of genetic programming to create an agent able to pass this test. This genetic process was human-guided, using a human designer to select between candidate programs, and was rather optimistic (Turing discusses whether a machine would be able to integrate socially into a school setting), but an interesting first look into the concept of evolving machine instruction sets towards a desired goal.

GP has an inherent advantage, compared to neural approaches, in that it can evolve both the discrete tokens (the operators, variable references, etc) and literal values such as floating-point constants or hard-coded character strings. While a neurally powered architecture could in principle have a symbolic output, for operator selection, and a continuous one, for literal value assignments, this has yet to see an implementation in the literature.

Genetic Algorithms, of which Genetic Programming is a subclass, is what is termed a ‘derivative free search’ technique (Rios and Sahinidis (2009)). These are techniques which search through a space (in this case program space), to optimise a given function (in this case to reduce the distance between the sampled program’s outputs and the example program’s outputs, termed the error or loss). These functions do not compute a derivative of this space, which would allow a gradient at any point to be computed. If this gradient were followed, by moving an infinitesimally small distance within the space, the new point is guaranteed to be more optimal (lower error) than the previous one. The reason GP processes do not compute this derivative to determine this gradient is simply that source code cannot be differentiated with respected to an input-output example. Discrete spaces (such as source code, with its symbolic tokens) do not allow continuous navigation with infinitesimally small steps. Further to this, the error of points within program space cannot be analytically determined without source-code execution as this would violate the Halting Problem (Turing (1937)).

Instead, genetic algorithms operate by sampling a set of points, a ‘population’, evaluating these based on a metric known as a fitness function, and stochastically generating a new population of points based on the differences in fitness of the population. The algorithm is designed such that the new population members (sampled points within the search space) have a higher probability to be closer to the higher-fitness members of the previous population than to the lower-fitness ones. In this way, the population as a whole migrates over time. It has been demonstrated that keeping the best-ever-found point (the elite) has very consistently high effectiveness (Dinabandhubhandari et al. (2012); Chakraborty and Chaudhuri (2003)).

The weakness is that this relies on the fitness function leading the population to a desired outcome. Since the population will travel based on the differences in fitness it samples, towards higher fitness, there must firstly exist such a difference, and the difference must be useful. Flat areas, with no perceived difference in fitness will cause the algorithm to either stagnate or wander blindly. Valleys with lower fitness which come between the population and a desired solution can slow and delay search in the direction of this solution, despite the eventually good outcome which would ensue. As can be expected of a field with decades of study, this core concern has been studied in depth (Renzullo et al. (2018); Folino et al. (2000)).

One approach to improving navigation around the space is to deliberately avoid getting stuck by dynamically changing the landscape. Novelty Search (Lehman and Stanley (2010); Doncieux et al. (2019)) is a range of approaches which aim to drive the GP towards exploring regions of program space which have not been yet visited. In its simplest form it is a fitness function which does not have a given aim, only a repulsive effect away from already-visited areas of the landscape. This allows, in time, more varied forms of programs to be generated. When combined with a standard GP fitness function, which attempts to achieve a set goal, it can supplement this by breaking it out of local minima and boosting exploration and population diversity.

2.2.2 State of the Art

The field of GP shows good current progress, with work on both regression tasks, in which programs are assembled to approximate or replicate a target mathematical function (Iba et al. (2018); Augusto and Barbosa (2000); Uy et al. (2011); Zhong et al. (2018)), or on full Turing-Complete function synthesis operating on common programming data-types and data-structures Helmuth and Spector (2015). This thesis' particular area of interest falls under Turing-Complete programming-by-example. GP work is able to handle a variety of data-structures and data-types, and combine these seamlessly into an overall framework, allowing great flexibility and diversity of generated programs.

A very good example of this is Pantridge and Spector (2020), which represents how flexible state-of-the-art GP is with types, able to seamlessly blend together functions which take and return a very wide range of data types. It demonstrates how types which would be considered objects, in an object-orientated language (such as lists and time-date representations) and primitives can be handled by a GP process.

In terms of effectiveness within a narrower, more studied GP domain, Helmuth et al. (2018)'s work evaluates its performance on the General Program Synthesis Benchmark Suite, and show high performance on this set of problems which cover numeric and string-based programs. We see 3 problems of the 29 remain unsolved (no solutions ever produced), two which require numeric processing followed by conversion into a string output and one which is purely string-based. These problems cover a range of primitive data-types, specifically floating point values, integers, Boolean values and character strings. In terms of complexity, the most complex of these problems may require approximately 20 lines of code for a human programmer to solve (depending of course on the programmer's implementation choices), with in the case of the never-found problem 'grade' (which requires assigning letter grades based on numerical test scores) 5 flow-control operators. Its primary contribution is in its improved mutation style, which allows the programs to evolve by addition and deletion independently. Rather than a new gene replacing an old one in-situ, the newly added one and the removed one can be from different parts of the program. This appears to be highly successful, and highly applicable to any future genetic programming work.

G3P (Forstenlechner et al. (2017)) is major work in the field of Genetic Programming, studied on the General Benchmark Suite, including further published analysis to assess its performance in greater depth Forstenlechner et al. (2018). While powerful, G3P, even with the improvements to its data-type handling abilities (including bugfixes) brought by the subsequent work, was unable to solve 11 of the 29 problems in the test suite, 8 fewer than Helmuth et al's. G3P is a grammatical style of program synthesis Ryan et al. (1998), which is able to transform a linear string of characters into an Abstract Syntax Tree able to be executed as a source-code function.

In terms of direct application, the work of Smith and Heywood (2019) evolves a game-playing agent, able to tackle the problem of controlling a character in the video game 'Defence of the Ancients 2 (DOTA2)', a problem also tackled by high-profile research in the field of Deep Neural Networks (Berner et al. (2019)).

2.2.3 Transfer Learning for Genetic Programming

Transfer Learning has been studied for GP before, with ongoing work showing good progress (O'Neill et al. (2017); Muller et al. (2019); Chen et al. (2020)). A core theme is the concept of transferring parts of existing generated programs into the population of programs being generated by the GP process which is attempting to solve the currently presented problem. This is done by extracting sub-trees from the Abstract Syntax Trees of population members of the previous GP run's last generation. It has primarily been investigated in the context of regression, in which a function is being evolved to approximate a scalar-returning mathematical function. The use of floating point values allows greater granularity in fitness, and the design does not seek a 'correct' implementation, but merely a high-fidelity approximation. This allows the last generation to have subsamples taken based on ranked fitnesses, in a way which a more granular fitness landscape might not allow. From these, commonly occurring sub-trees can be extracted, and used as primitives in future GP processes. They can be placed as atomic functions, with sub-trees which would previously require multiple mutations being added as single blocks of code.

Another approach which has been demonstrated to improve GP performance is re-use of previous solutions. Wick et al. (2021) demonstrate how switching tasks during a GP process, or re-using one problem's last generation as the current task's first generation, can improve GP performance. This approach can allow very large amounts of genetic material to be transferred, as the entire source code of the solution is re-used.

This is approached in an alternate fashion by Kelly and Heywood (2015), further developed in later work (Kelly and Heywood (2018)), which approaches the problem by allowing later programs to call earlier ones as sub-functions. Their approach differs to a degree from previously discussed Transfer Learning in that it treats previously generated programs as immutable functions, no longer subject to genetic processes. Their work focuses on a Reinforcement Learning domain, where the genetic algorithm is attempting to evolve a policy to transform sensory inputs into motor outputs in a game-playing robot or game character. Their approach allows later functions to delegate the action selection task to already-learnt policies, as opposed to selecting an output in a low-level fashion. This allows complex behaviours to be abstracted into simple elements, reducing the complexity of the programs generated, and demonstrating good results. Its inability to alter these programs themselves represents a major difference, however, and the authors themselves do not use the term "Transfer Learning" to describe their process, but rather "Knowledge Transfer". This inability to alter the re-used functions prevents certain interesting transfer learning capabilities, such as retaining code flow structures (loops and conditionals) while altering the blocks of code within the program.

This re-use of generated code is shared with the work of Keijzer et al. (2004), which similarly allows previously generated code to be re-used, this time as a library of functions which can be deployed by the newly generated programs. Again, this precludes the ability to mutate these libraries, which has the advantage of reducing the size of the program space which the GP must navigate, but reduces its ability to transfer learn in some domains, as code cannot be inserted between transferred lines. Similar work includes Jaskowski et al. (2007), which employs a GP to solve visual character recognition tasks, and allows transferral of geometric splines to allow shared properties of the glyphs to be exploited for faster learning.

Helmuth et al. (2020) similarly studies the use of code fragments from previous problems, and specifically studies the re-use of constants, which are critical to certain problems featured in The General Program Synthesis Benchmark Suite (Helmuth and Spector (2015)). That work clearly demonstrates that the selection of donor programs is critical to TL's success, suggesting that targetting specific sources rather than simply drawing from all pre-existing solved programs is valuable. This targetting could be done, but it would be optimal for autonomy's sake if a machine learner could perform the task on its own.

That work is supported by Muñoz et al. (2020), which examines how donor-recipient pairings of programs may benefit from TL across different problems, and studies the properties of these donations, which gives some insight into the forms of TL program donors which may be beneficial.

Transfer Learning also has interesting similarities to the Plastic Surgery Hypothesis (Barr et al. (2014a)). The Plastic Hypothesis demonstrates the effectiveness of transferring code elements from parts of the same program, in order to improve performance of a genetic algorithm. This shares the concept of employing existing code as a source of complex genetic material to use in mutations, allowing larger and more effective jumps through program space between generations. The tasks differ slightly, especially in that the plastic surgery hypothesis transfers human-written code, but it is promising evidence that the overall approach is broadly applicable.

Somewhat related to TL is augmenting GP search with additional input, which has been explored by Hemberg et al. (2019), using domain-specific knowledge and natural language guidance. Hemberg's work analyses the program specification, which in this case is provided in terms of both the IO examples and an accompanying natural language description, and specialises the GP process towards a particular problem – for example by augmenting the literals available or by biasing the mutations. This expands somewhat from programming-by-example, which usually relies on only the given examples, but could serve as a useful additional source of data for a GP algorithm in some cases.

2.2.4 Neural Network boosted Genetic Programming

Work exists on combining the capabilities of neural networks and genetic algorithms. Neural approaches have certain desirable properties which they can bring to GP. A core one is in automatic creation of high-level abstractions in representations of program behaviour. Neural networks can recognise properties within the IO examples which would require human designer knowledge to embed into the fitness function. If the GP is simply trying to reduce the euclidean distance between its generated programs' outputs and the target outputs specified by the IO examples, it may miss, for example, that all the values in the target output are in ascending order. This could inform the system that programs it is generating which also have outputs in ascending order are on the path to success, and valid intermediary steps between the empty program and a program which replicates the desired behaviour.

Even higher level behaviours could be spotted, such as extremely different outputs in terms of numerical values which are highly similar in terms of high-level behaviour.

For example a program which sorts a list into ascending order is only one operator away from a program which sorts programs into descending order. If the GP's fitness function computes Euclidean distance between sampled program's output and target output, or computes number of elements in their correct position, this near-perfect program would be assigned a near-minimum fitness, and discarded. A neural network, however, could learn to recognise this similarity and keep the program as a desirable candidate.

This high-level abstraction is possible for a human designer to implement into the fitness function, but to do so requires the designer to have foresight of which elements may be required, and how to balance these factors against other factors involved in navigating program space. Without machine learning it may be very difficult for a human designer to decide what weight to assign to each high-level feature compared to simpler ones such as distance between target output and sampled program output. A neural network could potentially learn this based on sampling program space, thus producing a fitness function tailored to the statistical properties of the program domain the GP process is tasked with operating on.

Neural networks can also be deployed in other areas of the genetic process, such as mutation guidance, or in implementing a dynamic component to the fitness function which learns in an online fashion as the GP samples the space.

Work exists in this domain. Bunel et al. (2017) employ neural networks to guide a GP process, but focuses on the domain of program optimisation. This is to say they start with an already completed program which possesses the desired IO behaviour, and wish to optimise its internal behaviour by seeking new programs which match its behaviour. As such it is somewhat outside of our domain of programming-by-example.

Nigam et al. (2020) employ neural networks to guide the process in a way which avoids stagnation. This is similar to Novelty Search (Lehman and Stanley (2010); Doncieux et al. (2019)), and attempts to avoid the issue of GP populations stagnating in a local maximum in the fitness landscape.

Work continues on this novel program domain, with unpublished work available in preprint (Mandal et al. (2019)) indicating that this direction is being actively pursued and may lead to interesting developments in the literature in the near future. This work brings a new direction to the task, by combining the domain of Transfer Learning with neural network guidance.

2.2.5 Other Gradient-Free Search Techniques for Source Synthesis

Certain other examples exist of search strategies which are not fully genetic, but do not use gradient-navigation methods employed by neural networks. One such example is FrAngel (Shi et al. (2019)), which is a search strategy for source code similar to genetic programming, with certain alterations. In terms of similarities, it employs a fitness function similar to multi-objective genetic algorithms (Konak et al. (2006); Erfani and Utyuzhnikov (2011)), which seek to maintain diversity in the population by retaining individuals which are specialised in different aspects of the problem being faced. In this case, the algorithm retains the best individual at each example within the programming-by-example IO input set. The second change is that it allows arbitrary crossover, as opposed to two-parent crossover. Any collections of parents may be selected as donors for genetic material. This may have an affect on its performance, but does not change the underlying functionality of the algorithm to a major extent, as all genetic transfer could occur in a genetic algorithm over multiple generations. Its core novelty is its lenient fitness function, which allows flow control operators to be evaluated as if their best-case were their current, and receive ‘partial credit’. In this fashion, if there exists a way in which a conditional at a given point could branch the code to improve the programs accuracy, any conditional receives a degree of credit for being placed at that position. Subsequent mutations can then alter how the conditional processes the variables to cause it to behave as the hypothetical one does, and thus achieved the known-possible but as-of-yet unrealised behaviour. Verma et al. (2018) uses an interesting technique of generating a training landscape using a neural network, then defining the examples the source code must replicate based on the neural network’s behaviour. Their domain is reinforcement learning, so their examples for programming-by-example are behaviours to replicate, and they generate these behaviours first by training a neural network to perform them, then by using its behaviour as the example of "what to do" given to the source code synthesiser. Liskowski et al. (2020) adapt the classic GP strategy by training a neural network to represent code as a fixed-length floating point vector, a latent representation which the network can decode into source code. This changes the nature of mutations, as they can take the form of movement within this latent space of arbitrary scale, with every point in this space able to be converted into symbolic source code. The ideal is for this continuous domain to improve mutation and thus search, and initial empirical results indicate this direction has promise. Currently, however, the mapping between latent space and source code is generated simply to maximise the expressivity of the latent space, not necessarily to reduce distance between functionally similar problems or to reduce local minima in the resultant fitness landscape.

2.2.6 Genetic Improvement

Genetic Improvement is related to, and may fall into, the field of Genetic Programming. It employs evolutionary algorithms to search program space, but starts with an existing program, and attempts to improve a given metric. Often, this improvement is the reduction or removal of bugs, as identified by unit tests, but it can also be employed to reduce computational costs or other useful characteristics of a program. The genetic approach to code modification is one of the most popular approaches to automated software improvement Petke et al. (2017) with recent progress in automated bug-fixing Forrest et al. (2009); Arcuri and Yao (2008); Sidiroglou-Douskos et al. (2015), optimising non-functional properties Mrazek et al. (2015); Langdon and Harman (2015); Landsborough et al. (2015), and automatic test-case generation Arcuri and Yao (2008).

Genetic improvement has been applied to source code Langdon and Harman (2015); Cody-kenny et al. (2015), abstract syntax trees Forrest et al. (2009); Haraldsson and Woodward (2014); Barr et al. (2014b), intermediate compiled code forms such as Java bytecode Orlov (2017), and compiled machine code Schulte et al. (2013). It is unclear whether any particular level of operation across this spectrum has quantitative benefits (in terms of the resulting program) over any other level, though in qualitative terms the modification of source code may have benefits in human legibility (and verification) of genetic modifications.

One of the most compelling results in genetic improvement is that of Forrest et al. (2009), which demonstrates effective automated repair of bugs. The approach uses a negative test case which activates the bug, together with positive test cases which verify correct behaviour. During the improvement process, mutation operations are preferentially applied to the execution path that is activated by the negative test case, which reduces the search space to a size tractable for genetic improvement to navigate in reasonable time. The specific genetic operators used are delete, swap, and insert; delete and swap only take place within the negative execution path, while insert may take a statement from anywhere in the program and insert it into the negative execution path.

This sub-field is not entirely related to this thesis, but worth noting, as it is touched on for some parts of this work.

2.3 Neural Code Synthesis

2.3.1 Deep Coder

DeepCoder (Balog et al. (2017)) represented a new direction for code synthesis, in which a deep neural network acted as the primary driver for source code synthesis. Its mechanism of operation is to take a collection of input-output examples and return a collection of operators it expects to be present in the program which generated this sequence. This allows ranking of all programs within program space, starting with those which have the highest-estimated presence operators, and ending with those which require the operators the NN had the least confidence in. In the first implementation, the language defines only 34 operators, and has a fixed problem length sufficiently low (highest tested was 5 lines) to allow for exhaustive searching over every possible program implementation. As such, its returned probability for each operator can be used to rank every program in the space of programs possible within its Domain Specific Language (DSL) and evaluate each in turn.

If the limitations on the language grow beyond the scope of possible exhaustive enumeration, we can consider generalised form of DeepCoder which takes the IO example inputs and returns a ranked subset of programs within program space (As opposed to ranking ALL programs within program space). This subset would be searched over, but the algorithm would return a failure signal if no program matching the desired behaviour was found therein. This concession allows DeepCoder to operate on arbitrarily high program space sizes. The mechanism by which it selects this subspace and provides the ranking can also be abstracted away as implementation details. The original DeepCoder is an implementation of this generalised architecture, just one in which the subset returned is the size of the program space.

2.3.2 Work Building on DeepCoder

A wide range of neural code synthesis techniques exist, with many techniques attempted after the DeepCoder approach, covering a very broad range of problem domains and architectures. Even with our singular focus on programming-by-example, a very broad range of approaches are employed by the community. This section groups some key works based on mutual similarity of architecture, then discusses two interesting directions in the field. These two directions are towards using partially-completed programs' current behaviour as guidance as to how to write the remaining lines and how to select programs for training the networks to make best use of the finite capacity of the network and finite training time available to the developer.

In terms of architecture, Devlin et al. (2017a) shares a strong degree of similarity with DeepCoder, in that it takes IO examples, processes them neurally and returns a searchable volume of program space (using beam search). Parisotto et al. (2017) also employ a similar overall structure, although use a complex method of representing programs using a Recurrent Neural Network architecture (Rumelhart et al. (1986)) to allow the network to model programs as Abstract Syntax Trees. These methods show good results on string-manipulation programs, although lack flow control. They can be seen to match to the DeepCoder's general model of mapping IO example inputs to searchable volumes of program space.

The same can be said for Shin et al. (2018) which uses a beam search, with the volume of program space to search being provided by an LSTM Hochreiter and Schmidhuber (1997) RNN. Bunel et al. (2018) improves upon the search process by using logical analysis of programs within the space returned by the NN to automatically discard syntactically invalid ones, reducing the cost per program. Sun et al. (2018a) demonstrates an architecture which returns a single program, equivalent to a search space of one, which places high demands on the neural network to accurately infer programs. Polosukhin and Skidanov (2018) shares a similarity to Parisotto et al.'s work in that it defines programs in a tree-like structure, to process them as Abstract Syntax Trees, and also operates as a neural network returning a search volume. It differs from other works in this section as it is not a pure programming-by-example algorithm, but also uses natural language as an input form.

2.3.2.1 Intermediary Values as Additional Data

Zohar and Wolf (2018a) shares a core similarity with DeepCoder, in that it operates within the same Domain Specific Language (DSL), and so is directly comparable, but differs strongly in that it is able to leverage partial completion of programs. For each token it places, it is able to execute the partially-written source code, and generate the intermediary inputs, the values the variables would be at if the program ceased termination at this point. In this way, it can potentially reduce the task difficulty, as its information is supplemented. It also reduces the complexity of the information generated by this process using a learnt garbage collection, which is able to mark certain variables as 'no longer necessary' and discard them from consideration. Further work, similar to this, exists in the form of (Ellis et al. (2019)), which also shares the same goal of using partial programs as guidance for the neural network. These particular frameworks, along with DeepCoder, lack flow-control operators, such as conditionals and loops. These are harder to integrate with the concept of partial values, as there is not a singular 'next step'. The lines of code produced by the neural network are not necessarily executed in the order they are generated by the network in a language with flow-control, and so the partially completed values are not necessarily truly representative. It is possible this partial execution and intermediate-variable-as-inputs system could allow some protection against the vast size of the program space. Rather than require the system to infer the full system, it can operate piecewise, inferring the first half of the problem, then using this code's output as if it were inputs to the second half of the program and inferring the remaining half of the code.

To succeed this requires programs to be generally amenable to this strategy. If the program is entirely contained within a loop, there does not exist an easy way to provide the values at the 'halfway' point, as in terms of execution, the first line of the code may be the operation executed halfway through the whole program's execution. It also requires that the first half of the program should be inferable from the program's output despite relative ignorance of how the second half might function. It is not necessarily true that this is the case for the majority of longer programs. Work exists Chen et al. (2019) to address this problem in the neural community, and the problem of conditional paths seems to be well addressed. This later work allowed the concept of intermediary variable states to be expanded to a full Turing-Complete framework, and empirically showed the worth of the idea.

2.3.2.2 Data Sets for Training

Another problem is in selection of inputs for the use in the input-output examples. Poor choices of input examples might fail to adequately demonstrate the desired behaviour. For example if the inputs are already sorted, a sorting algorithm would appear identical to an identity function. If the inputs were numeric and the desired function featured a division operator, the inputs should contain a 0 to demonstrate how this special-case needs to be handled.

If the neural network is to tackle full Turing-Complete spaces it must have an adequate training regime. While DeepCoder uses a uniform selection strategy, simply randomly sampling the program and input space, work on Turing-Complete neural synthesis has demonstrated the worth of guiding the process to maximise training coverage. Shin et al Shin et al. (2019b) demonstrates that selecting input examples to increase coverage of the range of behaviours a program can adopt can improve the effectiveness of neural synthesis techniques. This work specifically targets a full Turing-Complete language, and so also serves as a good example of synthesis which does not run too great a risk of being only applicable to a restricted domain. Despite this work's contribution, the issue of generating data sets to provide as training to a neural network remains an issue. If a training set has a billion examples (10^9) that only represents $1/(10^{31})$ of the program space for our scope. Even with Shin et al's work on maximising the informativeness of these examples, it is a demanding task for the neural network to infer from such a small subset of the space to adequately represent it.

Shin et al. (2019a) demonstrate a work in which neural networks are used to learn from distributions of programs in a library of already-solved problems. This is similar to Transfer Learning, which has been studied in the context of program generation in the Genetic Programming literature. In particular, this work extracts commonly used code structures using a neural network and redeploys them as primitives within the language for use by the searcher. This represents an approach which uses data taken from third-party sources (not the IO specification or the algorithm itself) as training data to accelerate problem resolution. Use of human-generated source code as training data for neural networks is an interesting direction, if sufficient source code relevant to the problem at hand can be collected. This strategy is perhaps most prominently featured in the GPT-3 algorithm discussed below.

Policy Gradient Descent Policy Gradient Descent is a navigation technique which attempts to navigate a policy space (that is to say a Reinforcement Learning (RL) agent's behaviour) by following an estimated gradient. This gradient is created by training a neural network to serve as a function estimate, which takes a given policy and returns an expected reward for that policy. The network can then also provide a gradient of fitness, estimating how a given policy could be changed to improve its expected reward. The network is trained online, updating as the agent takes action, to try to form an accurate fitness landscape. A key implementation of this is the REINFORCE algorithm by Williams (1992).

This has been applied to code synthesis by Bunel et al. (2018), who examined its effectiveness on the Karel dataset. In this implementation, the policy of the RL agent is instead taken to be the program being output. The RL agent in this case can be seen as a code-producing agent, whose actions are each token in the language, which iteratively selects tokens until an END-PROGRAM token-action is chosen.

As opposed to approaches which take a given 'target program' which they are attempting to match, in a supervised learning framework, this approach can find functionally equivalent programs which differ in their implementations. This is a considerable advantage, as there is not inherently a 'correct' way to write a program. Any program which meets the user's specifications (matches the desired behaviour) is satisfactory.

This approach, however, still relies on the ability of a neural network to accurately estimate the fitness landscape. Fitness landscapes in program space can be highly unpredictable, with extreme changes in behaviour from single token changes, such as the introduction or removal of a 'while' loop, or changing which variable is written to. There is a possibility this approach may find the network less effective when compared to tasks with smoother RL fitness landscapes, such as navigating a geometric/geographic space, which may be more forgiving to partially correct policies. This said, their approach when compared to previous neural code synthesis baselines shows good performance, and may be a good direction for neural code synthesis to take in the future.

2.3.3 GPT-3

GPT-3 Brown et al. (2020) is a neural network architecture released by OpenAI. It is a Transformer architecture Vaswani et al. (2017), which is a type of neural network based on neural attentionBahdanau et al. (2015). Neural Attention allows for a network to select which information to transfer to which parts of its internal structure. For example, it can have a neural structure which handles the verb in the sentence, and can use neural attention to select between words in a sentence it is trying to translate to extract the verb. The entire process of selecting which pieces of information to attend to, how the information will be processed once it is attended to, and how to encode it for better use by the attention heads and subsequent processing is all handled by the gradient descent training. This structure has shown itself very powerful in machine translation, and broadly in language processing Belinkov and Glass (2019).

2.4 Neural Algorithmic Imitation

Rather than program directly to source code, either as human-readable text or a lower level but still symbolic and discrete form, algorithms can be represented by a particular set of weightings in a neural network. The network would then 'imitate' the algorithm it has been provided with in its training set. This network would then execute the algorithm in question, as it itself is the function.

Two core advantages exist as design goals within this sub-field, ability to be integrated into an end-to-end differentiable system and learnability.

The first advantage is that this algorithm can now readily be included into a stacked super-algorithm which can be trained by popular stochastic gradient descent mechanisms. An example might be a sorting algorithm which takes its inputs from a convolutional neural network LeCun et al. (1989) (CNN) which reads raw sensory images of text. If trained as a joint function, a singular large neural network, the CNN could learn to recognised hand-written values, and the neurally implemented algorithm could then sort them and return them. This would be a strong advantage if made possible, as it would allow high-level algorithms to be employed without sacrificing any other capability delivered by modern Deep Learning networks. An example of a neural algorithmic processor of this nature is the HOUDINI framework Valkov et al. (2018) which allows for both processing of hand-written digits and photographs of street signs. This advantage could well allow the best of both worlds, maintaining the achievements of the field of Deep Neural Networks while also gaining the symbolic reasoning and generalisation capabilities of algorithmic synthesis.

The second advantage, learnability, hopes to circumvent the difficulties faced by more classical code generation techniques, such as genetic programming or deductive solvers, by changing the representation of the algorithm. Genetic Programming works by stochastically sampling program space, and employing stochastic techniques to navigate towards desired solutions based on the landscape's properties as sampled by the current genetic population. The neural network training technique known as gradient descentBottou et al. (2018) is able to produce an exact direction of travel for each data-point within the training sample, guaranteeing that if the weights of the network are changed in a given way the network's ability to replicate the desired function (also termed decreasing the loss, or decreasing the error) will improve. As long as the neural network does not over-specialise to the training error, termed overfitting, which would cause it to fail to generalise to unseen cases, this training regime may out-perform the genetic algorithm.

Two main disadvantages exist. Firstly, the black-box nature of neural networks, in which it is very difficult for a human to understand their underlying internal behaviour renders the algorithm inherently hard to inspect. Barring particular use-cases of intentionally difficult-to-understand algorithm implementations, such as code obfuscation, human-readability in code is considered desirable. Text-based source code is designed to be comprehensible to programmers other than the author, and to a reasonable degree is so. Algorithms embedded as neural weightings are unlikely to be so.

The second disadvantage is the entire network itself must be computed to execute the algorithm, or (in most cases), a single step of the algorithm. Since current computers are designed based on classical instructions sets, it would appear probable that it is nearly guaranteed to be slower to execute the floating point operations required to determine the output of a neural network than to perform the machine instructions produced by compiled conventional source code. It is outside the scope of this work to determine how far this inefficiency is obligated by the mathematical properties of computation, but holds true for all papers discussed in this section.

2.4.1 Neural Turing Machine & Differentiable Neural Computer

The Neural Turing Machine (NTM)Graves et al. (2014) and its extension, the Differentiable Neural Computer (DNC) Graves et al. (2016) are roughly modelled on the Von Neumann architecture (the broad design of modern general-purpose computers), with both possessing a central controller which performs operations on data, which then has access to a memory structure into which it can read and write information. The DNC builds upon the NTM with additional information provided to the controller, in the form of meta-data embedded into the memory store, but retains the same rough high-level functionality and broad design goal. This goal, as stated by the authors, is to mimic the Von Neumann architecture to enable effective training of algorithms within a framework which is Turing-Complete. As it is Turing-Complete, an NTM weighting exists which would allow it to perform any classically computable mathematical function.

The initial papers show promising and strong results within the field of neural algorithmic imitation. The DNC is able to learn to repeat back a sequence of images, indicating an ability to handle data symbolically (as it had trained on different images than those shown in the testing sequences), an ability to learn to sort these images (without requiring any additional training to give it help in assigning a numeric value to each image) and the ability to find a shortest-path between observed nodes in a graph. The NTM was able to generalise to sequence lengths of twice those it was trained on (6 in training, 12 in testing) with negligible error, and to lengths above that with progressive degradation in signal quality.

Both architectures operate by reading and writing to a memory structure, using Neural Attention Bahdanau et al. (2015). This memory structure takes the form of a matrix of floating point values, each row being a data element the neural attention can attend to, which then consists of a set of values. The NTM and DNC can read a symbol in from their input arrays, process it using an arbitrary number of neural layers, then determine how it should be stored. It can then be read for use at a later time by reading it using a second neural attention mechanism.

This writing process can jointly encode data and meta-data. For example it could write symbolically, simply transferring the inputs without transforming them to the first half of a row within the memory structure, and annotating them with a timestamp in the second half of the row. When reading out values it can learn to seek across this second portion of the data only, agnostic to the contents of the first half of the memory rows. Once it has selected the row to read it can access the full data stored therein and process it, say by returning it directly into its output arrays. As such, the actual data itself remains unchanged, and the network was agnostic to its values throughout the entire process, allowing it to operate the same regardless of the precise values it had been presented with, a core requirement for being considered a symbolic algorithm.

The NTM and DNC are example of what are termed Recurrent Neural Networks. Recurrent Neural Networks (RNN) are networks which retain an internal state between data-point input presentations (such as viewing an image, receiving a word from a sentence, receiving data from an agent's sensors...)Rumelhart et al. (1986). Even RNN architectures far simpler to the NTM have been demonstrated to be Turing-complete Siegelmann and Sontag (1995), and as such could assume a set of weights which could perform any classical computation. To compute this function, the RNN would need to be embedded inside an unbounded loop, and have a 'terminate loop' signal to use.

From a mathematical sense, therefore, the NTM and DNC do not expand upon the capabilities of the RNN, as the RNN is Turing-Complete already. What they do is allow new behaviours internally to the NN. Specifically the aforementioned ability to handle data in a far more symbolic fashion than a simpler RNN such as the LSTM Hochreiter and Schmidhuber (1997) can do, as it can read and write to its memory structure in a fashion which encodes additional data alongside the inputs, and read memory cells based on this data specifically.

Differentiating the NN's underlying mathematical function allows the training process to reduce the difference between its outputs and the outputs defined by a training set of input-output exemplars, but makes no guarantees for points which are not present within this training set. Successfully producing an NN which matches the desired behaviour on the training data points but fails to produce generality and thus does not match the desired behaviour on new data points is known as overfitting.

It is an ongoing area of research in the field of neural networks, and to the author's knowledge at this time remains unresolved.

The NTM paper demonstrates that this architecture is able to out-perform an RNN based system on the algorithms it is presented with, in that it overfits to a lower degree, but does not achieve generality.

We see from the paper that the architectures are able to achieve a degree of generality on certain algorithms of note (i.e. sorting, repeating a signal with delay...) which more commonly seen RNN architectures are not. We also see, however, that this generality only extends so far. While a sorting algorithm written in C may sort lists of 2, 4 or 1,000,000 elements, these networks' performances drop significantly as they encounter problem sizes orders of magnitude beyond those they were trained on.

It is possible, especially on the sorting and sequence-repetition tasks, that the issue is simply noise introduced into the network by its initial configuration, which the training regime failed to remove. It could be argued by proponents of the NTM that a superior form of gradient descent could be developed which would allow it to remove this noise (in the same way a weight regulariser Ng (2004) adjusts weightings in desirable ways, so too could this hypothetical system reduce this noise).

If so, it is entirely possible for the NTM and its successor to be highly successful tools to perform algorithmic imitation, and to have embedded within their weights a general sorting algorithm (or copying algorithm, signal repeater...). It would be a far greater claim, however, to claim that they are able to therefore learn ANY Turing-Complete function.

Their architecture may well be uniquely tailored to processing data in this fashion. Rather than being an example taken arbitrarily from the set of all possible algorithms, sorting may well be an algorithm which their architecture is suited for and naturally learns, especially if the need for a 'function terminate' signal is omitted, replaced with designed-knowledge of the number of time-steps required.

The ability to embed a Turing-Complete function is interesting, and their empirical results on some classes of algorithm make them well suited for certain applications. The ability to embed such a function, however, in no way proves ability to learn it, and no empirical evidence exists that the author is aware of that they are broadly capable algorithmic learners, nor that a training regime exists to allow perfect generality.

2.4.2 Neural Program Interpreters

Neural Program Interpreters Reed and Freitas (2016) (NPI) are a class of architectures which use a neural network to select discrete programmatic steps to perform on the data. Rather than require the data to flow through the network's activations, it is stored in a data-structure external to the network itself, which the network can act on. The network learns to select actions to perform an algorithm, sequentially calling sub-functions to perform transformations on the data it has received, until a termination condition is reached.

These algorithms do not currently fall under the umbrella of programming-by-example as the training regime consists of demonstrations of the desired algorithm in action, rather than requiring it to be synthesised during training or deduced by a GP process. They are briefly included in this discussion due to the potential for them to gain the ability to learn via programming-by-example in the future.

The first argument for these approaches is generality. These approaches are empirically shown to have a strong degree of generality. By training on shorter data sets, they are able to perform the actions required to solve the larger ones. For example by training on a sorting algorithm with sequence lengths of 20, an NPI has been shown to generalise to lengths of 55 without degradation, while an LSTM baseline showed degraded performance on even lengths of 25.

It also plausibly opens the path to broader ranges of applications. The interpreter itself uses only floating point values, as it is a neural system, but can operate on anything its designer-provided actions can operate on. For example it can perform precise integer operations, without risk of floating-point imprecisions. It could also go further and operate on data types which aren't possible for neural networks at all, such as natively handling character string operations or data-structures far larger than could be contained within a network of this size, such as image or sound files.

The programs themselves are essentially calls to sub-functions or atomic operators, as can be performed by executing standard source code. The programs are in the form of the weightings of a controller/interpreter network, however, not in the form of an artificial language program written by a human programmer. In principle this could allow new forms of learning, as the weightings are in a non-discrete form. Any program can be represented as a vector, and a continuous range of programs exist between any two points in program space. As of yet, to the author's knowledge, however, there is no major improvement in the discovery of novel algorithms by NPIs made possible by this continuous program nature, however. The NPI training regime requires program traces, precise demonstrations of which operators to call at which times. They can generalise a program's operator sequences to longer than the sequences they were trained on, but require that these sequences be demonstrated in the first place. As such, they do not synthesise algorithms from IO examples, as is the focus of this work. They may potentially lead to developments on this front, but for now are not directly applicable to the task of source-code synthesis from examples.

2.5 Logic-Based Deductive Solvers

In certain problem domains, certain properties of the solutions to a given IO specification can be deduced. For example, in a string manipulation Domain Specific

Language (DSL), if the string returned by the examples is longer than the string received as input, at least one operation which is able to produce longer strings must exist within a program producing the desired behaviour. If capital letters appear where none were present in the input, a function which capitalises letters must be employed. Following logical rules to assess the IO example and using this to remove portions of program space can allow the remaining space to be small enough for exhaustive searching. This form of code synthesis can produce extremely powerful results at extreme speeds Srivastava et al. (2010a,b), and has industrial applications which can be employed by non-expert users.

The best example of industrial application is FlashFill Gulwani (2011), which was included into Microsoft Excel, and as such available to millions of users worldwide. It is able to synthesise string-manipulation functions, for example recognising that the user wishes the initials of a set of first names followed by last names. The user would enter the full names in one column, the initials in the second, and the synthesis would produce the desired function from only three or four examples, in under a second, on limited hardware. It is succeeded by BlinkFill Singh (2016), which improves its search time by a factor of 41 and reduces the number of required examples (it requires a mere 1.27 examples to correctly recognise problems within its testing set, orders of magnitude fewer than deep neural approaches employ).

String operations are a common problem studied by solvers, with the programming-by-example track of benchmarking competition 'Syntax Guided Synthesis'Alur et al. (2016) (SyGus) being divided between 108 string manipulation and 450 bit vector manipulation tasks.

The solver-based approach depends on this deductive process being able to reduce the problem space down to a tractable size for an exhaustive searcher to operate across. This causes it to fail in cases where presence/absence of a given operator is ambiguous, or cannot be ruled out, most problematically in the case of a language which features loops. As demonstrated in So and Oh (2018), the deductive solver approach can be used to generate programs which feature loops, but must do so in a constrained fashion. While their returned programs did feature loops, the solver was presented only with the task of completing the body of the two nested loops, and did so by treating it as a linear program which took the loops' iterators as arguments. This renders the process ineffective for Turing-Complete languages, but it remains an exceedingly powerful tool in the domains it can function on, as demonstrated by So and Oh's sub-one-second search times as opposed to, for example, DeepCoder's hour long search.

Solver based solutions also face problems tackling larger program sizes. The work of Alur et al. (2017) seeks to address this issue, increasing the number of programming-by-example problems from the SyGus competitions by a factor of 18, by producing solutions to individual examples from within the IO mappings. These individual solutions can then be combined into a unified single program which resolves all cases.

2.6 Direction Suggested by Literature

This chapter has discussed four major approaches, Genetic Programming, Deep Neural code synthesis, Neural Algorithm Imitation and Deductive Solvers. Genetic Programming currently has the advantage in complexity of programs synthesised, but lacks many of the capabilities which the other two possess, indicating that the state of the art could still be improved substantially.

Of the major categories of approach discussed, it would seem that the primary area which is missing from the literature is in the combination of Neural Networks and Genetic Programming. Both approaches have distinct advantages. Neural networks are able to provide higher-level abstraction than a GP process can readily achieve without considerable human designer foresight as to which features would need to be present in the GP's fitness function. GP, on the other hand, has the ability to navigate the extreme ($> 10^{100}$) sizes of program space which appear to be necessary for complex Turing-Complete programs to be synthesised. Fortunately, the literature appears to provide a readily evidence direction for this hybrid algorithm to take. As seen in the discussion relating to neural approaches, many follow the DeepCoder model of presenting the input-output examples to the network which then produces guidance for a searcher. A promising direction would appear to be to replace the commonly-used exhaustive searcher with a genetic programming search process.

This requires analysis as to which forms of advice work well with a GP, how the search can be guided in a way which improves this. This improvement would consist of increasing the probability of a valid solution being found within a given number of generations or using a given amount of computational time, leading to either previously unsolvable problems to be solved or for existing problems to be solved with higher reliability or lower computational cost.

Careful selection of the fitness function used is important to success in genetic programming. The landscape must guide the population's migration from its starting point towards the solution (or a solution, if there are multiple).

Two questions are raised. The first is the very large question of how two programs' input-output mappings can be used to determine how related their underlying source code is, the second is the question of whether this is even possible. We ignore for now the Halting Problem Turing (1937) as the author is unaware of any empirical work on its connection to the problem of code synthesis. The proof merely guarantees the existence of undecidable problems, it does not indicate the proportion of program space which they occupy, and as such it does not in any way preclude extremely effective solutions from being generated with a high degree of reliability. It has been empirically demonstrated within the context of genetic programming, specifically the sub-field of genetic improvement, that partial solutions exist which lead to discovery of desired behaviours often do not alter the behaviour of the problem in and of themselves Renzullo et al. (2018). Intermediate variable state analysis, as discussed in subsection 2.3, could be a remedy for this issue, as long as the mutation produces changes in the variable states, but this has never to the author's knowledge been deployed on a genetic system so no empirical data exists.

Based on work from the neural synthesis community, intermediate variable state analysis could be a remedy for this issue, as long as the mutation produces changes in the variable states. Programs generated by these algorithms are sequences of instructions, many of which permute the variables the program has access to. The last step returns a set of data, the output, which can be compared against the desired output for the given input, but the intermediary values adopted by the variables could also be exploited. Work exists Zohar and Wolf (2018a) which uses the states of partially executed code to guide synthesis. In principle this could be used to form a richer fitness function than can produced by simply comparing function outputs. The author is unaware of any published work in this direction in the GP community, and this approach may well fail due to the need to create a mapping which maps from a space of extreme size (not only all of program space but also all possible inputs into all possible programs) to a single scalar value. While potentially of extreme value, the research question into neural use of intermediary values is therefore outside the scope of this work.

Alongside this is another direction explored in the literature which may work very well with our goals of improving the performance of code synthesis algorithms. Transfer Learning appears to the author to be a very natural choice of direction to pursue. No human learns to program commercially relevant programs in their first programming lesson. They build upon knowledge gained solving easier problems to solve harder ones. In this age of cloud-based services, it seems unreasonable to discard useful past work. If many users face many problems, an aggregated pool of solutions could be readily obtained which could improve the user experience of many future users. This work therefore will strongly focus on the concept of learning from past experience to guide future work, transferring learning between tasks as far as possible.

Transfer Learning has shown good success with the approaches studied, but, to the author's knowledge, has never been shown to operate without designer guidance as to which programs to use as donors of these trees. This would appear to be a critical next step, as by automating the process of donor selection, a full end-to-end system can be created to greatly augment the power of modern GP processes. Machine Learning algorithms have shown ability to deal with source code, and this work studies how neural networks' high level featurisation of IO examples can be used to guide Transfer Learning in an effective fashion, using all previously solved problems as potential donors for code fragments for Transfer Learning.

Large Language Models, such as GPT-3 represent a neural code synthesis approach this work does not use. GPT-3 has shown an ability to synthesise code from natural language descriptors. This does not strictly fall under our target domain of programming-by-example, but is sufficiently interesting to be worth discussing and considering its performance were it presented with programming-by-example tasks.

Programming from natural language specifications is a subtly different task from programming-by-example. The one first is primarily a translation task, translating a natural language into an artificial one (the source code). Specific words in the specification map to specific features in the source code, for example “it should have a red button" maps the word ‘button’ to a call to instantiate a UI element, and ‘red’ then to a method attached to the UI element which changes its colour. This is different from programming-by-example, in which the features in aggregate define the program, and there is no guarantee of a decomposition of the input features into individual elements which match to individual programmatic features or lines of source code. For example when attempting to determine if a given algorithm is a sorting algorithm, all elements in the input and output examples must be compared, to ensure they share symbols exactly and are sorted in all cases. Any individual token in the IO specification given to the system is insufficient by itself, all must be considered.

GPT-3 operates very well as a translator between natural language descriptors and source code and it does so by leveraging its ability to re-arrange and then re-map strings of tokens, by using neural attention to extract the necessary information at each step of the program generation process. It is helped in this by the similarities between the two languages, with natural language descriptors often resembling a form of source code in and of themselves. The instructions the user gives are imperative statements as to which UI elements to add, and the source code is an imperative sequence given to the underlying javascript libraries. As such, discrete clauses in the specification sentences readily translate to discrete code blocks.

If it must aggregate all tokens together into a singular representation, its neural attention does not grant it any advantage, as it cannot subdivide the data usefully, so must attend to all points equally. Without this, it becomes a classic neural synthesis architecture, which has taken a singular feature vector and is attempting to map it to a point or searchable volume of program space.

In conclusion, the literature suggests that building on Transfer Learning is a promising direction which has yet to be explored sufficiently. If combined with Neural Network capabilities, it could allow Genetic Programming Algorithms to achieve strong performance increases in Turing complete programming languages.

Chapter 3

Overview

This chapter introduces in a high-level fashion the proposed concept for how Genetic Programming and Neural Network based Machine Learning can be combined to produce a code synthesis system. This system would exhibit life-long learning, constantly improving itself by sampling code from successes in individual problems and learning how to deploy these code fragments to boost future success.

The Genetic Programming algorithm is boosted by a diversity measure, to boost performance by reducing the impact of local minima, and provide the core synthesis capabilities of the system, and forms the entirety of the system's functionality in initial conditions before any learning has taken place.

The Deep Learning Neural Network component provides an ability to improve the system over time, as it tackles problems given to it by the user. Using only Input-Output examples, it can learn to improve itself by code re-use. The networks can recognise high-level features of the IO examples which suggest that given code fragments would be viable for newly encountered problems, and thus guide the GP process in ways its general-purpose fitness function cannot.

3.1 Overview

The central component of the architecture is a Genetic Programming Algorithm. This work employs a custom language, described in Section 5.1, and is a linear GP process. This form of GP takes the form of direct modification of sequence-of-operators source code. This contrasts with Abstract Syntax Tree representations, which represent programs within the GP's populations as tree structures, or grammar-based GP processes, which map an integer sequence to a source-code form, and perform mutations on this integer sequence.

This linear code format has a high probability of including introns, non-contributing lines of code, into the population members. Tree-based approaches, which define a tree leading back to a single function exit point, can have non-contributing elements but these take different forms as they cannot have 'disconnected' nodes which are not part of the main tree. Conversely linear programming allows lines to both read from and write to variables which are not written to or read by any line contributing to the final outcome. Far from being useless, these lines serve as potential feedstock for future mutations Sotto and Rothlauf (2019), and this work exploits their flexibility for code-insertion purposes.

A simple fitness function is employed, attempting to serve as a broadly applicable one. For task-specific implementations a human designer may wish to replace it with a tailored one to improve performance, but this work seeks to explore the idea in a general sense. The fitness function is however augmented by a novelty search component, which turns it into a dynamic evaluation, relying both on program functionality and GP search process history.

This GP process is supplied with problem definitions in the form of Input-Output Examples by a user, in this case the testing and evaluation framework. The GP algorithm then seeks to produce programs which maximise the fitness function, until a solution is reached or the maximum number of generations is reached. Accepted solutions are defined as those which produce programs matching the provided Input-Output example mapping, that is to say if provided with each input example produce the same output as the exemplar. This is domain specific to our chosen tasks, as symbolic regression GP Augusto and Barbosa (2000); Uy et al. (2011) may elect to have a 'success' condition in which a small degree of error (mismatch between desired output and synthesised program output) is tolerated.

If a program is accepted as valid (that is to say if its outputs match those in the input-output pairs given as problem specification), a subsequent number of Genetic Improvement steps is performed. The fitness function includes an extremely small penalty for each non-intron line. This penalty is orders of magnitude smaller than a single mismatched value in the output, and so serves only to differentiate between programs with identical outputs. Since the accepted program is one with identical outputs to the desired one, it has zero error with regards to IO mapping fitness, and the GP process now acts purely on this line-penalty fitness. For a fixed number of generations, the GP process continues, generating programs which minimise the number of non-intron lines. This serves to ablate the program, removing unnecessary code elements. The process returns the program which scores best (lowest number of lines) after this improvement cycle terminates. The program is post processed to remove intron lines, based on a static code analysis function (the same which was used to determine which lines were introns for the purpose of fitness penalty).

Once an accepted solution is produced, it can serve as a candidate to learn from. Code fragments are extracted, attempting to keep a degree of semantic consistency. This consistency is produced by selecting fragments whose variable read operations do not depend on any earlier operations not included in the fragment. If a line's operation included in the fragment takes a set of variables, these variables must either have never been written to by another operation or the lines writing to them must be also included in the fragment. This includes write operations by loops, and treats both branches of a conditional as being executed.

Code fragments are ordered based on a novelty heuristic, to find those most novel, compared to existing fragments, and a subset selected. These fragments are used to create two synthetic training corpora. The first corpus is a set of programs which contain the given code fragment. The second corpus is a set of programs which do not contain this fragment. The evaluation for fragment presence is generalised, and abstracts the variables' naming. If the fragment had two lines, one writing to variable_5 and the second reading from variable_5, the requirement would simply be that the first writes to the same variable the second one reads from. The logical links between variables are preserved, their precise implementation details, the positions they would occupy in the interpreter's memory, are abstracted away.

Corpus creation occurs by taking all accepted program solutions which match the condition (either fragment presence or fragment absence) and genetically mutating them to produce new programs. Newly generated programs also become targets for potential mutation, allowing the corpus to include programs highly dissimilar to the 'seed' programs. In the event of newly discovered fragments, it is common for there only to be a single exemplar of it in use.

These two training corpora can be used to train a probabilistic estimator neural network. The programs are fed randomised inputs, and their resultant IO mappings fed into the neural network as its training features. The training labels are either 0 or 1, for absence or presence of code fragment, respectively. The network has a sigmoidal output, and is trained to minimise the mean squared error on its training data set, and so is able to produce a probability for any given IO example as to whether the program which generated the IO mapping contained the code fragment or not.

This, of course, has the issue that multiple implementations of the same functionality exist, and it is quite possible that an alternative implementation either has or lacks the fragment, and thus the training data could be considered 'misleading'. This remains an open problem.

Once new problems are encountered, the process repeats, but the GP process can employ fragments which have a high estimated probability of presence in a GP-produced solution. The GP has access to a mutator operator which inserts entire code fragments into the source code, selecting only those which have an estimated probability of > 0.5 and biasing towards those with higher probabilities.

After a solution is accepted, it can be evaluated to determine if any proposed fragments exist within it. If they do, new training corpora are generated, as new seed programs exist to demonstrate how this fragment can affect program IO. The

Algorithm 1 An overview of the process for transferring learning between GP synthesised code, allowing online learning from IO examples only

```
1: while unsolved_problems_exist do
2:   for all stored_fragments do
3:     present problem's IO to fragment's associated NN
4:     if NN estimate of probability $< 0.5$ then
5:       continue
6:     end if
7:     $S = arbitary\_ranking\_heuristic$
8:   end for
9:   run GP on problem with fragment with highest $S$
10:  if GP returns valid solution then
11:    extract all code fragments from solution
12:    for all extracted_fragments do
13:      generate training set $P$ from all found solutions featuring fragment
14:      generate training set $N$ from all found solutions without fragment
15:      train an NN to discriminate between $P$ and $N$
16:      evaluated trained NN on existing found solutions
17:      if NN accuracy on found solutions $> 0.5$ then
18:        store NN + fragment for future deployment
19:      end if
20:    end for
21:  end if
22: end while
```

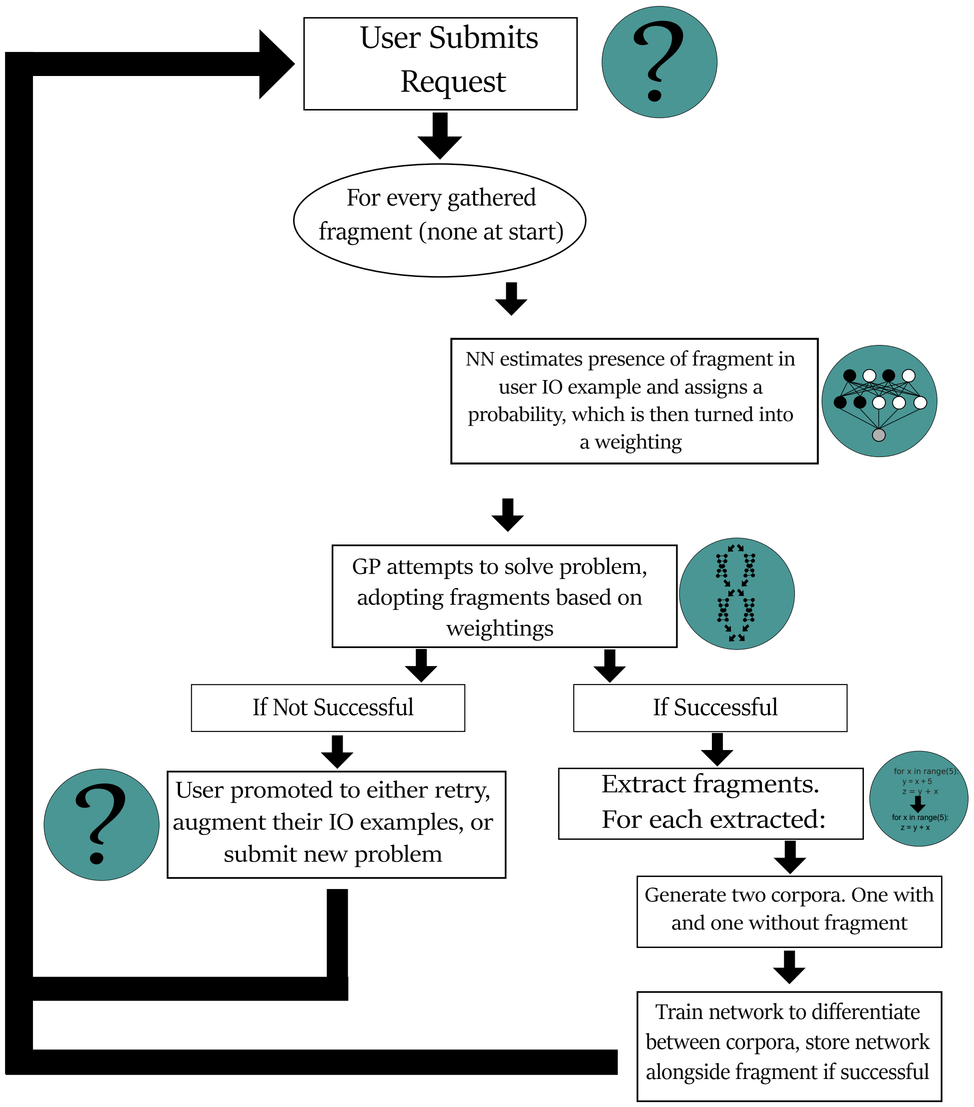
fragment's estimator neural networks can then be retrained, with a broader coverage of program space than would have been created by their single positive case (which would have been their first training set).

These two training processes allow the system to learn from past success, by training networks to recognise the presence of code fragments. A single network is used for each fragment (overwriting the previous one each time a new training corpus is produced), in an attempt to avoid scaling issues. The minimum required VC dimension for this is simply 2, as it only seeks to separate two classes, the IO examples which appear to have the assigned fragment, and those which do not.

Training never need end, with each success refining the networks' abilities to recognise code fragments and occasionally adding novel code structures to the repository.

The end to end approach is described in pseudo-code in Algorithm 1, and illustrated in Diagram 3.1.

The subsequent chapters discuss the experiments undertaken to determine how


User Submits Request
For every gathered fragment (none at start)
NN estimates presence of fragment in user IO example and assigns a probability, which is then turned into a weighting
GP attempts to solve problem, adopting fragments based on weightings
If Not Successful
User promoted to either retry, augment their IO examples, or submit new problem
If Successful
Extract fragments. For each extracted:
Generate two corpora. One with and one without fragment
Train network to differentiate between corpora, store network alongside fragment if successful

```
for x in range(5):
    y = x + 5
    z = y + x
```

```
for x in range(5):
    z = y + x
```

Figure 3.1: Diagram of the end-to-end code synthesis system

to best tackle the problem of code synthesis. They evaluate the various components of the system in the order they will be required by the architecture, starting with the data sources and ending with the full end-to-end system's design. The concept discussed in the preceding section is implemented in Experiment 6, after a set of experiments which confirm the effectiveness of its underlying systems.

It builds upon the direction suggested by the background section, of attempting to combine the abilities of neural networks to recognise high-level properties of the input-output examples provided with the powerful program space navigational capabilities of genetic programming. It also continues with the concept of Transfer Learning, as seen in the background section, which is anticipated to allow new progress to be made in the field of code synthesis by allowing previous success to be built on, in a fashion similar to how a human might learn to code, from easier problems to harder ones.

3.2 Argument for combined GP and NN From Scaling

If a neural network can successfully reproduce code simply by increasing its number of weights and available training data, the entire idea above becomes irrelevant. Certain mathematical concepts seem to suggest that its capabilities would not grow linearly. According to one mathematical analysis of the properties of neural networks, if neural network architectures scale in a similar fashion to simple feed forward networks, they may lack the ability to generalise to the sizes of programs required to be useful industrially. This section assess this, to support the argument for using a combination of genetic programming and neural networks, as opposed to attempting to expand neural networks alone.

Machine Learning algorithms can be studied in terms of their Vapnik–Chervonenkis (VC) dimension (Vapnik and Chervonenkis (2015)). A machine learning algorithm can be considered to take an arbitrary input and select a 'class' from its output domain. In many cases these classes are simple enumerable categories. For example in a binary Boolean domain, the output domain is {True,False}, and the two categories are 'True' and 'False'. The number of classes a machine learner can distinguish between is its VC dimension. In that case, the machine learning algorithm has a VC dimensionality of 2. This limit may not be due to its output domain being restricted, but due to the properties of the classifier itself. A linear classifier, which operates based on the Boolean function $f(x) = if((x_0 * k_0 + x_1 * k_1 ... x_n * k_x) > k \ then \ 1 \ else \ 0)$ structurally cannot separate more than two cases, as it depends on a binary conditional. It defines a plane within its input space, and returns based on the side of the plane the given input point exists. Adding an additional decision boundary plane allows for 4 cases of how input points may exist relative to these planes, therefore a VC dimensionality of 4.

This generalised DeepCoder approach discussed in the background section can be considered to be a machine learning algorithm selecting between classes, where those classes are subsets of program space to search within. Work exists in the literature with regards to the VC dimension of neural network, with Bartlett et al. (2019) demonstrating that a fully connected architecture using the reLu activation function has an upper bound on its VC dimension of $VC = C * L * log(w) * w$, where L is the number of layers in the network, w is the total weight count and C a constant whose value remains to the author's knowledge unproven. While this constant is not known, the function without it can be seen as the 'Big O' complexity function for a feed forward neural network, an assessment of how it scales with required VC dimensionality, and thus its theoretical computational tractability. While other neural network architectures exist, we will discuss this case initially.

When presented with the data representing an IO example to match, the network processes the data, then presents the searcher with a set of programs to search over (and a ranking/ordering, which is irrelevant to this part of the discussion). The maximum number of different sets it can direct the searcher towards is its VC dimension, with a proven upper bound of $VC = C * L * log(w) * w$.

We assume the searcher has an upper bound of T programs it can search with its finite computational resources. While program evaluation may vary in time, we define T as the best possible case, the maximum number of programs which can be evaluated on the hardware, and allow the benefit of the doubt that all evaluations will equal this performance.

The total number of programs able to be generated is equal to $T * VC$, if the subsets of program space do not overlap (the most efficient output configuration the network could theoretically reach). As program space size increases, we see that to maintain constant proportion coverage (including the option for full coverage, where $T*VC = program\_space\_size$), VC must scale linearly with the size of the program space.

VC itself scales as $w * log(w)$, as C is a constant and L is a fundamental part of the architecture of the network, and cannot be extended arbitrarily without altering training outcomes (He et al. (2016)). There exists a fixed computational cost for w, as each weight represents a floating point operation to perform.

$program\_space\_size$ varies exponentially with line count, however. If all line configurations are legally possible for every line of a program the space of all possible programs increases as $options\_per\_line^{number\_of\_lines}$. Obviously in most languages this is not a valid assumption, as certain options are not possible without preceding others, such as variables requiring declaring before they can be read from, but it serves as a rough guide to how program space grows.

The DeepCoder architecture therefore has two mechanisms to scale in response to being faced with longer problems. It can increase its T, which linearly increases the associated computational cost, specifically the time taken to search the problem subset returned by the NN. Alternatively it can increase its VC by increasing its weight count w, which increases its VC as $O(w * log(w))$. This second way linearly increases the computational cost associated with the floating point operations required to compute the weightings' outcomes. Each line of problem length, however, could be increasing the problem search space size exponentially. In the language we used in published work and through this thesis, each line added to the maximum permitted program size involves an increased program space size greater than three orders of magnitude. This does not preclude the DeepCoder architecture from success, but casts doubt on its ability to generalise from smaller program spaces to larger ones.

To put numbers to it, if the language is at a point where each line permitted increases the problem space size by three orders of magnitude, to maintain the same degree of performance (either the same marginal probability of an out-of-time outcome or to maintain full coverage of problem space) the computational cost would need to increase by a factor of 54. This assumes we increase both T and w by the same factor of 27 to optimise the growth in performance which follows $O(T * w * log(w))$ (we split our new resources equally between the two components, as the gains achieved are multiplicative). To go from a machine capable of synthesising a ten line program to one capable of synthesising a twenty line one therefore requires a computational increase of 10^{17}. If a program has 5 variables defined and able to be written to, and only 8 operators, each able to write to a variable and read from two others (such as addition or multiplication would do), it already allows 1000 new permutations for each line.

DeepCoder already reports a time taken of 2654 seconds to resolve 60% of its testing problem corpus of 500 problems of length 5. They report a problem space of size 10^{10} for this configuration, and the paper's experiments indicate that it may well scale in time approximately linearly with problem space size, as they reported a time of 0.11 seconds for the same proportion solved on the smaller test with a problem space size $2 * 10^6$. If the problem space rises to a size of 10^{30}, as could readily caused by a 10 line program space with the combinatorial explosion caused by the options allowed by multiple-input functions (DeepCoder's DSL only includes single-input functions), it is possible search times could rise to just under one million years.

As stated above, this does not mathematically prove the architecture is not capable of synthesising useful programs, and does not place any upper bound on the nature or complexity of a program which can be synthesised by a neural network which returns a subset of program space over which an exhaustive searcher will search. It does however, suggest that any architecture which relies on a neural network able to perform mappings between large domains of IO example inputs and usefully large program spaces should not be assumed to generalise to larger problem spaces with plausible increases in computational hardware. There is a strong possibility that its architecture is at its limit, and even large increases in computational resources would only yield modest gains in maximum problem complexity.

Chapter 4

Training Corpus Generation

4.1 Experiment 1: Designer-Guided Training Corpus Generation

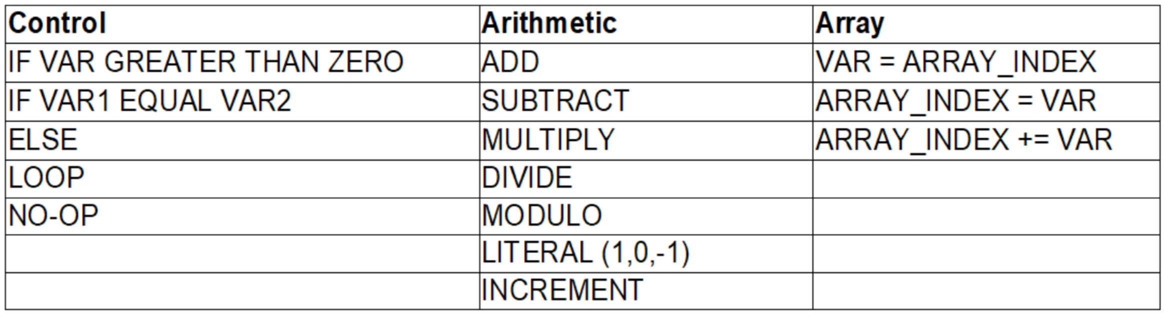
In terms of architecture flow, the first part of a machine learning system for use in the context of code synthesis (or indeed in any context) is the acquisition of data from which it can learn. If we are to train a machine learner we must produce a training set. This section discusses this PhD's research regarding this problem, and outlines the experiments undertaken with regards to this aspect of the work.

For all tasks for which a machine learning algorithm was employed in this work, it was necessary to sample program space, to produce a set of programs to use as examples for the machine learner. How to select programs from program space in a way which maximises learner effectiveness may prove highly influential on overall success of our approaches. As identified in Shin et al's work Shin et al. (2019b), the inputs fed into the problems, to generated input-output mappings, is also highly relevant.

This first experiment of this work was designed to assess how neural network machine learners can best be provided with training examples. The area of focus was in program selection, determining which subset of the vast program space would be selected to generate Input-Output (I/O) mappings to serve as training examples. This is a preliminary experiment, so the language is a constrained version of the full one, and does not employ any 2D arrays.

The experiment focuses on studying how the performance of a neural synthesis system can be maximised on a 'Human-Useful' problem corpus. These are a collection of short functions which could be seen in a human-written program and serve a recognisable purpose.

As a preliminary study, this featured a large degree of designer involvement, to



Control	Arithmetic	Array
IF VAR GREATER THAN ZERO	ADD	VAR = ARRAY_INDEX
IF VAR1 EQUAL VAR2	SUBTRACT	ARRAY_INDEX = VAR
ELSE	MULTIPLY	ARRAY_INDEX += VAR
LOOP	DIVIDE	
NO-OP	MODULO	
	LITERAL (1,0,-1)	
	INCREMENT	

Figure 4.1: The operators available in our simplified language.

study it in an initial fashion. This would establish what the later fully-automated systems should be working towards.

This work was published in the literature under the title “General Program Synthesis using Guided Corpus Generation and Automatic Refactoring" Wild and Porter (2019).

4.1.1 Simplified Language

In this test a reduced form of the language was employed, to simplify the problem, in order to rapidly study the underlying techniques.

To simplify the design of the neural network, the language is mapped onto the output neurons using a uniform set of possibilities per line. Each line of a program can have the same 1,332 different options, derived from 15 operators (see Table 4.1), from variable declaration to addition or a loop header. Once a program has been chosen, it is checked to see if it is syntactically coherent and the system then automatically corrects programs which are not. In C-like programs this creates two main corrections: cases in which there are too many ‘closing braces’, and cases in which there are too few (an unterminated loop). For the former case hanging braces are simply replaced with a no-op. In the latter case a closing brace is inserted at the very end of a program for any un-closed control blocks; in addition, any un-closed loops are converted to conditional blocks rather than loops. By taking this approach to neuron behaviour uniformity, the neural network does not itself have to learn special cases which limit what each line can be based on prior lines, which would create a much more complex network structure (which potentially might create a more difficult learning problem). Uniformity here simply means that the behaviour of any output neuron codes for the exact same way of writing a line when translated to code regardless of content of previous lines.

As further restrictions for this experiment, in all of the tests use programs 9 lines long, padded with the NO_OP operator. 6 integer variables are allowed to be accessed by the programs, of which two are fixed and unable to be written to. All of the tests involve passing a single array and a single standalone integer into the program. The two fixed integer variables are the input integer and the length of the input array. The program then has read and write access to both the input array and a second array used as output. These limits allow a wide range of functionalities, while still imposing limits to maintain the problem within computationally tractable sizes.

4.1.2 Neural Network and Search Architecture

The code synthesis architecture combines a neural network, used to derive an ordered ranking of possible options, with a search process which iteratively tries these ranked programs up to a configurable search depth.

For this particular study it is assumed every program can take two parameters: an integer array of length 8 as the first parameter, and an integer as the second. It is also assumed that every program returns an integer array of length 8. Every cell in an array can hold a value between -8 and 8, while the integer parameter can hold a value between 1 and 4. Reducing the range of the integer parameter to only positive non-zero values simplifies the search space, as they can always meaningfully use the parameter to refer to an array index.

While the language is capable of representing much more diverse function specifications and numerical ranges (equivalent to C), these restrictions are a first step to simplify the search space and neural network complexity. The crucial extension targetted is the ability to use LOOP and IF statements, allowing more complex programs in terms of flow than are possible in other code synthesis approaches. A trade off is accepted in terms of program length in return for being able to handle a new class of program.

The neural network is then designed as a standard feed-forward architecture as follows. The input layer uses 1,700 input neurons to take 10 I/O examples concatenated together. The output structure uses 9 layers, one for each potential line of a program; each such layer consists of 1,332 neurons, one for every possible way the respective line could be written (including the possibility of a no-op). Internally we use 8 residual layers, each consisting of two dense layers with a width of 512 and an additive layer skip (shown to improve deep networks Wu et al. (2017); Kawaguchi and Bengio (2018)), and using the ReLu activation function. Dropout was used on all layers, with a probability to keep of 0.75. A softmax activation is used for the output layers, and a crossentropy loss function. The optimizer was the Tensorflow implementation 'RMSPropOptimizer', with learning rate 10^{-5} and momentum 0.9.

The neural network is trained by automatically generating a corpus of example programs; the mechanics of this generation are described in detail in the next subsection. For each generated program in this corpus 10 input/output examples are randomly generated for that program. During training, randomly generated I/O examples are fed into the neural network's input layer as 170 integer values (each I/O is being composed of 8 values for the input array, one value for the input integer, and 8 values for the output array, this creates 17 values for one I/O example and thus 170 values in total for 10 I/O examples). Encoding integers as 10-bit binary numbers for input to the neural network was experimentally shown to perform better than using scalar inputs, and so the network has a total of 1,700 input neurons. The network is trained by back-propagating the corresponding output layer neuron values from the actual source code of the corpus program associated with these I/O examples.

Once training is complete, in the testing phase only the 10 I/O examples for a desired program are supplied and the neural network's probability distribution over its output layer neurons is used to create a ranked list of programs to search across, from most to least likely. The highest-confidence program would therefore be generated by selecting the highest activity neuron from each output layer. Each layer mapped to a line in the program being generated, and each neuron mapped to one of the 1,332 valid statements which could appear on that line. The 9 highest-activity neurons, one from each of the 9 output layers, therefore map to 9 statements which then make up the highest-confidence program.

To generate a volume of program space, the N highest ranked neurons are chosen per line, giving N ways that particular line could be written in the sampled program. The search volume would therefore consist of every combination of these options, i.e., $number\_of\_options\_per\_line^{number\_of\_lines}$. For this experiment, when not otherwise noted, 4 options are used per line for standard programs, and 6 when searching within the human-useful program set (Programs in this set are listed in Figure 4.1).

This system is illustrated in Figure 4.2.

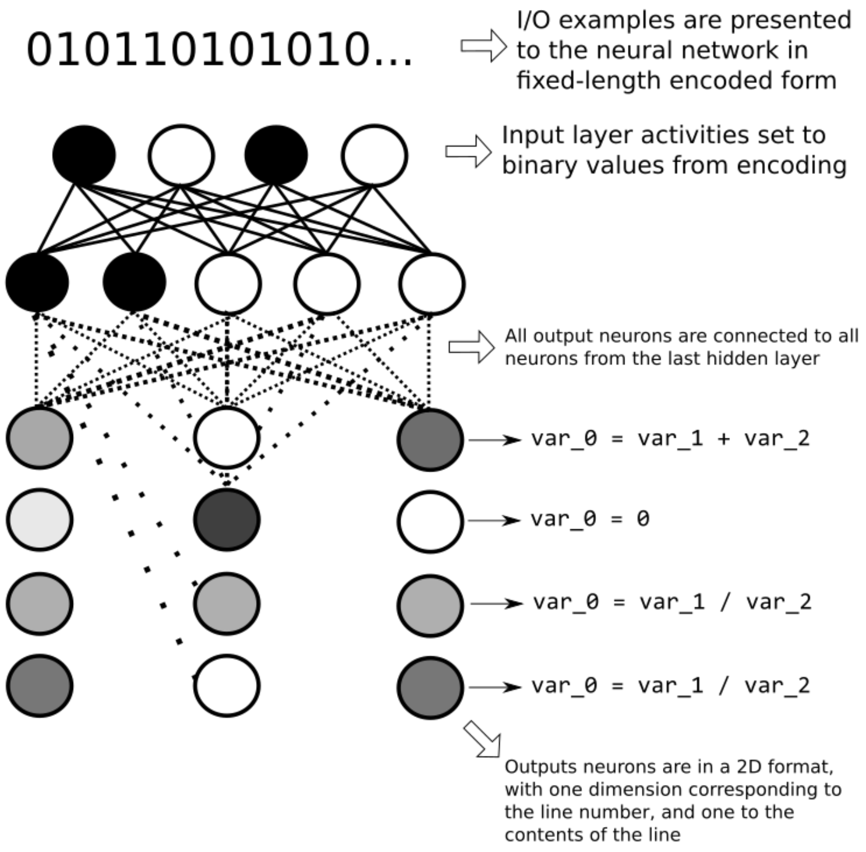

010110101010...
I/O examples are presented to the neural network in fixed-length encoded form
Input layer activities set to binary values from encoding
All output neurons are connected to all neurons from the last hidden layer
var_0 = var_1 + var_2
var_0 = 0
var_0 = var_1 / var_2
var_0 = var_1 / var_2
Outputs neurons are in a 2D format, with one dimension corresponding to the line number, and one to the contents of the line

Figure 4.2: Diagram of the neural network, with three lines of outputs, and a single line's output neurons described.

4.1.3 Corpus Generation

In a simple Domain Specific Language, a training corpus for a neural network could be generated by sampling uniformly at random from the space of all possible programs (Chen (2017)). For these purposes, however, the search space of the more general-purpose language is far too large for uniform random sampling to be effective. When sampled in this way, the resulting corpus of programs is highly repetitive, each program has a high probability of being made up of only (or mostly) lines of code that have no effect, and very few programs contain condition or loop elements (which feature heavily in human-useful programs).

As an alternative to uniform random sampling this experiment designed an approach which combines genetic programming with a set of abstract search biases and a dissimilarity measure. The generator starts with a seed program, which is an abstract problem reflecting the kinds of search biases that are needed; for example a program that uses a loop and a conditional branch, and which reads all of the input array values once and writes each cell of the output array. Starting from this seed program, the genetic algorithm creates iterative populations of mutations. Within a population, it promotes code length and an even distributions of all operators, and it penalises writing to loop iterator variables. Finally, mutated programs are only accepted into a population if they are behaviourally dissimilar to the rest of the population. This similarity is measured by feeding 25 randomly generated inputs to each program, and marking the programs dissimilar if any of their output arrays contain a single different value as a result of the inputs. Programs are also rejected if any program reads from or writes to the same memory address in an array twice, further reducing the search space. To gain good learning coverage of flow control, this experiment seeds five separate sub-corpuses to form the overall corpus. The first had 0 flow control operators. The second had 1 loop only. The third had 1 loop and 1 CONDITIONAL_GREATER_THAN_0 operator. The fourth had 1 loop and 1 CONDITIONAL_EQUALITY operator. The fifth had 1 loop, 1 CONDITIONAL_EQUALITY and 1 ELSE operator.

The result of this generation process was a diverse set of $10,000$ functionally distinct programs, split between the 5 sub-corpuses of $2,000$ each. This experiment determines functional similarity by feeding both programs a set of 25 randomly generated inputs and checking for any difference. It then splits these programs amongst training, testing and validation for the neural network. Training received $8,000$ programs, the other two corpuses received $1,000$ programs each. As a result, each corpus' programs were functionally dissimilar, with no program's functionality replicated between corpuses. Note that none of the set of human-useful programs is involved in training the neural network; all such programs are therefore unseen by the system.

4.1.4 Automatic Corpus Refactoring

The corpus generation approach tries to train the neural network with a diverse set of programs that facilitate its ability to synthesise human-useful programs. However, corpus generation itself does not necessarily maximise the neural network's internal generality or its use of available model representation space.

A novel approach to enhancing the generality and model efficiency of the neural network was tested, by altering the corpus based on the network's own success rate – an approach termed in this work *automatic corpus refactoring*.

The neural network is first trained using the corpus generated as above. It is then asked to locate every program in the training corpus by being given the set of I/O pairs which should result in the given program being found. Because the neural network outputs a ranked list of potential programs, the actual program match may be 10's or 100's of programs down this ranked list. However, during experiments it was observed that a *functionally equivalent program* would often exist earlier in the ranking than the exact-match program in the training corpus.

In corpus refactoring, a test was run to see if such a functionally equivalent program exists earlier in the ranking, and if so it *replaces* the training program with this equivalent version. The system then retrains the neural network again (with weights re-initialised) based on this new corpus. It can perform this refactoring iteratively, using a new corpus to again replace programs with earlier-found equivalents, until the performance converges to a maximum. As the results demonstrate, refactoring in this form increases performance not just on the training corpus, but also on the testing corpus and on the number of human-useful programs that were correctly constructed – in other words, by adjusting its own training corpus without actually adding any new information, the system is able to find more programs in total than it previously could.

Empirical testing described below indicates that as many as 0.8 of the found programs could be replaced in this fashion, replacing the generated corpus' implementation of the function with the neural network's preferred implementation.

4.1.5 Results

This subsection firstly examines the system's overall code synthesis performance in its intended normal configuration. This allows examination of its performance, when attempting to solve a human-defined testing corpus of unseen programs. We examine the effects of our automatic refactoring (AR) technique over a set of iterations, to isolate its performance from the initial success of the corpus generation and neural network training steps. AR allows us to improve the performance of a system by adapting its training corpus in response to its current behaviour.

This experiment then investigates the genetic corpus generation approach in further depth, by performing ablation studies on its ‘requirements’ and fitness function. Following these two studies, it attempts to shed light on the performance gains produced by the AR technique, by examining the changes it makes on the corpus.

The approach is evaluated with a set of ‘human-useful’ programs that the synthesis system is required to find – which are distinct from the set of programs that the synthesis system finds during its own automated corpus generation phase. These were chosen by the author to be similar to programs which a novice first year programmer might encounter and be tested on and are described in Table 4.1.

4.1.5.1 Program Synthesis

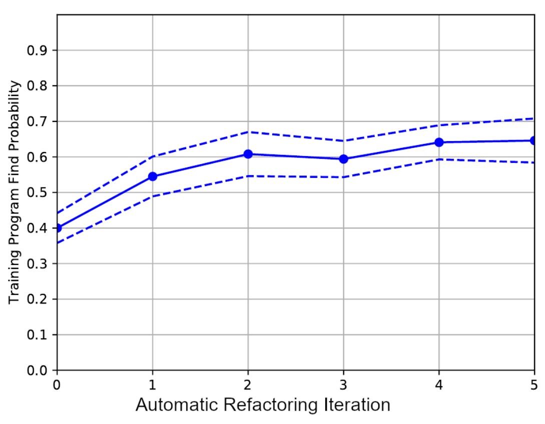
To test general program synthesis capability the experiment runs the end-to-end approach 10 times to gain average results. There are two sources of stochasticity in our approach: the way in which the corpus generation phase works, which is based on randomised mutations; and the way in which the neural network is initially configured, which uses randomised starting weights prior to training. The experiment runs corpus generation, five rounds of automated corpus refactoring, and then presents the input/output examples for the set of (previously unseen) human-useful programs to see how many the system can find.

Two encoding schemes were tested, one with every integer encoded simply as a real value, giving an input length of 170 (8 in the input array, 1 integer, 8 in the output array, for each of the 10 examples), and a binary encoding in which each of these values was represented by a 10-length binary vector (1 bit for sign, 9 for magnitude), for 1700 input values. Without automatic refactoring, on two sets of 5 corpora, this experiment found that binary encoding produces a success rate of 0.262 ($\sigma = 0.0241$), while real-value encoding produces a success of 0.185 ($\sigma = 0.0174$). This shows that the binary encoding scheme is highly preferable. All later tasks involving feeding integer-based input-output examples in a neural network in this thesis therefore use the binary encoding scheme.

The results are shown in Table 4.1, detailing both the find rate before any corpus refactoring and also the find rate after the final round of refactoring. For each successfully found program, the neural network has output source code which correctly derives the output from each corresponding input. As an example, for the program “max(value,param)”, the array could be [-5,3,-2,3,-4,1,5,8], the parameter ‘1’, and the output [1,3,1,3,1,1,5,8]. From these results it is seen that the system locates an average of 38% of the human-useful programs as a result of its initial corpus generation process; this rises to an average of 44% (and a maximum of 60%) after five rounds of corpus refactoring. Examining individual programs in this target set, it is seen

Program name	**Without AR**	**With AR**	**GP n=60**	
Absolute *	18.18%	**52.94%**	0.00%	The output array is the absolute value of the elements in the input array
Add one	18.18%	52.94%	**85.00%**	The output array is the value of the elements in the input array plus one
Add param	72.73%	88.24%	**96.67%**	The output array is the value of the elements in the input array plus the parameter integer
All negatives to zero *	90.91%	**94.12%**	0.00%	The output array is the maximum of (the value in the input array;zero)
Array length	90.91%	94.12%	**100.00%**	The first value of the output array is the length of the input array, the remaining values are 0
Count param *	**90.91%**	88.24%	8.33%	The first value of the output array is the number of times the parameter integer appears in the input array, the remaining values are 0
Iterator up to param	63.64%	41.18%	**100.00%**	The first N values of the output array are set to their index, the rest are zero, where N is the parameter integer
Keep above param	0.00%	0.00%	0.00%	The first N values of the output array are the corresponding values in the input array, the rest are 0, where N is the parameter integer
Max(value, param) *	0.00%	**5.88%**	0.00%	The output array's values are the maximum of either (the input array's corresponding value;the input parameter)
Min(value,param) *	0.00%	**5.88%**	0.00%	The output array's values are the minimum of either (the input array's corresponding value;the input parameter)
Modulo 2	0.00%	0.00%	0.00%	The output array's values are the modulo 2 of the values in the input array
Modulo param	90.91%	**94.12%**	21.67%	The output array's values are the modulo parameter of the values in the input array
Multiply by param	90.91%	**94.12%**	0.00%	The output array is the input array, but offset by N rightwards, losing the last value and setting the first value to zero, where N is the input parameter
Negative *	54.55%	**94.12%**	1.67%	The output array's values are the negative values of the values in the input array
Offset by one	27.27%	11.76%	**66.67%**	The output array is the input array, but offset by one rightwards, losing the last value and setting the first value to zero
Offset by param	0.00%	0.00%	0.00%	The output array is the input array offset by N rightwards, losing the last value and setting the first value to zero, where N is the parameter
Return max *	0.00%	0.00%	0.00%	The output array's first value is the highest value in the input array, the other values are 0
Return min *	0.00%	0.00%	0.00%	The output array's first value is the lowest value in the input array, the other values are 0
Reverse	0.00%	0.00%	0.00%	The output array is the reverse of the input array
Sum values	54.55%	**76.47%**	26.67%	The output array's first value is the sum of all the values in the input array, the other values are zero

Table 4.1: Percentage success rates for two experiment sets, with and without the automated corpus refactoring stage (the first set averaged over 11, and the second over 17 runs). A simple genetic programming algorithm, using the same linguistic constraints, is used as baseline. It can be seen that GP (as described in Subsection 4.2.1) succeeds on simpler problems, but has lower performance when a conditional statement is required. Highest success rate bolded. Programs requiring conditionals marked with '*'



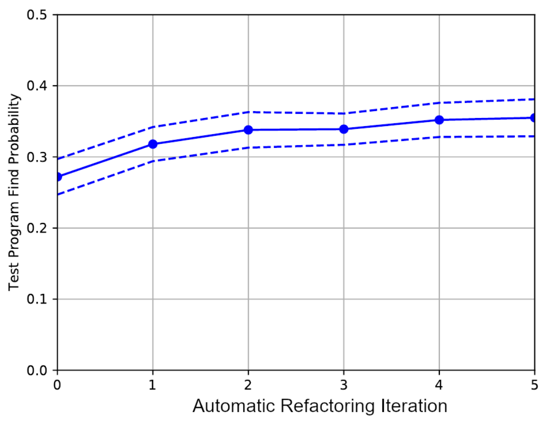


Table 4.2: Success rate on training (left) and test (right) corpus, over each iteration of automatic refactoring, starting from the unmodified corpus, with Standard Deviation

that the majority of find rates tend to increase, while a couple of find rates (for example the offset-by-one program) notably decrease. It can be speculated that the decreases in some program find rates may be caused by those programs lying outside the generalised space into which the neural network moves during corpus refactoring.

The experiment next examines the find-rate of the training, test and human-useful program set over each iteration of automatic corpus refactoring. These results are shown in Fig. 4.2 for the training and testing sets, and in Fig. 4.3 for the human-useful set.

Both the training and test data set show a steady increase in the find-rate of programs from the respective set. For the training set, which shows a find-rate increase from 0.4 to 0.65 (where a value of 1.0 would be all programs found), the process of self-adjusting the training corpus in automatic refactoring clearly shows an enhanced ability to correctly locate more entries in the training corpus. The effect in the test corpus is similar, in this case showing an increase from 0.27 up to 0.36. However, in the case of the test data set the result is much more significant. The increase in find-rate here (i.e., for programs which the system has never seen before) indicates an unexpected phenomena: having the neural network's training corpus refactored, without adding any data, allows the neural network to locate more unseen programs than it previously could. It is worth noting that performance decreased in some cases. This is potentially due to the neural network specialising to a particular form of program (the most common) at the expense of others. While this specialisation is overall beneficial, some degradation occurs in certain types of program.

A similar effect is seen in the human-useful programs over successive refactoring iterations, as shown in Fig. 4.3. Again, all of these programs are unseen by the

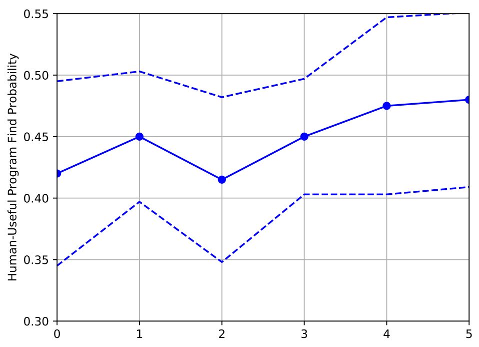


Figure 4.3: Success rate on the human-useful corpus, over each iteration of the automatic refactoring process, with unchanged corpus as first data point. Corpus size is 20, so each increment of 0.05 corresponds to an average of one program found. 1st Standard Deviation displayed

system during training, but reshaping the training data enables more of them to be successfully synthesised. This suggests that the use of automated refactoring approach perhaps causes the neural network to become more generalised in its capabilities. However, the data in Fig. 4.1 provides a more mixed picture: here we see that find-rates for most programs increase after refactoring, but some find rates actually decrease.

4.1.6 Requirements in corpus generation

The effects of the initial requirements on corpus generation can now be examined. These requirements are used as input to the genetic algorithm to guide its generation of a set of programs on which the neural network is trained. As a consequence of this training, the neural network is then able to find (or not find) a set of previously unseen human-useful programs. The precise nature of these requirements for corpus generation are therefore an important, if indirect, element of how successful our synthesis approach is at finding programs after training.

This subsection examples how the use of different requirements affects synthesis success. The complete set of requirements, used across all of the experiments reported so far, includes three major categories as follows.

4.1.6.1 Array Access

This requirement is that all programs containing a loop operator must access every element of the input array. This requirement was included to overcome a perceived problem in the input-access of generated programs. These would often access their inputs in ways which human-written programs rarely would, such as only reading a single element of the input array, or altering their loop iterator and as such skipping elements.

4.1.6.2 Program Flow

This requirement involved subdividing the corpus into 5 sub-corpuses, each with its own requirement as to how the flow-control operators should be used. The first corpus required all its programs to have no flow-control operators at all. The second corpus required only a single loop operator. The third required a single loop operator and the first type of conditional operator. The fourth required a single loop operator and the second type of conditional operator. The fifth type required a single loop, the first type of conditional operator and an else block. These particular requirements were chosen to demonstrate interplay between flow operators, which weren't featured together frequently in a randomly generated corpus. For each of these corpuses, a single "seed" program was supplied. This program was what was considered to be the "maximally simple" implementation of the requirements; as an example of this the loop-only requirement, from the second corpus, would read in all input values, then write them out unchanged to the output array. The seed programs were implemented due to the genetic search's inability to start generation without them.

4.1.6.3 Genetic Fitness Function

Lastly, the genetic algorithm fitness function rewards particular operator ratios: all operators are expected to be used at least once, with flow control operators in particular weighted twice as highly as others. This was done to promote the use of flow-control, while penalising operators repetition. This was necessary to move away from the "maximally simple" seed programs to those with more commonly useful features. This is termed the 'complexity maximisation' heuristic. It occurs by taking an 'ideal' distribution, which is for a program to have one of every operator (including flow) control. A vector of length equal to the number of operators can be computed for each function, where each value is the number of times that operate exists. This vector can then have its Euclidean distance compared against a vector of all 1s (representing a program with one of every operator). This 'perfect' program is impossible due to there being more operators than lines available. The program fitness is then divided by the Euclidean distance produced, the distance being the complexity heuristic.

The effects of the above requirements are examined by selectively switching them off during corpus generation and comparing how many of the human-useful program set is found as a result. The results are shown in Fig. 4.4, in which each experiment was run 5 times and the data averaged.

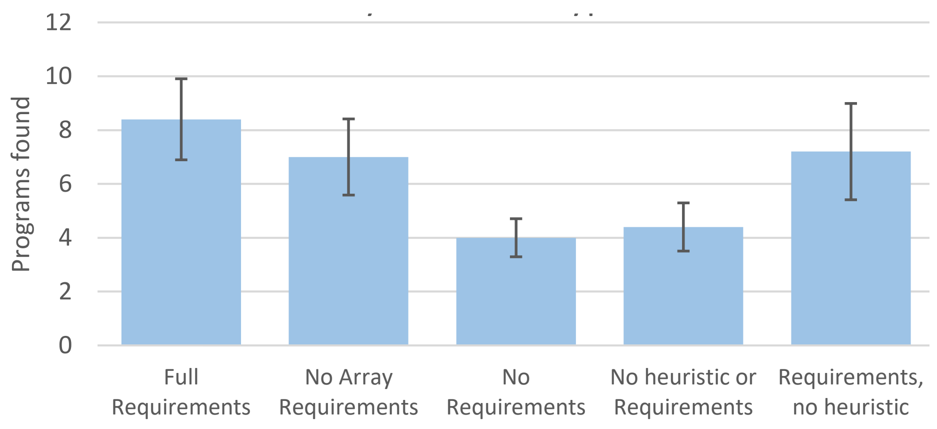


Figure 4.4: Average performance for sets of corpora with varying requirements for constituent generated programs.

The first test shows the full set of requirements, as used in the earlier experiments reported in this section. This achieves the highest performance of any experiment, finding an average of 8.4 programs ($\sigma = 1.5$) from the human-useful target set. The second experiment removes the array access requirement, but keeps the program flow and fitness function heuristics. This performs slightly worse, achieving an average of 7 human-useful programs ($\sigma = 1.4$), indicating that most of the programs are in an area of the total search space in which the input array is uniformly accessed. The third experiment removed both the array access requirement and the program flow corpus generation technique, leaving only the fitness function heuristics. This resulted in the worst overall success, with only an average of 4 programs found from the set ($\sigma = 0.71$).

All requirements were then removed as well as the fitness function heuristics, which actually shows a slight increase in performance with an average of 4.4 ($\sigma = 0.98$) programs found. Finally, an experiment was run with only using the array access and program flow requirements and not the fitness function heuristics, which results in the second-best performance overall – indicating that these requirements are more important to success than the fitness function heuristics.

Altogether, these results support the hypothesis that achieving good performance on the human-useful corpus requires a set of corpus generation biases that are reflective, at a very abstract level, of the typical form of useful programs. A corpus generation function for producing training sets for neural networks should therefore attempt to replicate the success of these restrictions, to maximise a neural network's ability to learn the useful regions of program space.

4.1.6.4 Effects of Corpus Refactoring

This subsection examines the effects of automated corpus refactoring in more detail, to better understand why it enables more programs to be found without adding any new data to the system. This work characterises this as not adding new data because, even though the training corpus is modified, it is modified only as a result of the neural network's own output from the initial training corpus; the only thing being 'added' is therefore the neural network's apparent preference for which precise form of a target training program to use, but this preference is itself entirely derived from the original training corpus and the neural network's inherent behaviour. This subsection therefore attempts to better understand why this effect occurs, in so far as is possible with the black-box nature of neural networks.

In broad terms, the use of feeding output of one neural network as training labels to another has been demonstrated previously in teacher-student distillation network training Hinton et al. (2015); in this case however it is believed the success is in fact due to an interplay between the network and the search process. This subsection analyses this effect using the entropy shown by output layers before and after automatic refactoring. These experiments help to verify whether or not the neural network is 'self-generalising' as a result of its search process, or if in fact it is specialising to certain kinds of program in which it tends to become an expert.

These experiments measure the entropy of each output layer of the neural network, where each layer corresponds to one line of code. As discussed in subsection 4.1.2, each output layer of the network can select from one of a fixed set of possible operations for that line of code – where each option is represented by one neuron. The highest-activated neuron in an output layer is taken as the network's best guess for this line of code. It is hypothesised that one of the reasons for corpus refactoring finding extra programs is that the network becomes better generalised in its representation of algorithms. This experiment tests this theory by examining the entropy of each output layer – in other words, across all programs, how 'specialised' is each output layer to always choosing the same operation for their line of code, versus their ability to represent a balanced spread of output options. The more balanced the spread is for each output layer, without losing the ability to synthesise programs, the more generalised the neural network may be.

Theil inequality metric is used to measure the 'inequality' of each line. Maximum inequality (only one option ever used) would be minimum entropy (perfectly predictable). To compute this it finds the probability of every option for every line being chosen, across all the programs in the training corpus. If every option were chosen with equal probability, they would each be the average, μ, which is equivalent

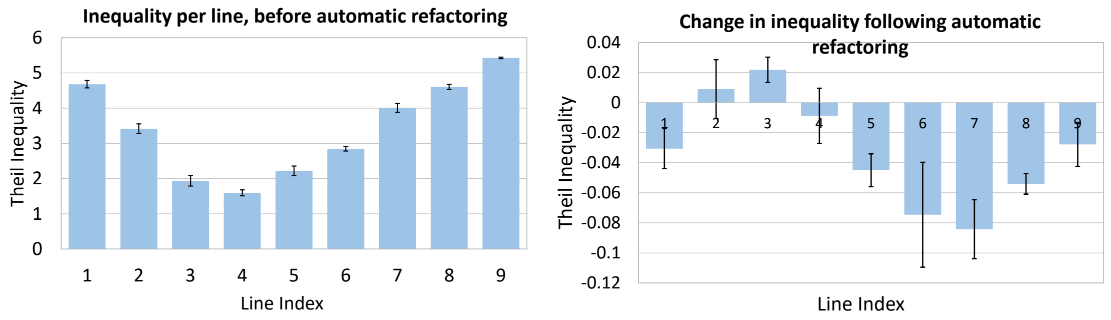


Figure 4.5: Reverse entropy of operator distributions by line, as measured by the Theil index. Lower values imply a more even distribution of operator use for a particular line, therefore higher entropy.

to $1/N$, where N is the number of options per line. The Theil index is computed by:

$$T = \frac{1}{N}\sum_{i=1}^{N}\frac{x_i}{\mu}ln(\frac{x_i}{\mu})$$

Where x_i is the use probability of option i.

Fig. 4.5 shows the results of this experiment. On the left can be seen the entropy of each output layer before any refactoring has taken place, and on the right can be seen entropy after five refactoring iterations. A clear trend can be seen towards a more balanced ability for each output layer to select a broader range of options, supporting the hypothesis that the refactoring process may aid in generalising the capacity of the neural network.

It can also be inferred from these experiments that program length becomes more consistent after refactoring. This is to say, the number of non-empty lines are counted (which can appear on any line post-refactoring), it tends towards a closer average across all programs (the length goes from 7.11 lines with a standard deviation of 1.82, to a line average of 7.78 with a standard deviation of 1.41). This suggests that the refactoring process tends to choose longer forms of programs to effectively specialise the network towards programs of a certain length. This is an unexpected duality: as a result of corpus refactoring our network seems to train towards specialising to a certain length of program, while simultaneously generalising itself within that program length by increasing the ability for different lines to take more diverse operations. This said, this remains only a hypothesis, and the changes in Thiel inequality have not conclusively been shown to be the cause of the performance increases.

4.1.7 Conclusion

This experiment set gave a preliminary overview of the needs for a neural code synthesis system. The approach employed depends heavily on human involvement, both to provide a seed program and to define the requires for each corpus generated, which will need to be replaced with an automated system if the lessons learnt are to be deployed into a fully autonomous synthesis system.

The lessons learnt can be generalised to more than simply neural code synthesis systems, as they can inform the training of any code synthesis approach which relies on neural networks, including GP and solver-based approaches which employ neural components.

The following conclusions can be drawn and applied to future experiments:

Guided training corpus generation, based around requirements, is critical. It can be seen that neural networks have superior performance if given a targetted training set, as opposed to a much broader one generated by uniformly sampling program space. Due to the vast sizes involved, only attempting to encode a subset of the program space may be more computationally tractable than attempting to embed the entire space into a network's weights. If portions of program space are never relevant for problems the system will encounter, it could well be very useful to exclude these.

Heuristically pushing towards more complex programs has a small empirically demonstrated utility, but only to a minor degree. This fitness function in the training corpus generation was shown to have use, but its effects were minor enough that it may be ignored without compromising performance excessively. This would both simplify the system and perhaps allow it a greater degree of generality, as it would not be targetting any particular program size.

Seed programs are necessary for creating training corpora. In this experiment's setup, it was shown to be necessary to include human-designed exemplars of the requirements in a working program, to allow the genetic process to generate variants. While seed programs are necessary, automating their generation would be ideal.

Binary encoding schemes for numeric values are superior to real-valued scalars in terms of neural network performance. This may be due to the changes in which pieces of information are readily available within the encoding, with a core piece of information being "is this number even?". This encoding format can be used for all future experiments as it is broadly applicable.

4.2 Experiment 2: Discriminator-Based Corpus Generation

This second experiment attempts to remove the human element, replacing it with a mechanism to automatically identify which types of programs should be included in a neural network's training corpus.

This is accomplished by training a 'discriminator' neural network which attempts to recognise key characteristics which separate human useful program behaviours from less useful ones. It seeks to classify I/O examples into two classes, one which is useful to learn from, and one which is not, without ever seeing the source code. Features which could be selected for are 'interesting' patterns such as alternating values or consistent arithmetic transformations of the input values. Features which could be selected against are extreme values, far beyond the magnitudes of the values being input into the functions, or functions which ignore any inputs and simply consistently return a patternless collection of values.

This is based on the concept of Generative Adversarial Networks (Goodfellow et al. (2014)), although those classically are two neural networks, while this employs a Genetic Algorithm instead.

4.2.1 The Genetic Programming Implementation

The Genetic Programming Algorithm employed is a conventional linear GP implementation. It operates by producing an initial generation of programs by mutating an empty program. It then iterates through generations by evaluating each program's output compared to the desired one, then using tournament selection to select programs to use as parents for the next generation. Mutation of children occurs by either altering, deleting, inserting lines, or by inserting entire blocks from previous successes.

4.2.2 Initial Setup

The problem faced by each particular run of the GP is presented in the form of a set of input-output examples. Specific number of examples is up to the user, with a higher quantity of examples reducing the risk that the produced solution would not generalise in the way the user desires. An example number, to give a sense of scale, is 10 inputs followed by 10 outputs. The initial population is produced by mutating the empty program (as described below). The empty program is first added, then mutants produced until the generation reaches the desired generation size. There is no requirement that any of these programs be mutually dissimilar in any way, so new instances of the empty program may be added, e.g. if the only mutations applied were the delete-line mutation.

4.2.3 Fitness Function

The Fitness Function used to evaluate each member of the population is the evaluation of the program's performance in terms of output similarity to the target behaviour. Each program in the current generation is executed with each input provided by the user. It generates an output, and this can be compared to the example output. The first case is that the dimensions of the arrays do not match. This applies to both the 1D and 2D arrays. In the event that any of the returned outputs do not match the desired output dimensions the fitness is set to a very negative PENALTY value. This PENALTY value is fixed at -10,000.

If the arrays can be compared element-wise, due to matching dimensions, the fitness value is the negative of the number of mismatches. For every element in the programs' output which is not the same value as the corresponding element in the example the fitness decreases by 1. As a result, 0 is considered a perfect score, and all other scores are negative.

This means that the difference between them is binary, either they match or do not. While the function could be changed to a Euclidean distance metric, whereby values which were closer to the target output were penalised to a lesser degree, this was deemed undesirable. Initial small scale testing indicated this produced a lower rate of success. It is possible that the landscape it generated affected the GP behaviour during symbolic operations, such as re-ordering functions (sorting, reversing...). Specifically, it could lead it to 'approximating' a re-ordering operation by applying an arithmetic function to each element in the input array, moving the values closer numerically to the target output, but in a way which prevented it from discovering the true solution.

4.2.4 Parent Selection and Crossover

4.2.4.1 Elitism

Elitism is the genetic algorithm technique of retaining a number of the highest-fitness population members from the previous generation Baluja and Caruana (1995). They are added unchanged to the newly forming generation. This work retains the single highest-fitness population member.

4.2.4.2 Parent Selection

Parent selection is performed by tournament selection Miller and Goldberg (1995). This process occurs by selecting a single member of the previous generation at random (with no biasing) and stored as the *chosen_parent*. For 20 subsequent iterations, a random other is chosen, and if its fitness is higher than the current *chosen_parent* it takes its place and becomes the *chosen_parent*.

This parent selection strategy is scale-invariant. Only comparators are employed, so a minor difference in fitness is equal to a large one. This allows minor differences in fitness, despite being much smaller in absolute size than the fitness values themselves, to drive major change. Small improvements are still able to propagate themselves well throughout the population, while retaining a stochastic component which can preserve lower-fitness members to maintain population diversity.

Two parents are selected for each child program to be added to the new generation. These two parents are selected independently by tournament selection, and can be the same program.

4.2.4.3 Crossover

Each program is of fixed length, and crossover initially occurs using the fairly simple process of taking the first half of the first parent program's source code and concatenating the second half of the second program's source code.

The child program is evaluated for ENDBLOCK consistency, by counting the number of operators requiring an ENDBLOCK and the number of ENDBLOCKs present. While there are too few, the lowest non-ENDBLOCK line is overwritten to become an ENDBLOCK operator. While more are present than are required, the lowest ENDBLOCK by line-number is set to NO-OPERATION. This forces flow-control operators to match correctly with an ENDBLOCK, even if the crossover process caused a mismatch, but runs the risk of overwriting relevant code.

This process is in addition to previously described ENDBLOCK problem tolerance, and may render it inapplicable in practice. The ENDBLOCK tolerance was primarily introduced for Experiment 1, and maintained for consistency's sake.

4.2.5 Mutation

When a child program is produced for the new generation, there is a probability it will undergo mutation. This probability can be set by the GP designer as they deem appropriate for the problem they are facing, but defaults to 0.3.

If a mutation occurs, the mutator function selects how many changes to make. Multiple changes are permitted to allow the possibility of complex new additions to the program taking part, allowing mutations which require two or more lines to be added to succeed (e.g. if either line individually would decrease fitness).

The mutator is guaranteed to make a single change, and then iteratively either makes another change or terminates the mutation process with an 0.5 probability for either option. A maximum of 9 mutations are permitted. The mutations permitted are detailed below.

The program is considered for the purposes of mutation to be only the lines with an operator on them, that is to say removing all NO-OPERATION lines. To bring the program back up to requisite length the program is appended with NO-OPERATION lines after mutation is completed.

4.2.5.1 Insert

Inserts a line of code into a randomly chosen position in the program. This line is formed by uniformly randomly selecting from a list of all functionally distinct lines permitted in the language (regardless of current context). That is to say every operator is iterated through, and every permutation of every variable it can take (since all variables are effectively always declared, every potential variable is always present) is added to the option set. For example if 12 variables are permitted, and the operator takes 1 variable input and writes to 1 other, 144 options are available based on that operator.

If this new line requires a closing ENDBLOCK, an ENDBLOCK is inserted into the code at a randomly chosen point anywhere later in the program than the previously inserted line.

4.2.5.2 Delete

The delete mutator simply removes a line, chosen at random.

If this caused the program to have more ENDBLOCK operators than flow-control operators requiring one, the last one in the program is deleted.

4.2.5.3 Single-Point

This mutation operation allows single values to be changed in lines. Each line takes the form of an operator and 4 parameter integers, indicating which variables to read and/or write to (some of which may be irrelevant to the operator at hand, if it takes fewer than 4 arguments).

The mutator first decides a maximum mutation count, the maximum number of these to alter. It always allows 1 alteration, then iteratively either adds 1 to this or terminates, with 0.5 probability for either outcome. As such it has a probability of 1 of altering at least one point in the line, 0.5 to alter at least two, 0.25 to alter at least three, and so on until the maximum of 5 alterations.

It then selects uniformly from the set of all line options (as detailed in the insert mutator). Any change to the operator or to a parameter counts as a single change, so a full line replacement would be 5 changes, while simply swapping one variable reference to another would be 1. The new line must have at most the maximum mutation count as decided above.

This means that a line can be altered in functionality in a precise way, keeping large amounts intact. An addition operation can be replaced by a subtraction one, with the variables written to and read from being maintained.

4.2.5.4 Selection of mutation

The mutations are selected in a weighted fashion, with the mutation types insert, point-mutate and fragment-insertion having a weighting of 1, and delete having a weighting of 2. They are selected based on this weighting, then applied as described above.

Multiple mutations can occur during the same child creation process, and the type of mutation is chosen independently each time, so any mixture of mutation operations may occur.

4.2.6 Methodology

The overall approach to code synthesis is to feed in a set of ten input/output (I/O) examples to a neural network, and have the output layers of that neural network select the most likely operation for each line of code for a function of a given upper length limit (where the number of output layers is equivalent to the maximum number of lines of code in the function, each neuron in an output layer is equivalent to selecting one particular operator for that line, and each line of code can be set to 'no-op'). The probabilistic nature of a neural network's output allows us to gain a set of possible programs to search through up to a given search depth; this basic approach is similar in spirit to DeepCoder Balog et al. (2017) in that it is attempting to guess the probability of each 'feature' of a program – except in this case these features are relatively low-level instructions.

Neural networks for code synthesis are usually trained on a uniformly sampled set of programs from the total space of all possible programs; while this is viable for simple domain-specific languages, it becomes intractable for more general purpose languages. With the aim of achieving code synthesis using a much more general programming language, this experiment's methodology focuses on how training data can be effectively drawn from a much larger search space without human input.

The approach to this uses a novel hybrid solution inspired by genetic programming but using a neural network as a fitness function. This work terms this neural network a *discriminator*, as it attempts to discriminate between the algorithms the genetic programming element is currently generating, and the likely features needed by requested I/O examples for human-useful functions.

4.2.6.1 Language Used

The language is expanded, thus increasing problem complexity considered by implementing an operation which instantiates empty arrays of given length. It also implements operations to allow array length to be determined, and the ability for a function to call itself, to allow recursive functions. This experiment's methodology also avoids using any human-provided hints about the likely features of source code for problem solutions, instead relying only on a set of I/O examples for unsolved problems.

The full set of language operators available and restrictions used is seen in table 4.3.

Table 4.3: The 16 operators available in implementation of language, with the inputs to the functions and their purpose

Operator	**Input**
Assign from Variable to Array	Array: Write Target; Int: Array Index; Int: Read Target
Assign from Variable to Array	Array: Read Target; Int: Array Index; Int: Write Target
Create Array	Array: Array to initialise/overwrite; Int: Length of Array
Get Array Length	Array: Array to read; Int: Write Target
Variable to Literal	Int: Write Target; Literal: Can be -1, 0 or 1
Arithmetic Add	Int: Write Target; Int: Read 1; Int: Read 2
Arithmetic Subtract	Int: Write Target; Int: Read 1; Int: Read 2
Arithmetic Multiply	Int: Write Target; Int: Read 1; Int: Read 2
Arithmetic Divide	Int: Write Target; Int: Read 1; Int: Read 2
Arithmetic Modulo	Int: Write Target; Int: Read 1; Int: Read 2
Assign Variable to other Variable	Int: Write Target; Int: Read 1
Flow: Loop	Int: Iterator; Int: Loop to
Flow: Condition: Greater than 0	Int: Variable to test
Flow: Condition: Int equality	Int: Variable to test 1; Int: Variable to test 2
Recursive Function Call	Array: Array to set to function return; Array: Input 1; Int: Input 2
No-Operation	No Inputs

4.2.6.2 Human useful corpus

This experiment's human useful corpus is a set of I/O mappings for 40 unique functions, each of which takes one array input parameter (of any length) and one variable, and returns an array of any length. The set of problems includes reversing arrays, appending arrays with new values, and summing the values in an array (a full list is given in appendix A.1). It is assumed that these I/O mappings have been requested by human users, but that none have yet been solved. It is also assumed that each function has been requested at least five times, with different I/O mappings for each. This gives a total corpus of 200 I/O examples as a guide to the kinds of input-to-output transformations that are considered 'useful'. As opposed to the previous experiment, a key difference is that at no point is the system ever provided with any source code for these examples.

During early experimentation it was found to be beneficial to apply some conventions around these I/O examples, specifically for the first three examples in the set of ten for a given problem. The first I/O example for any problem is such that the content of each input array cell is the index of that cell, starting from 0, and the input variable has a random value. The second I/O example has the same properties but the second array cell is randomised to a new value between -8 and 8, inclusive. The third I/O example for every problem is the same as the first example except that the input variable (only the stand alone integer, not the array) is re-randomised to the same range. The remaining seven I/O examples can be anything, and are randomly generated in the corpus. The logic behind this, was that it would allow the system to see how changing part of the input, but not other elements, affected the output.

The synthesis pipeline has the challenge of starting from this corpus of unsolved problems, specified by I/O examples, and solving as many of them as possible by synthesising the correct source code which correctly maps the given input to the given output for each problem.

4.2.6.3 Neural network architecture and search process

Two neural network architectures are used in the synthesis pipeline, a synthesiser and a discriminator. The discriminator is discussed later, and is used in generating corpora on which the synthesiser is trained. The synthesiser network is that which receives I/O examples and attempts to build a source code program to match the required functionality.

This neural network has an input layer which accepts both the input and output values of each pair of 10 examples for a given problem, such that each input neuron takes a single bit of each integer. Internally a set of 8 layers of 256 neurons were used, each connected to all previous layers, with selu activation, a simplified version of the net used in Zohar and Wolf (2018b). For output, the network has one output layer per line of the program to synthesise. Each output layer has one neuron for each way in which a line can be written (all valid operations), including no-op. A labelled program would be represented as an array of one-hot vectors, with the non-zero value mapping to the way that particular line should be written. Crossentropy training loss is used on each line.

When reading a program out, the activities of all output layers' neurons are taken and ranked, giving a confidence for each option for each line. A search can be performed over the top 1,000,000 programs the network returns as 'most confident', using a beam search technique.

To perform the beam search, each line is given a "depth" to search, ranging from a single option to up to a maximum of 10 options. The combination of all programs within this space are initially added to a set, which therefore has a size equal to the product of all the depths. The beam search constructs the initial search volume by iteratively increasing the search depth of a single line to maximise the exploration value function. This function is as follows:

$[a_0, a_1, a_2...a_n]$ represents the ranked option confidences from the neural network, now sorted such that a_0 is the highest-confidence option. These are normalised by dividing by a_0.

S is the search volume size, and S'_i is the search volume size if line i had its depth increased by one.

D_i is the current depth of search of a given line minus one (for example if D_i is 0, only the first option will be added to the search volume, if it is 1, the top two ranked options will be added to the combinational set of programs (thus doubling the search space size)).

The exploration value for adding increasing the depth of any given line i is $\frac{(a_{D_{i+1}})*0.75^i}{S'_i - S}$

This process attempts to drive the neural network towards exploring lines in which multiple options are highly confident, and away from lines where a single option has been given a high confidence and all others given a low or negative confidence.

Once the iterative addition has produced a set of $>= 1,000,000$, the 1,000,000 options with the highest sum confidence are selected and searched exhaustively.

4.2.6.4 Initial Sub-Corpus Generation

The synthesiser neural network requires a training corpus comprised of the source code of example programs together with the I/O pairs for each program. Based on this training it is then able to solve some of the problems in the set of human useful I/O examples.

To generate this training corpus an iterative process was used, alternating between genetic programming and discriminator training to create a series of increasingly relevant corpora. At the start of this process, an initial corpus of 1,000 functionally unique programs were sampled at random from the total space of all possible programs (where a program is considered unique if at least one output value is different from any other program when given the same five randomly-generated input parameters).

Corpora other than this starting corpus are created by using a parent corpus from the set of accepted corpora (creation process detailed below). A child corpus is accepted if it finds an implementation for a human-useful I/O mapping which was not found by an existing accepted corpus, otherwise it is discarded and cannot be used as a parent.

When creating a new child corpus, a parent corpus is selected by roulette selection from all currently accepted potential parents, with each potential parent's weighting being $(0.1 + number\_of\_successful\_children)/(0.1 + number\_of\_children)$.

4.2.6.5 Discriminator training and usage in sub-corpus inheritance

After the first corpus, further corpora are generated based on the use of the discriminator. This is a neural network designed to classify input/output pairs generated by programs in the generated corpus as being *closer to / further away from*, those of I/O pairs in the human-useful set from users.

A new corpus is created by selecting programs from a parent corpus which are measured to be most similar to human-useful programs in their input/output mappings (specifically the form of their outputs, and how these outputs seem to relate to corresponding inputs). By doing this, it can be hypothesised that the kinds of source code features found in these programs will similarly move closer to those needed to synthesise programs solving the human-useful I/O examples.

The discriminator neural network then works as follows. Architecturally, it consists of 2 dense layers of 16 nodes, with a single output. The featurisation of programs is identical to the synthesis neural network (as described above). This network is trained by providing all of the I/O examples for all unfound human-useful target programs, and generate I/O examples for each of the 1,000 programs in the parent corpus. The discriminator network is trained as a classifier to determine which of the I/O examples are from the human-useful set, and which are from the generated set. This training continues until a threshold T_k is reached. T_k is the proportion of programs which would be retained by the discriminator (as described below), if run on the parent corpus. T_k is a randomly set value between 0.1 and 1, set as $max(0.1, r^2)$ where r is a random real value uniformly distributed between 0 and 1. A random value is used here to increase the diversity in corpora formed, some being highly similar to the parent and some being fairly different, in a bid to maximise coverage.

This trained discriminator therefore returns an estimate F_d for how *human-like* a program is, ranging between 0 and 1. A program is said to pass the discriminator if it has an estimate of $F_d > 0.1$. This second threshold was chosen based on preliminary experiments, particularly based on analysis of the distribution of estimates, which was found to be highly biased towards either end of the spectrum.

These selected programs form the basis of a new child corpus. This child corpus is then expanded to have 1,000 functionally unique programs of its own, by selecting one of the existing programs in the corpus (using roulette wheel selection) and mutate it, then accepting or rejecting that mutated program as an additional member of the corpus based on a fitness function F_q. This value F_q is simply how much it exceeded the discrimination threshold $(F_d - 0.1)$. During development, roulette selection was found to produce far superior results than tournament selection if the discriminator values are offset by 0.1, due to bias away from programs which only just passed the threshold.

This process is iterated to create new child corpora with a desired total number of programs. Unlike the previous experiment, which involved seed programs, the system in this experiment employs only the I/O examples that users have requested to be generated. After multiple rounds of generating self-training data in the above fashion, to reach source code features that are increasingly likely to be involved in solving the requested I/O examples, it can then begin to be able to successfully synthesise solutions to the I/O examples of programs requested by users.

4.2.6.6 Evaluation of overall system

The synthesis pipeline returns a collection of solutions to I/O problems in the human useful set, and also returns a set of generated training corpora which were used in finding these solutions. All generated corpora are kept, and their corresponding trained synthesiser networks, which are 'kept' in the above iterative process as either a parent or final child. Note that the intermediate parents are kept along the way because a child corpus is sometimes unable to solve some of the problems of its parent, even though it can solve new problems that its parent could not.

These trained neural networks can then be re-used when given new I/O problems not present in the initial human useful set; each individual trained synthesiser network processes an I/O problem in ~ 0.25 seconds. An alternate approach, also evaluated, is to collate all corpora into a single one to serve as a training corpus for a network.

4.2.7 Results

This evaluation compares the approach to training corpus generation against competitive baselines. For all experiments a total of 200 sub-corpora were generated, using this experiment's approach, with each part of the experiment repeated 20 times. The HU corpus was fixed for all experiments, and the I/O examples did not vary, including

Approach	**I/O examples solved**	**Unique functionalities**
Discriminated sub-corpora	$\mathbf{81.5}(\sigma = \mathbf{8.28})$	$20.3(\sigma = 2.08)$
Random sub-corpora	$43.5(\sigma = 4.29)$	$11.8(\sigma = 1.20)$
Genetic Programming	$64.6(\sigma = 3.37)$	$\mathbf{22.5}(\sigma = \mathbf{1.86})$

Table 4.4: Success rates for the human-useful problem set by approach type. The set consists of 200 I/O examples, derived from 40 ground-truth programs

in baselines, to allow consistent testing and repeatability. Noise between repeated experiments therefore derives only from the different systems' internal stochasticity.

To evaluate the system it was compared against two baselines: one without using the discriminator, instead using randomly-generated corpora, and one using genetic programming on the same problem set. It was then compared against one using uniformly-sampled training data, as is used in related research for simpler domain-specific languages, and lastly this section explores the the effects of decisions made by the discriminator in more detail.

4.2.7.1 Performance compared to baselines

This part of the experiment compares the find rates of programs against two baselines. For the first baseline the system was identical other than that the discriminator removed as a fitness function, such that successive training corpora are generated only using randomly selected parents and mutations. The remainder of the pipeline is kept the same for this baseline.

For the second baseline, a genetic programming is used due to the inability to re-use baselines from other work in the literature, as for example seen in the DeepCoder framework, caused by the relative generality of our programming language for synthesis. The genetic programming technique was designed to require roughly the same total computational time as the full sub-corpus generation pass, to fairly compare the options for user I/O mapping resolution. It therefore uses a population size of 1,000, and a maximum generation count of 10,000, which approximately leads to the same time cost. It uses tournament selection, with a tournament size of 6 and a probability of mutation of 0.15, which were found to be optimal in preliminary testing. The fitness function is a function of the Euclidean distance between the desired output and the target output, unless the outputs differ in length, in which case it is set to an extremely negative penalty value. It is noted that this is a common theme across synthesis research (including solvers) which use various specialist baselines (Balog et al. (2017), Sun et al. (2018b), and Devlin et al. (2017b)).

The results are shown in Table 4.4, demonstrating that the set of networks

Approach	I/O examples solved	Unique functionalities
Collated discriminated corpus	$\mathbf{44.5}(\sigma = \mathbf{7.5})$	$\mathbf{11.3}(\sigma = \mathbf{2})$
Random corpus	$16.1(\sigma = 2.5)$	$3.85(\sigma = 0.59)$

Table 4.5: Success rates for the human-useful problem set by approach type. The set consists of 200 I/O examples, derived from 40 ground-truth programs

produced by discriminated sub-corpus generation between them produced the highest resolution rate for the 200 human-useful I/O mappings, returning an average of $81.5(\sigma = 8.3)$.

Interestingly, the genetic programming technique found more unique functionalities. Of the 40 unique ground truth programs, the GP baseline found 22.7, while the sub-corpora found 20.3. This means that while the sub-corpus technique had a higher probability of finding a program to match a user-supplied I/O mapping, there existed a stronger inequality between the find rates than the GP algorithm, which was more consistent in its probability of finding any given program.

It is highly likely that this is a phenomenon produced by the nature of the discriminator itself, driving the system towards producing certain types of algorithms predominantly. It seems likely that certain properties of certain programs would be more recognisable by a neural network, and so emphasised by the discriminator's fitness function; this is a key avenue of future work.

4.2.7.2 Comparison with uniformly generated training data

This set of experiments evaluates the performance of our iteratively-produced training corpus against a baseline randomly-generated training corpus. To do this we take a subset of 5 of the sub-corpora produced, attempting to maximise the number of separate I/O examples found by the set of networks. The collated corpus then discards any duplicated functionalities, giving a training corpus of approximately 4,200 examples. For each of these, we write out a set of training examples, with 5 randomly generated inputs into each function being used to generate the feature I/O examples. We then train the same synthesis neural network on these as before.

For the comparative baseline training data we simply sample 5,000 programs uniformly at random from the space of all possible programs, and again generate 5 training I/O examples using each of these programs. In both cases the corpora are then divided between training and validation in a 0.9:0.1 split, with validation programs being functionally distinct from those in the training set; the training data is then fed into our synthesis neural network and tested.

As can be seen in Table 4.2.7.2, the collated corpus produced by the discriminated sub-corpus generation process more than doubles the performance of the randomly generated training corpus. The discriminator has driven program generation towards a set of programs which is far more representative of the types of behaviour present within the human-useful programs.

Inspection of the programs within the corpora matches this expectation: there is no drive for a randomly generated program to, say, contain a loop, and if they do there is no drive towards writing to all/most cells in the output array. While the behaviour of each training program is thus distinct, this is not necessarily in a 'useful' way, relative to the types of problems the user wishes to solve.

The discriminator, however, can drive towards useful training programs, by emphasising programs which write relatively uniformly to output arrays, using loops and with values remaining within sensible ranges; as well as generating programs with outputs dependent on the inputs, rather than fixed-output programs. Without ever receiving training labels or information about the nature of useful features, the discriminator learns to maximise their presence in the sampled programs.

4.2.7.3 Analysis of effects of iterated discriminator use

This section explores in more detail the effect of the discriminator design. We first demonstrate the difference-over-time from our first baseline in Sec. 4.2.7.1, in which we generate successive corpora using either our discriminator or using simple random selection of parent. The results of this are shown in Figure 4.6, in which we see how many programs from our target human useful set are solved over time as successive corpora are generated. As expected, both the discriminator setup and the random setup start at the same point, as the initial corpus does not use a discriminator. The first discriminated sub-corpus trains a network which finds an additional 6.7 solutions to I/O examples on average, compared to the non-discriminated corpus' increase of only 2.8. This progress continues as new sub-corpora are added, and is also seen on the graph of unique functionalities found. This clearly demonstrates the value of the discriminator in corpus generation.

This experiment next examine the find rates over the 'ancestry' of sub-corpora in detail, to give indications as to the behaviour of the system in response to the discriminator's iterated use. Each sub-corpus past the first uses another as a parent; the ancestry of a sub-corpus is therefore the number of parents since the starting corpus. This varied by experiment, with all accepting a corpus with ancestry of at least 5, and the maximum being a single run which had a sub-corpus of ancestry 12.

Figure 4.7 plots the find rates of each program over ancestry progression. It is discovered that certain programs are trivial to find, and do not require discriminator use (Array Length; Array to Zero...). These have a find rate of nearly 1.0 before the discriminator is used (black in first cell).

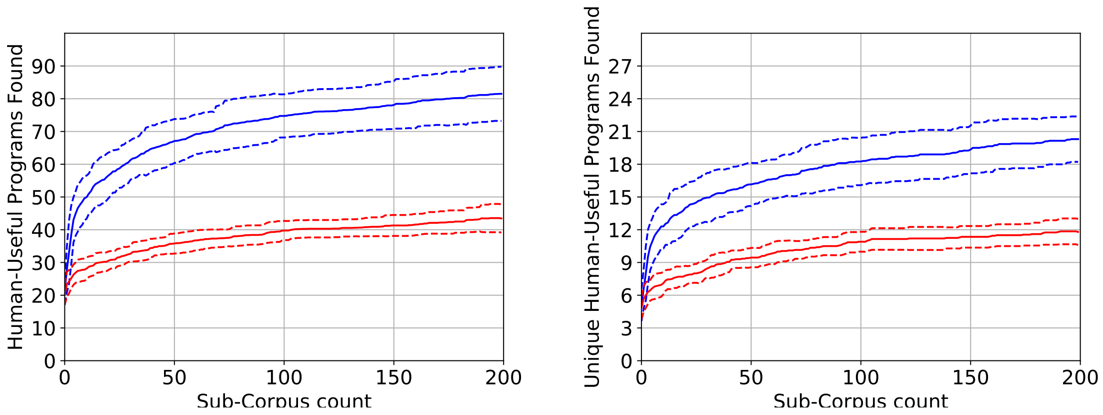


Figure 4.6: Success rates of programs satisfying I/O specifications, over sub-corpus count, for both this experiment's approach (blue, higher) and random baseline (red, lower). Success rates for all 200 I/O specifications on left, find rates for unique functionalities on right. First standard deviation shown.

Certain programs are found with high reliability past ancestry of 1, for example the identity program. This program was almost never found without use of the discriminator, but a single use lead to it having a nearly 1.0 find rate. This indicates that the discriminator lead to a set of sub-corpora which represented the identity function's programmatic behaviour far better. It is found that these sub-corpora's programs nearly universally featured loops and sequential array write operations, functionalities required to produce the identity function.

The second use of the discriminator showed similar programs, such as Identity Parity and 'Iterate from Start'. These were rarely found in both a non-discriminated sub-corpus, or an $ancestry = 1$ sub-corpus, but featured regularly at later depths. This reflects the discriminator iterating on its previous selections, attempting to discriminate between programs produced by a first-generation discriminator and the human useful corpus. These programs now found feature more complex loop-using behaviours than simple reproduction of the input array, such as using conditionals and literals.

Past $ancestry = 2$, however, no further sudden find-rate jumps are seen. This indicates that the discriminator loses its effectiveness and can no longer guide as reliably towards the functionalities of the human-useful corpus. It could be speculated that the discriminator is unable to force the presence of the required functionalities in the produced training corpus either because (i) the genetic algorithm doesn't produce any for it to select; (ii) it does not have the capacity to represent the behaviours (due to low layer width or depth); (iii) these functionalities are simply not identifiable from I/O mappings alone.

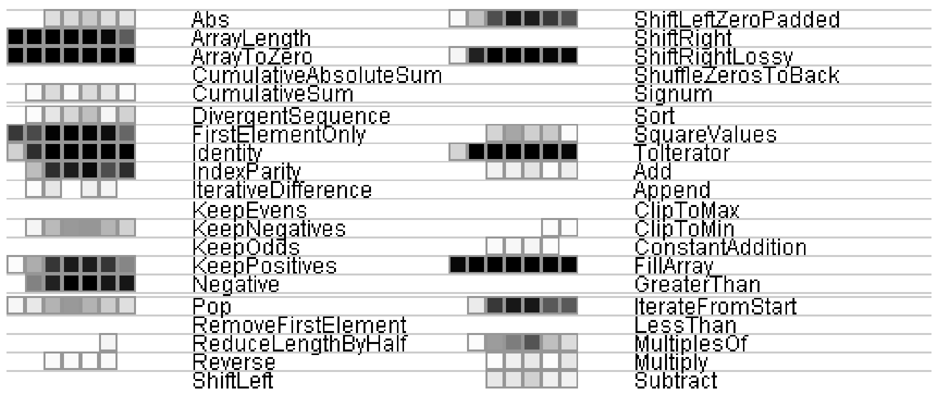


Figure 4.7: Program find rates over sub-corpus ancestry, with each cell representing the success rate of an attempt by 20 NN to solve all 200 IO examples. From white at 0 success rate, to fully black indicating 1.0 success rate. Border used to indicate a non-zero value. Plot cut off at *ancestry* = 8 due to low sample size past this point. The way actual program outputs evolve is detailed in subsection 4.2.7.4.

4.2.7.4 Generated program examples

Table 4.6 shows outputs from 15 randomly generated programs, from 3 randomly chosen sub-corpora. The first is the starting corpus of the run, which had no discriminator. The second has a discriminator trained between the human-useful I/O examples and its parent corpus. The third sub-corpus then has a second generation discriminator, which was trained based on a discriminated corpus and the HU I/O corpus.

All outputs are responses to the function being run with an input of input_array = [0,1,2,3,4,5,6,7] and input_integer = 2.

It is seen that the programs in the starting corpus, ancestry=0, which were randomly sampled from program space, differ greatly from the style of program this experiment is attempting to train the network to synthesise. The majority of all returned values are 0, and the array length varies considerably. There is little evidence that the input array is being read in, or indeed any use of loops at all.

The second generation corpus, ancestry=1, shows little use of the input values, but has outputs in more consistent ranges. The output lengths now appear to always be the length of the input array, and the programs clearly use loops to write to the output array. Despite this, the output patterns are highly uniform, often being the same value repeated for most of the output array.

The third generation corpus, ancestry=2, shows more complex program still. Negative values, which would require arithmetic operations to produce, are present. The values vary across larger ranges, and they show elements of the input array (the

Table 4.6: Examples of outputs of generated programs, randomly chosen from 3 sub-corpora. These sub-corpora are organised based on ancestry, how many parent-child relationships exist between the first sub-corpus (ancestry=0).

Sub-corpus ancestry	**Program Output**
Ancestry=0	[0]
Ancestry=0	[0,0,0,0,0,0,0,0]
Ancestry=0	[0,0,0,0,0,8,0,0]
Ancestry=0	[2]
Ancestry=0	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
Ancestry=1	[7,7,7,8,7,7,7,7]
Ancestry=1	[0,2,2,2,2,2,2,0]
Ancestry=1	[7,8,8,8,8,8,8,8]
Ancestry=1	[8,8,8,2,8,8,8,8]
Ancestry=1	[2,2,2,2,2,2,0,0]
Ancestry=2	[0,8,16,24,32,40,48,56]
Ancestry=2	[-1,-7,-6,-5,-4,-3,-2,-1]
Ancestry=2	[0,8,16,24,32,40,48,56]
Ancestry=2	[0,-7,-7,-7,-7,-7,-7,-7]
Ancestry=2	[-2,1,2,3,4,5,6,7]

last example being the input array changed by a single element).

This is believed to be fairly illustrative of the behaviour of the discriminator, although more in-depth analysis of the programs produced could form a good direction for future work.

4.2.8 Conclusion from Experiment 2

This experiment has shown a number of things, some surprising, which will be built on in future experiments.

4.2.8.1 Performance of Genetic Programming

One major discovery was that despite its success against a more classic neural baseline, this approach did not strictly out-perform a genetic programming (GP) approach. While its find rate was higher, certain programs were not found by the neural approach but were found by the GP.

This suggests that the search approach, beam search guided by the neural network, lacked certain key advantages which GP possessed. This lead to future experiments being run with a neural network guiding a GP process, to leverage the power of the GP.

4.2.8.2 Success of guided corpus

It is clearly seen that a neural network is better trained on programs which are representative of the target domain, as opposed to a uniformly selected subset of programs within the space of all possible programs.

Certainly a targetted training corpus is beneficial for neural network training success, but the process employed in this experiment is time-consuming. It is possible a faster approach could be designed, especially one based on use of found solutions to human-useful programs as seeds to corpus generation.

4.2.8.3 High I/O example count required to avoid neural network overfitting

During preliminary testing, it was shown that more I/O examples were needed to successfully train the discriminator than can reliably be expected to be produced. The I/O examples were also structured in terms of which input values were employed in a portion of the example set, which may be possible for a human-user to provide but limits the system's flexibility. Ideally, the system would not have these limitations on its I/O examples, and a more general-purpose approach should attempt to operate on a more flexible and smaller I/O example set.

4.2.8.4 Automatic Refactoring

While successful in this experiment, attempts in future experiments to recreate the success of automatic refactoring were found to be unsuccessful. Initial experiments with the full language and increase line counts employed in future experiments failed to replicate the automatic refactoring performance gains. This, coupled with the awkwardness of using the system in future experiments, as it would require a full neural code synthesis system to be employed each time, lead to this discovery not contributing to future work.

Chapter 5

Experiment 3: Guiding Genetic Programming

This section addresses the question of how a genetic programming algorithm can be guided, and whether a deep neural network can then recognise when to deploy this guidance. First it introduces the genetic programming algorithm (GP) which will be used for the rest of this work, then discusses the first experimental direction taken, that of guiding a GP process.

5.0.1 The Genetic Programming Implementation

The Genetic Programming algorithm employed is altered for this and for the later experiments, to improve its performance and to allow it to handle a new data type.

The Fitness Function used to evaluate each member of the population is the product of two components. The first is the evaluation of the program's performance in terms of output similarity to the target behaviour with an additional very small (orders of magnitude lower than the values relevant for the other two factors) penalty for each line used, to drive search for shorter programs if all other factors are equal. The second is a repetition factor, used to penalise programs which resemble already-seen programs, to drive the population away from stagnant areas of the fitness landscape.

Specifically, the fitness is $(error + per\_line\_penalty) * repetition\_penalty$. Note here that both 'error' and 'per_line_penalty' are either negative or zero, so fitness is negative. The algorithm is still attempting to maximise fitness, and has 0 fitness as its maximum possible outcome. A multiplicative function was used, rather than an additive penalty to maintain the behaviour and impact of the penalty regardless of the absolute magnitude of the fitness produced by the classic fitness function. This was needed due to the changes the fitness saw, with high errors initially as the programs generated were far from the target solution to far lower ones towards the end of the GP's successful runs.

5.0.1.1 Behaviour-Based Error

Two types of output can be synthesised by this framework, 1D integer arrays and 2D boolean (implemented as integers but limited to only 0 and 1). For the first, the system works the same as in Subsection4.2.1, for the second, no major difference is required conceptually, only changing the implementation to handle 2D arrays. As before, each program in the current generation is executed with each input provided by the user. It generates an output, and this can be compared to the example output. The first case is that the dimensions of the arrays do not match. This applies to both the 1D and 2D arrays. In the event that any of the returned outputs do not match the desired output dimensions the fitness is set to a very negative PENALTY value. This PENALTY value is fixed at -10,000.

If the arrays can be compared element-wise, due to matching dimensions, the fitness value is the negative of the number of mismatches. For every element in the programs' output which is not the same value as the corresponding element in the example the fitness decreases by 1. As a result, 0 is considered a perfect score, and all other scores are negative.

For integers in the 1D arrays, this means that the difference between them is binary, either they match or do not, and this matches well to the boolean data types used by the 2D arrays (which can also be seen as 2D arrays of integers bounded to only contain values of 0 or 1).

5.0.1.2 Per-Line Penalty

A function was implemented to statically analyse a program in the language to determine which lines were able to contribute to the output. Specifically, the variable interdependencies were evaluated. An operation which writes to the returned values (either calling a write operation on the 2D array, or writing to the output array in the 1D domain) causes all its input variables to be flagged as 'contributing'. Any operation which could have been performed previously (not simply above it in the source code, if loops are present) which writes to a contributing variable can then have its input variables similarly flagged. This iterates until no further operations can be found which write to contributing variables. Any operation which does not can then be determined to be irrelevant to the program's outcome.

One edge case exists, in which operations, while non-contributing, caused the program to timeout, simply due to being performed and counting towards the termination counter. While technically these are contributing, as they alter the output of the function, they are still considered non-contributing.

The per-line penalty depends on this static code analysis. Each line which is flagged as contributing causes the fitness function to apply a $-(10^{-11})$ penalty to the program. This penalty is added onto the behaviour based fitness value, computed as described above.

This penalty is irrelevant when the programs differ in output error, as described above, due to multiple orders of magnitude difference. However, if the GP produces a zero-error program, it can continue to iterate, to reduce the lines used, attempting to minimise this value. It also serves as a tie-breaker for programs with equivalent or identical outputs.

5.0.1.3 Diversity Boosting Repetition Penalty

Diversity Boosting is implemented by using a diversity boosting repetition penalty multiplier, implemented as a ‘repulsion’ from previously attempted programs. Each generation the highest-fitness population member is determined, termed the elite.

Each elite has their code stored, after using the static code analysis to ablate any non-contributing lines. Each subsequent program has a penalty added based on how close they are to any previous elite, also after applying static code ablation. This ablation serves to ignore irrelevant code changes. The goal is to cause the programs' functionality to alter, mutations which only affect introns (the non-contributing lines of code) are irrelevant to this. The elite then serves as a ‘repulsor’, exerting a repulsive force on the GP population by decreasing the fitness of programs in the local region of program space (previous generations' repulsors are kept, and also used for the remainder of this GP search).

This repulsion penalty's calculation requires a code distance metric, to determine how similar two programs are to one another, termed D. D is calculated by examining each line in the stored elite program and calculating how far away (in lines of code) the same line is in the generation's program. Each repulsor adds $max(0, 1 - D/15)$ to the $repetition\_penalty$, such that multiple repulsors can exist in the same area of program space, leading to stronger avoidance of those areas.

To compute D we create a one-to-one mapping between the two functions' code lines, selecting the mapping which minimises the sum of per-line costs. The per-line cost is $offset{*}0.35$ plus 1 if the operators do not match, plus 0.2 per parameter which differs. Lines ‘inherit’ the offset of the previous line, so an inserted line only causes one line to gain the 0.35 offset penalty. Since all programs are padded with NO-OPS, this new line will be matched against one of these, and so will incur a penalty due to new operators and parameters.

This $repetition\_penalty$ is increased by 1, to bring its value to $>= 1$, being at 1 if there is no repulsion (no repulsors generated or none near enough to affect the sampled program), and growing as additional repulsors are created nearby.

This allows it to be multiplied to the previously computed sum of *error* and *per_line_penalty*. Since these are both always-negative, with 0 being the perfect score, multiplying it by a larger value will correspond to a larger fitness penalty. Note it is impossible for fitness to reach 0 as it would require a 0-line program which matches the target functionality perfectly.

For example, a program A, with lines:

```
MULTIPLY  VAR_2  VAR_1  VAR_0
  ADD     VAR_2  VAR_1  VAR_0
```

Would have a repulsion D of 1.0 to program B (seen below), as the operators on the first line has changed, despite the parameters having changed. If instead the first parameter were different, it would only have a repulsion of 0.2

```
DIVIDE  VAR_2  VAR_1  VAR_0
 ADD    VAR_2  VAR_1  VAR_0
```

Program A would have a repulsion penalty D of $0.35 + 1 + 0.2$ compared to C (seen below), as the new line incurs an offset penalty of 0.35, and the new line would be compared against a NO-OP line and receive a penalty of 1.0 for the operators not matching and a new parameter being used, for another penalty of 0.2.

```
CONDITIONAL  VAR_2
  DIVIDE     VAR_2  VAR_1  VAR_0
   ADD       VAR_2  VAR_1  VAR_0
```

5.0.2 Parent Selection, Crossover and Mutation

These all operate the same as in Section 4.2.1, without any need for changes.

5.0.3 Genetic Improvement

If during the search GP finds a program which matches the desired functionality, that is to say if the program produces the same outputs as the example when fed the example's inputs, it immediately begins Genetic Improvement.

This is done to improve the quality of the transferred material, as only a finite number of lines can be transferred between programs (quality here being the resultant gain in GP success rate). If too many are selected, they may simply exceed the space remaining in the recipient program, and be forced to overwrite functionality it was depending on. Longer fragments may also be less generalisable, and far more task specific, if the lessons learnt from full programs are applicable to program elements David and Kroening (2017).

The process simply requires adding additional permitted generations to the GP, thanks to the per-line penalty element of the fitness function. In the event of success by the GP, the fitness penalty from behavioural error drops to 0 for the population's highest fitness members, and as a result the fitness function is now equivalent to $-(contributing\_line\_count) * repetition\_penality$. Any program within the population which has behaviour not matching the desired behaviour will contribute its genetic diversity to offspring to a degree and rapidly be selected against in the tournament parent selection.

The GP is therefore now operating as a diversity boosting genetic improvement system. It continues to search for novel implementations without causing behavioural changes. The non-contributing lines, the introns, are preserved as the fitness function does not penalise them, allowing diversity to be maintained.

After the set number of additional generations has been iterated through, the highest-fitness program is returned. This program will have zero behavioural difference to the example behaviour, and the lowest line count the GP was able to produce.

If the GP did not find a matching behaviour, it does not perform this genetic improvement pass, and terminates immediately after the first GP generation count is reached.

5.1 Language Employed

To allow the remaining experiments run in this work to have a common footing the language used will remain the same from here on out, designed to have its source code synthesised by both a Genetic Programming algorithm and by a neural synthesis system. As before, features of the language are designed specifically to enable such processes to operate efficiently and without need for extensive pre or post-processing at any stage. It is full described in this section, but is based on the language used in the previous experiments.

5.1.1 Overview

The language takes the form of a matrix of integers, representing a program's instructions, and an interpreter able to take this matrix along with a set of input values and return the output of the program the matrix represents. Each line in the program, or row in the matrix, is 5 integers, the first representing the operation to perform, the subsequent 4 are parameters to pass into that operator.

When executing the program the interpreter sets the program's variables, which consist of a configurable number of integer variables, 1D integer arrays and 2D boolean arrays, to their default values. 2D boolean arrays are used to represent black and white pixel images. The input variables are then input by overwriting the variables, starting with the first. As such, if there are 3 integers fed into the program and it is configured to have access to 12 variables, the first 3 will be set to the values of the inputs, and the remaining 9 will be 0. Array inputs will also set integer variables to the length of the array being read in, as well as setting an array variable.

When executing operations, the interpreter starts at the first instruction, performs it and steps downward. Flow control operators exist in the form of conditionals and flow (employing ENDBLOCK operators to terminate their code blocks). These allow the interpreter's read position to be altered. The program terminates when either a set number of operations have been performed (which includes all actions the interpreter takes, therefore flow control including ENDBLOCK are counted), or when it performs the last operation and attempts to step outside the program matrix's row count (equivalent to having a forced END_PROGRAM operation at the end of all programs).

Values are then returned as specified by the program's configuration. The values returned are those of the last variable in the program, in the state they were when execution terminates. The type of variable returned can be configured, such that the program can be made to return its last variable, its last 1D integer array or its last 2D boolean array.

5.1.2 Operators available in the language

The list of all operators implemented in this language is detailed in Table 5.1. During certain tests, certain operators are disabled, as detailed during the discussion for that experiment. The 2D array operators are only available to the language if the function takes or returns a 2D array, otherwise they are disabled. Despite the 2D array in principle being an array of Boolean values, it is implemented as integers, taking the values of either 0 or 1, to allow easy conversion between integer variables and 2D array values.

5.1.3 Tolerance to anomalies and consistency of behaviour

A core goal of this language's design was that of consistency of behaviour. Ideally every integer matrix of width 5 should be mapped to a programmatic output. We relax this requirement slightly, by guaranteeing that every sequence of correctly formulated line options can be executed and a value returned. Correctly formulated in this case is simply defined as having valid operators, then assigning its parameters values within their legal ranges (e.g. not referring to variables which do not exist, noting that variables are automatically declared and initialised).

As a result of the design decisions taken, any line which is valid at any point in any

Operator	**Input Output Types**
Assign Variable To Array	Array, Int : Void
Assign Variable From Array	Array, Int : Int
Make Array	Int : Array
Variable To Literal	Int, Int between {-1,0,1,2} : Int
Add	Int, Int : Int
Subtract	Int, Int : Int
Multiply	Int, Int : Int
Divide	Int, Int : Int
Modulo	Int, Int : Int
Assign Var from Var	Int, Int : Int
Loop	Int, Int
Conditional (var > 0)	Int
Conditional (var1 == var2)	Int, Int
Conditional (var1 > var2)	Int, Int
Create 2D Array	Int : Void
Get 2D Array Size	2D Array : Void
Var to XY Point from 2D Array	2D Array, Int, Int : Int
Set 2D Array to 0 at XY Point	2D Array, Int, Int : Void
Set 2D Array to 1 at XY Point	2D Array, Int, Int : Void
ELSE	
ENDBLOCK	
NO-OPERATION	

Table 5.1: Operators available in the custom language used in this work, along the the parameters they take (format is input types, followed by colon, followed by output type). Each line define which variable to read or write from by the integers following the operator symbol. As such if 'Add' operator requires an integer, the line must set the parameter to a value in the range [0..*number_of_integer_variables*]. The Loop operator will set the value of its iterator variable to 0 at the start of its loop, erasing any previous value it may have held.

program is valid at any other point in any other program. There is no dependency on previous or later lines to ensure syntactic correctness. As such, the set of valid lines is always the same, and can be determined simply by examining the set of operators and their associated input requirements (which data type variable can be fed into the operation, as defined by the integers coding for parameters following the operator in the matrix).

Special casing the language avoids requiring the neural network to avoid creating syntactically incorrect programs. They can simply have a set of output neurons, one coding for each line option, and select from that pool. This removes the requirement for the network to encode for the syntactic considerations we have already dealt with as language designers.

5.1.3.1 Flow control operator inconsistency

Flow control requires a set of special-case handling. Flow control operators may not match with ENDBLOCK operators, therefore there could be no indication of when a loop should flow back to the top of the loop's code block. ENDBLOCKs may also exist where no flow control operator does, leading to a block being closed without ever having been opened. In the event that too few ENDBLOCKs exist, the outermost flow control operators are considered to have their ENDBLOCKs missing, and each ENDBLOCK assigned in a one-to-one fashion with the flow-control operators, starting with the innermost. Once done, this produces certain failure cases. The handling for these cases is as follows:

- **Conditional with no ENDBLOCK** This case is simply handled by terminating the problem if the condition evaluates to false. This is equivalent to having the entire problem contained with the conditional's code block. If the conditional returns true, code execution can proceed as normal

- **ELSE with preceding conditional but no ENDBLOCK** This is similar to a conditional with no ENDBLOCK. If the conditional evaluated to false, the interpreter skips until it reaches the ELSE block, then the code following it is executed as normal. If the conditional is treated as true, the entire remaining program past the ELSE-flow-control-operator is considered to be part of the ELSE-block's code, and skipped

- **ELSE without preceding conditional** An ELSE operator without a conditional is treated the same as a NO-OPERATION operator, and skipped

- **Loop with no ENDBLOCK** This is equivalent to the loop being a conditional with the ENDBLOCK. The loop itself can be considered to be a conditional, passing if the loop's bounding variable is greater than zero. If so, the code contained within the loop executes once, and the program terminates. If not, the program terminates immediately. This was considered a simpler and more predictable outcome than having the end of the program sometimes act as an ENDBLOCK.

- **ENDBLOCK with no associated flow-control** This case is handled as if the line were blank, and the NO-OPERATION operator had been called. The ENDBLOCK is ignored other than counting as a step performed for program timeout purposes

5.1.3.2 Divide by zero

Two of our arithmetic operators, the division and the modulo operators, must have a special case handled, the divide/mod zero case. In this case, we simply set their returned value to 0. This is obviously not mathematically valid, but is tolerated in order to improve the reliability of the programs generated. Syntactically, and indeed internally, it is equivalent to having an automatic conditional block surrounding these two operators, which either calls them if the divisor is non-zero or sets the value to their return-target variable to 0 if it is.

5.1.3.3 Array index out of range

If an operator requests to either read or write from an array variable it must specify a variable to use as the index. That variable's value is then the index of the cell within the array to read from or write to. Naturally, this has a high chance of producing incidents where an out-of-range value is requested, such as referring to a negative index or one beyond the range of the array. In this circumstance, the interpreter simply ignores the instruction, and performs no action. This is equivalent to post-processing any generated program in order to surround all array access operations with conditions checking the index variable is in range.

This choice may cause the synthesis system to produce code which a human designer would not necessarily consider elegant, and may introduce a number of irrelevant instruction calls which are discarded. Despite this, it increases the number of implementations which successfully produce the same functional characteristics, and as such may simplify the task of code synthesis, boosting success rates. Code elegance and code efficiency, while desirable, are not immediately within the scope of this work, and work in other related fields, such as Genetic Improvement Petke et al. (2018) could allow found solutions to be improved after synthesis.

5.1.4 Experiment 3: Methodology

To evaluate how a GP process can be guided a set of programs were examined, drawn from two domains of program (to boost confidence in the generality of the approach). These domains are array-to-array integer manipulation tasks, as studied before, and a new domain of 2D image drawing programs, derived from the corpus used by So and Oh (2018), which take only an integer size and return a 2D canvas of required size of boolean values. For each program examined, a set of subsets of the lines of code of a human implementation of the given program were selected, and used to guide a GP process. This GP process then operated on a subset of program space, the subset of programs which contain the required lines of code. This was done to assess whether knowing a portion of the program could be productive, and could improve the performance of a GP process, despite the remaining lines being unknown.

Once the performance of this approach was studied, a second analysis was performed, to determine if a neural network could recognise the presence of these lines of code in a target program, based purely on examples of that program's behaviour.

Next a study was run in which human-selected code fragments are deployed into a GP based on the neural network predictions as to their usefulness. This test employed a common set of fragments across all problems, as opposed to the first stand-alone GP test which studied fragments extracted from the programs they were being deployed into.

Finally, an initial end-to-end system was studied, which combined these two elements into a single system. This architecture takes only problems, defined by input-output pair sets, and learns from its successes, extracting code fragments and redeploying them.

5.1.5 Operators of the Language Used

Tables 5.2 and 5.3 provide lists of all the operators used for the two corpora used in the language employed in this section's experiments. Language variants are improved when compared to the language used in the previous sections' experiments, in order to allow effective processing of the two corpora's specific problem domains.

Operator
Assign Variable To Array
Assign Variable From Array
Make Array
Variable To Literal
Add
Subtract
Multiply
Divide
Modulo
Assign Var from Var
Loop
Conditional (var $>$ 0)
Conditional (var1 $==$ var2)
Conditional (var1 $>$ var2)

Table 5.2: Operators available for GP, when using the 1st (array-to-array) corpus

Operator
Assign Variable To Array
Assign Variable From Array
Make Array
Variable To Literal
Add
Subtract
Multiply
Divide
Modulo
Assign Var from Var
Loop
Conditional (var $>$ 0)
Conditional (var1 $==$ var2)
Create 2D Array
Get 2D Array Size
Var to XY Point from 2D Array
Set 2D Array to 0 at XY Point
Set 2D Array to 1 at XY Point

Table 5.3: Operators available for GP, when using the 2nd (2D pattern) corpus

5.1.6 Human-Useful Corpus

As stated above, two different problem sets are used which represent human-useful programs of the kind that may be input into the system. The first problem set is array-to-array programs such as *extract even numbers*, *append arrays*, or *sort*. The second problem set is provided with an image to draw on a canvas and must synthesise the program which draws that image (this problem set is taken from So and Oh (2018)).

Each problem within each problem set is presented to the system as a set of 10 I/O examples. These are generated for each problem by feeding in 10 inputs and corresponding outputs of the form the problem requires (either a randomly generated input array and integer, or a canvas size). These 10 inputs are randomly generated from a fixed seed, to reduce internal variability between tests, allowing more accurate evaluation of the changes made by alterations to parameters or by guidance to the GP. This is designed to represent an unbiased input set.

Tables 5.4 and 5.5 provide a list of all human-provided problems used to test the system across the experiments. These are defined by a source-code implementation in both the custom language used in this work. An example of the behaviour of the array-to-array problems (First corpus), has been given, using a fixed sample input, to illustrate the behaviour.

Problem	**Example**
Abs	[4, -2, 1, 0, 3, -5] − > [4, 2, 1, 0, 3, 5]
ArrayLength	[4, -2, 1, 0, 3, -5] − > [6, 0, 0, 0, 0, 0]
ArrayToZero	[4, -2, 1, 0, 3, -5] − > [0, 0, 0, 0, 0, 0]
CumulativeAbsoluteSum	[4, -2, 1, 0, 3, -5] − > [4, 6, 7, 7, 10, 15]
CumulativeSum	[4, -2, 1, 0, 3, -5] − > [4, 2, 3, 3, 6, 1]
DivergentSequence	[4, -2, 1, 0, 3, -5] − > [0, 0, 1, -1, 2, -2]
FirstElementOnly	[4, -2, 1, 0, 3, -5] − > [4]
Identity	[4, -2, 1, 0, 3, -5] − > [4, -2, 1, 0, 3, -5]
IndexParity	[4, -2, 1, 0, 3, -5] − > [1, 0, 1, 0, 1, 0]
IterativeDifference	[4, -2, 1, 0, 3, -5] − > [4, -6, 3, -1, 3, -8]
KeepEvens	[4, -2, 1, 0, 3, -5] − > [4, -2, 0, 0, 0, 0]
KeepNegatives	[4, -2, 1, 0, 3, -5] − > [0, -2, 0, 0, 0, -5]
KeepOdds	[4, -2, 1, 0, 3, -5] − > [0, 0, 1, 0, 3, -5]
KeepPositives	[4, -2, 1, 0, 3, -5] − > [4, 0, 1, 0, 3, 0]
Negative	[4, -2, 1, 0, 3, -5] − > [-4, 2, -1, 0, -3, 5]
Pop	[4, -2, 1, 0, 3, -5] − > [4, -2, 1, 0, 3]
RemoveFirstElement	[4, -2, 1, 0, 3, -5] − > [-2, 1, 0, 3, -5]
RetainFirstHalf	[4, -2, 1, 0, 3, -5] − > [4, -2, 1]
Reverse	[4, -2, 1, 0, 3, -5] − > [-5, 3, 0, 1, -2, 4]
ShiftLeft	[4, -2, 1, 0, 3, -5] − > [-2, 1, 0, 3, -5]
ShiftLeftZeroPadded	[4, -2, 1, 0, 3, -5] − > [-2, 1, 0, 3, -5, 0]
ShiftRight	[4, -2, 1, 0, 3, -5] − > [0, 4, -2, 1, 0, 3, -5]
ShiftRightLossy	[4, -2, 1, 0, 3, -5] − > [0, 4, -2, 1, 0, 3]
ShuffleZerosToBack	[4, -2, 1, 0, 3, -5] − > [4, -2, 1, 3, -5, 0]
Signum	[4, -2, 1, 0, 3, -5] − > [1, -1, 1, 0, 1, -1]
Sort	[4, -2, 1, 0, 3, -5] − > [-5, -2, 0, 1, 3, 4]
SquareValues	[4, -2, 1, 0, 3, -5] − > [16, 4, 1, 0, 9, 25]
ToIterator	[4, -2, 1, 0, 3, -5] − > [0, 1, 2, 3, 4, 5]
Add	[4, -2, 1, 0, 3, -5] , 4 − > [8, 2, 5, 4, 7, -1]
Append	[4, -2, 1, 0, 3, -5] , 4 − > [4, -2, 1, 0, 3, -5, 4]
ClipToMax	[4, -2, 1, 0, 3, -5] , 4 − > [4, -2, 1, 0, 3, -5]
ClipToMin	[4, -2, 1, 0, 3, -5] , 4 − > [4, 4, 4, 4, 4, 4]
ConstantAddition	[4, -2, 1, 0, 3, -5] , 4 − > [4, 2, 9, 12, 19, 15]
FillArray	[4, -2, 1, 0, 3, -5] , 4 − > [4, 4, 4, 4, 4, 4]
GreaterThan	[4, -2, 1, 0, 3, -5] , 4 − > [-1, -1, -1, -1, -1, -1]
IterateFromStart	[4, -2, 1, 0, 3, -5] , 4 − > [4, 5, 6, 7, 8, 9]
LessThan	[4, -2, 1, 0, 3, -5] , 4 − > [1, 1, 1, 1, 1, 1]
MultiplesOf	[4, -2, 1, 0, 3, -5] , 4 − > [0, 4, 8, 12, 16, 20]
Multiply	[4, -2, 1, 0, 3, -5] , 4 − > [16, -8, 4, 0, 12, -20]
Subtract	[4, -2, 1, 0, 3, -5] , 4 − > [0, -6, -3, -4, -1, -9]

Table 5.4: The first corpus of problems, taking either a single array, or an array and an integer. Example provided of the behaviour of each problem, given a standard example input.

Problem
Square
HollowSquare
Parallelogram
HollowParallelogram
MirroredParallelogram
MirroredHollowParallelogram
RightTriangle
HollowRightTriangle
MirroredRightTriangle
HollowMirroredRightTriangle
InvertedRightTriangle
HollowInvertedRightTriangle
InvertedMirroredRightTriangle
InvertedHollowMirroredRightTriangle
IsoceleseTriangle
HollowIsoceleseTriangle
InvertedIsoceleseTriangle
HollowInvertedIsoceleseTriangle
RectangleWithEmptyTrapezoid
InvertedRectangle
obtuseTriangle
hollowObtuseTriangle
mirroredObtuseTriangle
mirroredHollowObtuseTriangle
invertedObtuseTriangle
hollowInvertedObtuseTriangle
invertedMirroredObtuseTriangle
hollowMirroredInvertedObtuseTriangle
VShape
Trapezoid

Table 5.5: The second corpus, a set of 2D image generation tasks, drawing simple geometric shapes. Examples illustrated in Figure A.1

5.1.7 Neural network design

The goal of the end-to-end system discussed in this experiment set is to extract a code fragment from a successfully-found program, then train a neural network to estimate which unsolved programs (as described to the NN by I/O examples) will contain this fragment. The network's probability estimate can then be used to guide which fragments to deploy in future problems by presenting their I/O to the NN.

The neural network is provided with the I/O examples in the form of a concatenated string (shorter-than-maximum arrays padded to maintain constant length). Values are presented in binary, 10-bit signed integers, with an additional bit for 'padding character' such that an empty array and a zero-length array do not appear identical.

The NN is trained on a synthesised training set of programs; the programs in this training set are generated at random but must exhibit some of the program features present in programs in S_F. The intuition here is that the neural network will thus learn to recognise which program features are likely to be present for unseen I/O examples, based on program features known to be useful in other programs requested by users, which are then useful to guide the GP search.

Each synthesised training corpus based on S_F has 20,000 programs for training and 2,000 for testing. The 20,000 programs are divided into 10,000 which do have a particular program fragment of interest, and 10,000 which do not have that fragment, with the NN trained to determine whether or not a given I/O example is likely to have that fragment in the corresponding implementation program. Each program in each set of 10,000 is assured to be distinct in functionality from every other program in the set, tested on its behaviour with respect to a fixed 10 I/O examples. Each new program in a training set is generated by selecting uniformly at random two programs already accepted, applying a crossover, then applying between 1 and 8 mutations (as described above in the GP section) while assuring that the fragment of interest still exists.

These experiments use a Feed Forward Network Network (FFNN) architecture for our array-to-array tasks, and a Convolutional Neural Network (CNN) for the 2D canvas tasks. The FFNN architecture takes the form of 4 layers of 128 neurons, seLu activation, interspaced with dropout layers set to 0.75 keep rate. Each layer was connected to all previous layers (dense block architecture). The single output node was a sigmoidal node (values constrained to between 0 and 1.0), representing the probability of a given fragment being present in the program assumed to have implemented the presented I/O example. The CNN architecture was two sets of: one convolutional layer of stride 2, kernel width 3, 32 channels, reLu activation followed by one max-pooling layer with a pool width of 2.

5.1.8 Experiment 3.1: Guiding GP based on fragments of human-written solutions

In this experiment the thesis firstly evaluates the performance of the Novelty Search fitness function, which is used by default in the remainder of all this work's experiments. To do this a core GP process was run both with and without novelty search on both of the test corpora, repeating each experiment 20 times (Full results in Appendix A.5). For the first corpus of 35 array-to-array programs, it was found that the GP without Diversity Boosting Repulsion Penalty (RP) achieves a find rate of 33% (n=20), with 27 programs having a find rate of > 0. With RP enabled, the find rates rises to 36%, a moderate boost equivalent to solving one additional problem. No problems which were at 0% find rate were boosted to > 0, however. For the 2D pattern corpus of 30 problems, the non-RP GP process had a success rate of 28%, with 15 problems having > 0 find rates. The addition of RP boosted this to 30%, with 18 problems with > 0 find rates. This indicates that the RP component of the fitness function had a measurable benefit, but further examination would be required to determine if the additional computational cost was worth the moderate gains. As it produced a positive effect, it was retained for all subsequent experiments.

With this baseline established, this experiment now evaluates the effects of guidance in the forms of provided 'hints' on a GP process. To do this the experiment examines the set of 1 or 2-line code fragments which can be cleanly isolated (with no dependencies) in the ground truth solution to each problem. This work defines this 'clean isolation' such that no variables used in the fragment have an assignment before the fragment (so their values are default by obligation). A GP pass is then run with the code fragment as a forced requirement, such that any program produced by the GP which does not include them automatically receives a penalty fitness of -10,000. Each experiment in this series is repeated 30 times to account for the inherent stochasticity in the GP process.

The experiment tests these fragments by deploying them into a GP search on 8 problems from the array-to-array corpus and 6 problems from the canvas corpus. From the first corpus the experiment selects mostly low-find-rate problems, to study which form of fragments would be useful to provide to achieve success in the GP, while in the second corpus it selects a more representative sample, to study how constraint-forcing behaves in general. Two problems of the same class are selected from the 2nd corpus (right triangles), to ensure that similar fragments provide similar results in similar circumstances (to give confidence in generality of these results). The baseline success, best find rate increase, and best fragment's operators for selected problems are presented in Table 5.6.

Here it can clearly be seen that forcing the inclusion of even the simplest code elements (one or two lines of the ground truth) into the GP's population allows the GP to find previously unsolvable problems. It can also be seen that fragments containing arithmetic operators (especially literal assignment) appear to have a stronger influence on success, possibly as they are harder to find, as their effects are far more subtle and complex than, say, presence of a loop operator. These should therefore be studied as high-utility candidates for deployment into GP processes in the future.

It can also be noted that some examples show a *decrease* in success rate (e.g. “Shift Right Lossy"). While no fragment reduced find rates to zero, it can be speculated that in some cases the provision of a fragment places the GP into an area of program space from which it is harder to reach the solution using our general-purpose fitness function (for example, meaning that this point in program space has larger regions of neutral landscape around it which are harder to traverse over), although this could be simply due to variance.

For full reports of results of every fragment for every program, see Appendix A.3

Problem	**Baseline**	**Max**	**Best Operators**
Append	0%	27%	Var=Literal 1; Addition
Cumulative Abs Sum	0%	3%	Loop; Read
Keep Evens	0%	7%	Var=Literal 2; Make Array
Retain First Half	0%	13%	Var=Literal 2; Divide
Reverse	58%	80%	Var=Literal 1; Make Array
Shift Right	0%	13%	Var=Literal 1; Loop
Shift Right Lossy	84%	80%	Var=Literal 1
Sort (Bubblesort)	0%	0%	(None)
Parallelogram	7%	30%	Var=Literal 2; Divide
Mirrored Hollow Parallelogram	13%	60%	Var=Literal 2; Divide
Hollow Right Triangle	87%	90%	Var=Literal 1; Subtract
Hollow Mirrored Right Triangle	63%	93%	Var=Literal 1; Subtract
Inverted Isosceles Triangle	46%	23%	Var=Literal 2
Trapezoid	7%	10%	Var=Literal 1

Table 5.6: Success rates for guided Genetic Programming Algorithm with forced inclusion of code fragments from ground truth. Baseline is unguided GP. Maximum is single best performing fragments. Best operators are those used in the highest-scoring fragment (first if tied) (n=30 per fragment, percentage success)

5.1.9 Experiment 3.2: Fragment Recognisability

This experiment starts with selected high-success seed programs from this section's first experiment. Both corpora are divided into logical categories (in the canvas corpus for example there are ‘triangles’ and ‘parallelograms’, in the array to array there are ‘conditional-using’ or ‘no loop’), and from these the experiment selects the problem with the highest baseline find rate to use as seeds (Baseline the same as in

5.1.8, Appendix A.5). It then takes 10 source code fragments present in these seed programs which exhibit a positive effect of find rates, and uses these fragments to generate completely synthetic training corpora for neural networks trained to predict *whether or not an I/O example will include a particular fragment* in its source code solution. From the array to array problems it tries to select fragments which allow the behaviours demonstrated by the logical category of program they are drawn from. The source codes of the canvas programs were less able to be readily decomposed into these logical blocks, so in the canvas set we include a set of 3 fragments with two being subsets of another, to assess the effects of fragment complexity on recognisability.

Table 5.7 shows how effective the trained neural networks are at then correctly predicting the presence of these fragments in the I/O examples from the two problem sets (which are not part of the synthetic training sets). The array-to-array corpus uses a Feed Forward NN, the 2D pattern corpus uses a Convolutional Network architecture. More detail on fragments used in Appendix A.6.

Fragment	**Accuracy**
Nonstandard Array	73 %
Loop + Iterator Minus One	N/A
Loop + Read	65 %
Literal 2	76 %
Read	67 %
Add	48 %
Loop	67 %
Loop + Iterator Plus One	N/A
Conditional	63 %
Loop + Iterator Mod 2	73 %
Two Draw Operators	67 %
Add	77 %
Loop + Draw On Iterator X	62 %
Half	84 %
Half + Loop to Half	80 %
Half+Loop+Draw Depends on Iterator	85 %
Conditional	62 %
Loop + Conditional	59 %
Loop + Loop, Add Iterators	53 %
Loop + Loop, Subtract	45 %
Fragment	**FFNN**

Table 5.7: Percentage accuracy on neural network estimates of fragment presence in non-seed programs, after training on synthetic datasets derived from seed programs (one corpus from seeds with fragment, one for those without). Results are averages of networks which passed validation (> 0.5 accuracy on seed programs). N/A results are those where non passed validation. (n=20)

This data shows significant variance of prediction success (from 45% up to 85%) but overall shows that the neural networks do exhibit the ability to accurately predict that a particular source code fragment will exist in the solution to a given I/O problem – even though these neural networks are trained on entirely synthetic data.

In practice a system can use these trained networks when determining which fragments to recommend for inclusion in a GP search; this is presented in the following experiments, culminating in a full end-to-end system which uses any programs found by GP processes as seeds, and automates fragment extraction, then deploys them using this approach to neurally guided fragment presence estimation.

5.1.10 Experiment 3.3: Chosen Fragment Deployment

In this experiment, the fragments from the above NN experiments were employed to guide the GP process, based on the average estimates of fragment presence by the trained neural networks. This demonstrates the efficacy of the system in taking successfully-found solutions to easy (high success rate for an unguided GP) problems and transferring their characteristics, via trained NN predictors, to find solutions to more difficult problems (low base success rate).

From all fragments predicted to be present in the source code solution of an unseen I/O problem, the system ranks the fragments to actually deploy based on how 'difficult' they were to find. The hypothesis is that certain fragments will have greater effect on GP performance. NN recognition of properties such as "contains a loop" is of low utility, as the GP would find this element of the solution rapidly. What is of greater interest is recognition of fragments which are specific to the target problem and with a lower probability of being generated by random mutation. As such, for each problem to solve, select the fragment taken from the seed program with the highest expected number of generations to find a solution (if a problem is found 10% of the time, it has an expected generation count of $30,000$, while if a problem is found 100% of the time, its difficulty is the average the generations taken to produce a solution).

Corpus	Success Rate (vs Baseline)	Gained	Lost
1st (1D Arrays)	46 % (38%)	5	1
2nd (2D Array)	39 % (36%)	4	1

Table 5.8: Success of the GP when guided by the fragments selected by the neural network. 'Gained' are problems which have a $> 0\%$ success rate which have a baseline of 0%, 'Lost' are those which previously had $> 0\%$ but now have 0%. (n=20)

As can be seen in Table 5.8, guiding the GP using this process produces notable improvements to the overall program synthesis success rates. Average find rates were boosted, and crucially a number of problems became solvable which were not before, with only a single problem in each problem set failing to be solved. This set of newly-solved problems here includes instance in which the bubblesort algorithm, a problem which throughout this work has rarely found by any approach, was successfully found.

This improvement was strongest on the approach which selected the estimated-rarest fragment for use, clearly indicating that the fragments are not equivalent in their usefulness. The ideal way in which to extract the most useful fragments for the GP to use, based on those available from NN predictions, therefore remains a topic of future work.

The full set of guided GP results is available in detail in Appendix A.4.

5.1.11 Conclusion to Experiment 3

A number of lessons were learnt from this experiment, which can be used to design the desired end-to-end transfer learning genetic programming algorithm.

Neural Networks can recognise code fragments is a core discovery. This confirms that the capabilities displayed by feed forward networks in DeepCoder Balog et al. (2017) are transferable to this thesis' target domain of code synthesis in a Turing-Complete language. Specifically both this thesis and the literature show that parts of a program can be recognised by a neural network based on how they impacted a program's behaviour.

Genetic Programming can be guided successfully by partial programs. This experiment demonstrated a mechanism by which a GP can have its performance enhanced by a neural network's suggestions. This was achieved by essentially reducing the program-space which the GP operates in, to restrict it to only the programs which contain the supplied code fragment. This space is empirically shown to be superior to navigate, although work would be needed to disentangle the effects of reduced space size and the effects of supplying a starting point other than the empty program. This is not necessarily the only mechanism of GP guidance, with a core element currently missing being the ability to accept multiple code fragments as guidance. This would allow for even higher specialisation of the search space, which may allow far higher performance.

These two properties of neural networks and Genetic Programming can now serve as the basis for the remaining work in this thesis. The aim is to maximise the advantage provided to the GP by the NN. To do so optimally requires as much as possible to be drawn from as many donor programs as possible, but must be balanced against the risk of poorly targetted guidances reducing the GP's performance by constraining it to subspaces of program space which: do not contain a valid solution; have poor navigability; start the GP at a harmful point in space (one which does not have a clear path towards a desired solution); and/or feature valid solutions far less frequently than the entirety of program space does.

Chapter 6

Experiment 4: End-to-End System

This experiment uses fully *automated* fragment extraction, sourcing the fragments from solved problems and redeploying those fragments to new problems based on NN guidance. It builds on the the lessons learnt in the previous experiments, to create an end-to-end Transfer Learning System. It represents the full system proposed by the thesis.

This work was published twice in the literature. Firstly under the title “Neurally guided transfer learning for genetic programming" Wild and Porter (2021) and secondly under the title “Multi-Donor Neural Transfer Learning for Genetic Programming” Wild and Porter (2022).

6.1 Fragment Extraction and Evaluation

Once a successful GP run has completed, having synthesised source code which replicates the desired behaviour, as defined by a set of I/O examples, it can be used as source material for later successes.

Various subsets of the source code can be extracted, programs generated to either include or exclude them, and a network trained to recognise their presence or absence. This allows the network to learn information about the functionality of the code fragment, how it affects the output of programs it features in.

When later problems are encountered, all neural networks trained so far can be presented with the I/O example set, to determine which fragments might be present in the program which generated the example behaviour.

6.1.1 Selecting Fragments for Extraction

The number of possible subsets of program lines is $2^{number\_of\_lines}$, which may rapidly becomes prohibitively expensive as the number of lines in the solution grows.

This work employs a mechanism to select fragments in a way which attempts to extract the most unique aspects of the program and preserve as much of their functionality as possible.

A subset of the lines of a program is considered valid for use if no variable read by a line in the subset has been written to earlier in the program by a line not in the subset. This ignores the effects loops would have on the order of operations, to allow loop contents to be extracted despite possible variable writing earlier in program execution order. Loop operators themselves, however, are included, as they write to their iterator variable.

This avoids lines depending on lines which do not exist in the fragment, and therefore would not be transferred to a new context, and provides a way to keep functional blocks of code together.

A maximum number of lines is also imposed, to further cut down on the number of potential fragments requiring evaluation. For this work, a maximum of 4 lines and a minimum of 1 is employed. ENDBLOCKs are also excluded, as the fragment injection mutation process allows them to be inserted automatically. NO-OPERATION lines are not considered a part of the program for these purposes.

Once a set of fragments matching these requirements has been produced, they are compared against existing fragments. The aim is to determine which element is most 'novel' within this synthesised program.

The purpose of fragment transfer is to reduce GP search times as far as possible (including from the timeout case where the GP process terminated unsuccessfully), therefore fragments found in fewer GP generations are not preferred. These fragments were rapidly synthesised by the GP algorithm, and therefore poor candidates for transfer, as they provide lesser benefit while preventing other, more computationally expensive to synthesise fragments, from being deployed. As a result, fragments similar to those from faster-to-solve problems are to be avoided.

In order to achieve this, each fragment is compared against all other fragments extracted from problems which were solved in fewer GP generations than the current problem was. The distance metric from the GP diversity boosting repulsion penalty search component, described in subsection 4.2.3, is re-used. This provides a metric to determine the relatedness of all fragments to all previously seen fragments. Note that the fragments are normalised prior to the point, refactoring variable naming as described in subsection 4.2.5, removing any code distance due to variable naming alterations.

For each fragment, the distance to the closest already-seen fragment is found. Note this represents all fragments passing the requirements described above from all programs synthesised by the GP which took fewer generations than the currently solved program, not merely those selected via the process described below. Fragments which are identical are rejected entirely. If no fragments are viable as a result of this exclusion, the process terminates here.

The fragments are then sorted, ordered such that the most 'novel', the one which has the highest distance to any existing more rapidly found fragment, is first. The measure of this 'rapidity of finding' is how many generations it has taken to find the solution from which this fragment was taken. This includes previous runs of the GP on the same problem.

6.1.2 Fragment Evaluation: Synthetic Training Corpora

The process then iterates through this ordered list of fragments. For each fragment under consideration, a fragment evaluation pass is performed.

Two corpora of programs are generated, one which features the fragment in all its programs, one in which it is never present. These are 2500 programs in size, and are created via a genetic mutation process. This number was chosen based on preliminary experiments, which showed that corpora of sizes up to 25,000 did not lead to any further performance gains. The populations are initially seeded from the programs which have been synthesised by the GP process on user-supplied I/O examples. The corpus requiring the fragment's presence will only have a single seed program, the recently solved problem from which the fragment was extracted. All other programs will form the seeds of the second corpus, where the fragment is not present. This process was designed to best apply the lessons learnt from Experiment 1, in the most automated fashion possible. Preliminary experiments in which the last generation from the GP, or sub-populations from GP runs, were found to lead to far lower accuracy neural networks.

For each corpus, once the seeds have been added, the population is grown genetically. Two parents are selected, uniformly randomly from all programs currently in the population, and crossover applied, as in the GP. The child is then mutated, again as described above, including randomised number of mutation steps.

The code is assessed, to determine if it has retained the desired fragment or to ensure it has not generated it (as required by the respective corpora). If it passes the fragment presence test, all introns are removed via static code analysis, and it is executed on a fixed input. This input sequence is randomly generated at the start of the corpus-generation process, and is identical for all programs tested. To be added to the corpus, its outputs on these inputs must differ from all programs already present therein, to ensure that it has a unique functionality. This continues until both corpora reach 2500 programs.

In certain conditions, no new programs can be added, as the requirements fail to be met. A primary cause of this is inability to generate sufficiently functionally dissimilar programs. If this occurs, a timeout exists, terminating fragment evaluation. This timeout is set to 210 seconds, for this work.

If the fragment fails to produce these training corpora, it is skipped, and the next fragment evaluated, to a maximum of 8 attempts total. The process also ends if 4 fragments succeed in being evaluated.

6.1.3 Encoding Values for use in Neural Networks

Once two sets of 2500 programs have been generated, one containing the fragment under examination and one excluding it, the neural network can be trained to recognise its presence in I/O mapping examples.

First these examples need to be generated. Random inputs are selected, to more closely resemble the expected inputs, or at least to cover a broader range of cases. While more complex input generation strategies exist, as discussed previously, these may bias the network's recognition abilities, as they do not reflect a uniform input distribution. Therefore a simpler strategy is employed, to ensure the underlying framework is effective, and improvements such as coverage-boosting input selection is left as future work.

10 examples are used, to match those supplied by the simulated user of the system. These are passed into the programs in the corpora, to produce sets of 10 I/O examples.

These can be converted into fixed-length binary strings, based on the input-output types. Two type structures are studied, programs taking both a 1D array of integers and a single integer inputs and returning a 1D array of integers, and programs taking a positive integer and returning a 2D array of boolean values (implemented as a 2D array of integers forced to adopt only either 0 or 1).

Two separate encodings are used for the two types of problem faced, along with two separate network architectures.

6.1.4 1D Arrays of Integers

In the first case, the encoding strategy must describe both integers and arrays of integers. Integers are represented as binary encodings, each input into the neural network corresponding to an input neuron, and taking a value of either 0 or 1, with a fixed length of 10. The first binary value represents the sign of the integer, and is 0 if the integer to represent is positive, 1 if it is negative. The further 9 represent powers of 2, their sum representing the value of the integer. The first of these, the second value in the representation, is equal to $encoding[1] = integer\%2$, the second equal to $encoding[2] = integer\%4$ and so on. If the integer exceeds the maximum which can be represented the encoding will be incorrect, but this is a necessary concession to make in order to keep the encoding of fixed length. To maintain input integers below sizes which have a high probability to cause an overflow, all input integers are chosen to remain in the range $[-256, 256]$.

The arrays can then be encoded as sequences of integers. The arrays must be assigned a maximum length for the same reason as integers must adopt a fixed length encoding, because the network input will be fixed length and cannot expand at runtime. The maximum array length is set as 16 for this work. This imposes a limit of 8 on the randomly generated input arrays, as the problems studied include functions with concatenate the input array to itself, therefore requiring double the length of output as input.

Each array is encoded as a fixed length sequence of annotated integer encodings. For each index in the maximum array length, a value of 0 or 1 is appended to an integer encoding, to indicate if the index falls within the array's current length or not. A value of 1 indicates that the array extends this far, and that the following integer encoding is a true value. If the array does not, the integer encoding is set to 10 values of zero and the preceding value of 0 indicates that the array is shorter than this index.

Each I/O example can then be assembled, by encoding first the input array, then the input integer, then the output array. This forms a fixed length sequence with consistent ordering, allowing effective processing by the neural network.

6.1.5 2D Array

For the 2D data type problems, the only input required is the desired size of the output array, as this encapsulates all necessary information for the problems which are studied in this work.

Similarly to 1D arrays, a fixed maximum is necessary, and a representation of the current array size in comparison to this maximum.

The encoding form is simpler, due to the binary nature of the values involved. A single encoding vector is produced, representing a 2D array of maximum size (a 12 by 12 square). It is transformed from 2D to 1D by concatenating each of its rows into a single vector.

Each element in this vector is formed of two binary values. The first indicates whether this given point falls within the current output 2D, with 1 indicating the output is large enough to encompass this point, 0 that it is not. If the 2D array does encompass it, the second binary symbol is set to 1 if the output 2D array is 1 at the mapped index, 0 if not (and defaults to 0 if the 2D array does not extend to this point).

6.1.6 Neural Network Architectures Employed

With all I/O example sets expressed as sets of fixed length binary values, they can be fed into the networks for training. Again, for the two cases, two separate network architectures are employed, as was shown to be most effective.

6.1.7 1D Arrays of Integers

For this data-type problem set, a feed forward neural network (FFNN) is employed. The architecture is relatively simple, employing a single type of network layer, and a single type of activation function (other than for the output neuron).

The architecture consists of 5 dense layers of 128 neurons, the first connected only to the input layer, the subsequent 4 connected to every previous dense layers, in what is termed a densely connected architecture Huang et al. (2018). The activation function on every layer is SELU, or 'Scaled Exponential Linear Unit' Klambauer et al. (2017), which has been shown to have superior performance on deeper Feed Forward networks than commonly used other functions. Preliminary experiments into neural network performance indicated this structure had higher performance than any combination of standard FFNN architecture, in which layers were connected simply to the previous layer or industry-standard ReLu activation functions.

Dropout (Srivastava et al. (2014)) is used on every dense layer, with a retention probability of 0.75.

The output layer, reading only from the last of the dense layers, is a single neuron with sigmoid activation, asymptotically limiting its output range to between 0 and 1. This allows it to represent probability estimates and guarantee a valid (within acceptable range) output.

The input to this network is therefore a fixed length vector of values (which are set to either 0 or 1), and the output value is either 0 or 1. All 10 I/O example pairs are concatenated together into a single vector for this purpose. For labels, 0 indicates that the I/O example set encoded into the input vector does not contain the target fragment, 1 indicates that it does.

6.1.8 2D Array Inputs

For this data-type, a Convolutional Neural Network (CNN) is employed.

The CNN architecture is formed of 10 separate processing elements, each of which handles a single 2D canvas example. Each example in the problems studied takes the form of a single 2D array, and is processed by a set of convolutional layers. The results of these sub-elements is concatenated after processing, to combine the results into a single output.

Each sub-processor takes a 2D array of maximum size (which defaults to 12), initially filled with 0s. For every $element(i, j)$ in the example, if the value in the example is 1, the input array at (i, j) is set to 1. This transfers the pattern from the example into the input array, discarding any array size information.

Each of network's sub-processors consists of a Convolution layer, reading in the initial image, followed by a Max Pooling layer, followed by another Convolution layer, followed by another Max Pooling Layer, followed by a dense layer to return the data in a 1D format (The previous two process data in a 2D fashion).

The Convolution layers are 32 channel, with a kernel size of 3, a stride of two and ReLu activation. The Max Pooling employs a 2 by 2 window. The final dense layer is 96 neurons, ReLu activation.

The final dense layers of each sub-processor has its outputs concatenated and fed into another dense layer, of 96 neurons with ReLu activation. This layer processes all seen data, and feeds it into the output node.

As above, the output node is a single sigmoid activation neuron, representing probability estimates.

6.1.9 Training the network

The network is trained to minimise the mean squared errors between its estimates and the training data set's values. The ADAM optimiser is used Kingma and Ba (2015), with a training speed of $6 * 10^{-4}$. Minibatches of size 32 are used.

Three sub-corpora are formed from each of the two main ones, by subdividing the corpora after shuffling them. The first, the training set, is 80% of the data, the next 10% are used as validation set, and the remaining 10% as a held-out testing set, to evaluate the performance of the networks.

6.1.10 Fragment Injection

For the later experiments, the GP is expanded to grant it a new mutation, which employs the extracted fragments. This mutation allows fragments to be transferred from previously solved problems, based on neural network estimations of presence.

For every fragment which has been extracted and evaluated, in the mechanism described in subsection 6.1, a weighting is determined.

This weighting is computed by the product of two factors. Firstly how strongly the neural network estimates it to be present in the program which produced the I/O example, secondly how rare it is, as a fragment.

The first value is simply computed as $neural\_network\_output - 0.5$. If the neural network estimated the fragment as having less than 0.5 probability of presence in this program it is automatically discarded. As such this value ranges between 0 and 0.5, with fragments barely above threshold having very low probability of being selected.

The second value is $1/(0.1 + presence\_estimate)$. This value $presence\_estimate$ is the number of I/O example sets which the neural network estimates the problem to be in divided by the total number of I/O example sets, so a value between 0 and 1 inclusive. The fewer I/O examples the network estimated the fragment to be present in, the higher this value, up to 10, if no IO examples are believed to contain this fragment (although that would imply it would never be selected due to previous factor). Presence in this case is simply taken as IO example sets in which the neural network's presence estimate > 0.5. It is assumed the system is presented with a batch of I/O examples by the user, and can view upcoming problems in the corpus, but this functionality would remain unchanged if it were operating purely on IO examples it had previously solved as well as the currently faced problem.

This rarity factor aims to select for 'specialist' fragments, which are unique to this particular problem class, as they are hypothesised to have a higher utility to the GP.

Fragments are selected based on this weighting using roulette selection producing a single candidate for insertion. Insertion takes place by selecting a random point in the existing program. The first line of the fragment is inserted there (displacing later lines downward in the program). For each further line of the fragment to be inserted, if there are later lines in the program, n of these may be skipped, causing them to become interlaced into the fragment. n is chosen by iteratively selecting either to increase n by 1, or to terminate, with 0.5 probability of either option, so skipping 0 lines has a 0.5 probability, skipping 1 has an 0.5 probability, 2 has 0.25 and so on. This injection process allows complex mutations which retain existing functionality while adapting it, such as by wrapping code elements in a loop or conditional.

Fragments are also adapted before injection by refactoring their variables. First they are normalised, such that the first variable references is set to the first variable which won't be used as inputs into the program. The second is then the next variable sequentially, and so on. When injecting the fragment, these variables are randomly offset, by adding to the variable index referenced. A random integer value is uniformly selected between 0 and the maximum which would not cause the largest variable index to exceed the permitted number of variables. As such, if a fragment initial was $variable\_6 = variable\_4 + variable\_8$ it would initially be normalised to $variable\_2 = variable\_3 + variable\_4$ if there were two input variables. It could then subsequently be offset to become $variable\_6 = variable\_7 + variable\_8$, as long as $variable\_8$ was in the permitted number of variables.

This allows fragments to retain elements of their logical inter-dependencies, while potentially injecting them into existing code in a way which allows both code structures to be merged into a single whole. Capability exists both variable re-use, such that variables will be written to and read from by the fragment and the existing code, or for the fragment to be inserted in a way which avoids use of already-used variables. Due to the generations sizes involved and mutation probabilities, both mutation approaches are likely to be attempted during the same GP process, with the most successful approaches retained in the population.

In the same mechanism as before, if insufficient ENDBLOCK operators exist, additional ones are appended to the end of the program, and while excess ones exist the last one is removed.

6.1.11 Re-Evaluation of Fragments

Once a program is successfully synthesised by the GP, it can be used not just to extract new code fragments but also to re-train networks assigned to existing fragments.

Each program synthesised is assessed to determine which already-accepted fragments are present in it. For all that are, new training corpora can be generated, now with a larger number of positive-case seed programs. This is desirable as the initial training positive corpus (the corpus featuring the fragment) for each fragment will often have only a single seed program to generate from, the program from which it was extracted. This could lead to considerable bias in terms of programs generated, and misrepresent the region of program space occupied by programs containing this fragment. By allowing retraining from a larger collection of seed programs, a more representative distribution of I/O examples can be generated.

The process is identical to the first training pass. Two corpora of 2500 programs are generated, using all synthesised programs as seeds, dividing them between the two corpora's requirements. A newly generated network is employed (rather than re-training the existing one), and assigned to the fragment, replacing the existing one.

6.1.12 Summary

This subsection has covered the design of a Transfer Learning system for Genetic Programming, which is able to operate without human assistance. The system takes only problems, in the form of input-output example sets which demonstrate desired program behaviour, and synthesises solutions using a Genetic Programming algorithm. If successful, the newly generated programs' code are decomposed into code fragments, which a network learns to recognise based on its influence on I/O examples.

These extracted fragments can then be redeployed using mutations to solve future problems, aiming to remove the need to re-evolve complex structures which have already been generated for an earlier problem. This allows easier problems, those solvable without Transfer Learning, to be used to provide genetic material to allow solving harder problems, which the GP would be unable to solve in isolation.

The system is able to operate in a continuous fashion, constantly receiving I/O examples as problems and maintaining an ever-growing list of useful mutations. It is also able to refine its ability to estimate the presence of fragments as it successfully deploys them, to better generalise the neural networks' training.

6.2 Results

For both the array- and canvas-based corpora, the experiment runs through the entire problem set twice, in the same order both times. This allows the system to successfully find any problems that are possible in its first pass, then use extracted fragments on unsolved problems in a second pass. In order to provide a fair comparison against a baseline GP-only system, the experiment allows that baseline system to also have two full attempts at both problem sets, yeilding a roughly equivalent number of total GP generations available.

Corpus	Success Rate (vs baseline)	Problems Solved
1D Arrays	**76%** (37%)	**33** (28)
2D Arrays	**49%** (37%)	**29** (22)

Table 6.1: Success of the transfer learning system in a fully automated pass through both the 1st corpus (1D array to array problems) and 2nd corpus (2D array/canvas pattern generation problems).

High-level results are shown in Table 6.1 against the GP-only baseline, measured in both success rate and total problems solved. The *problems solved* column indicates, as a total across 30 repeated runs, how many of problems from each corpus were solved under each approach. The *success rate* column is calculated by determining, for each problem individually, how many of the 30 repeated runs found a solution to that problem; then averages this percentage across all problems in the corpus to determine the above average success rate.

Detailed per-problem results for the array-based corpus are shown in Table 6.2. Out of a total of 30 experiment repeats, this table shows the number repeats for which the baseline (B') succeeded at each problem, and the number of repeats for which the synthesis framework succeeded. It also shows the number of those successes which were found to *generalise* (g) to an additional 1,000 I/O examples beyond the 10 examples used to synthesis the code. Fisher's Exact Test was used to establish a statistical significance for each separate problem. This was chosen as each problem should be considered independently, as the TL system could have different effects on different problems, especially as different problems have different baseline starting values. As a result, each problem can be considered to have two probabilities (probability of baseline GP succeeding, and probability of TL GP succeeding). Fisher's Exact Test is designed to handle specifically this form of significance test. This work considers a statistical significance of 0.01 to be sufficient to report as a result. This lets us see which problems the TL system can be considered to have improved the performance, and which it showed limited success on.

Problem	**Baseline (g)**	**TL (g)**	**Fisher Significance**
Add	14 (7)	25 (17)	0.002
Append	0 (0)	28(15)	$4.193 * 10^{-15}$
CumulativeAbsoluteSum	2 (1)	12 (5)	0.002
CumulativeSum	5 (4)	21 (14)	$2.917 * 10^{-5}$
KeepEvenIndices	7 (2)	30 (23)	$8.705 * 10^{-11}$
ClipToMin	16 (2)	27 (17)	0.001
RetainSecondHalf	0 (9)	10 (3)	$3.985 * 10^{-4}$
Sort	0 (0)	0 (0)	1.0
Subtract	10 (5)	25 (23)	$8.247 * 10^{-5}$
Abs	11 (5)	22 (19)	0.003
GreaterThan	3 (2)	5 (5)	0.25
IndexParity	29 (22)	30 (29)	0.5
FirstElementOnly	10 (3)	30 (23)	$7.167 * 10^{-9}$
Identity	29 (25)	30 (30)	0.5
DivergentSequence	12 (0)	28 (19)	$8.975 * 10^{-6}$
Double	14 (4)	30 (28)	$9.720 * 10^{-7}$
ShiftRight	0 (0)	26 (16)	$3.921 * 10^{-13}$
ShiftRightLossy	18 (8)	30 (28)	$6.181 * 10^{-5}$
ShiftLeft	5 (2)	30 (24)	$2.744 * 10^{-12}$
ShiftLeftZeroPadded	20 (8)	28 (24)	0.009
RetainFirstHalf	0 (0)	17 (9)	$3.092 * 10^{-7}$
LessThan	4 (0)	9 (5)	0.076
Multiply	16 (8)	24 (22)	0.020
Negative	23 (15)	30 (27)	0.005
Pop	6 (1)	30 (26)	$1.646 * 10^{-11}$
KeepPositives	18 (3)	30 (27)	$6.181 * 10^{-5}$
KeepEvens	0 (0)	0 (0)	1.0
ArrayLength	29 (25)	30 (30)	0.5
ArrayToZero	29 (25)	30 (30)	0.5
KeepNegatives	18 (9)	29 (22)	$5.022 * 10^{-4}$
KeepOdds	0 (0)	1 (1)	0.5
Reverse	13 (8)	24 (21)	0.003
CatToSelf	3 (0)	21 (21)	$1.611 * 10^{-6}$
CatZerosToSelf	7 (2)	30 (24)	$8.705 * 10^{-11}$
ClipToMax	13 (10)	27 (18)	$1.159 * 10^{-4}$

Table 6.2: Full breakdown of all problems in the first corpus, 1D arrays, with success rates for 30 runs of both baseline (GP without TL enabled) and the Transfer Learning system. Two attempts were permitted for both approaches, and success counted if either attempt succeeded. Programs which were found to generalise to 1,000 additional random inputs are shown in brackets. Statistical significance established per problem by Fisher's Exact Test.

For the unsolved problems, such as ‘Sort’ and ‘Keep Evens’, the system clearly did not improve upon the baseline at all, and the results were unchanged. Similarly for those which the baseline GP process solved with near 100% accuracy, such as the Identity function, there is not sufficient data to indicate that the system had an influence, and that the results aren’t due to pure chance. On many tasks the results are clear, however, that the approach has improved significantly on the performance of the baseline. It is seen that there are multiple tasks, such as ‘Shift Right’ and ‘Pop’ which have probabilities of both results being drawn from the same distribution (that is to say the probability that the null hypothesis is wrong) of far less than the chosen limit of 0.01.

One core set of problems on which the approach was more successful was those which required the output array length to differ from that of the input array. Various problems of this class existed in the corpus, specifically “append" and “pop", which changed the length by 1, “retain first half" and “retain second half" which halve the length of the output array compared to the input’s length, and “cat to self" and “cat zeros to self" which concatenate an array onto the end of the input array, and “first element only" which requires an output array of length 1. This suggests particular success of transferring useful material between problems which have similar traits.

In terms of generalisability of the synthesised code, similar increases in good solutions using the approach across each problem are seen compared to the baseline. Both the baseline and this work’s approach have fewer generalisable solutions than total solutions (total solutions here being those which simply fulfill the 10 IO examples, but fail on new unseen examples), indicating neither solution leads to perfectly generalised solutions each time.

Problem	**Baseline (g)**	**TL (g)**	**Fisher Sig.**
Square	30 (24)	30 (26)	1.0
HollowSquare	30 (21)	30 (25)	1.0
Parallelogram	0 (0)	2 (1)	0.25
HollowParallelogram	1 (0)	2 (1)	0.5
MirroredParallelogram	11 (2)	13 (7)	0.2
MirroredHollowParallelogram	9 (2)	6 (6)	0.166
RightTriangle	30 (23)	30 (25)	1.0
HollowRightTriangle	30 (15)	30 (25)	1.0
MirroredRightTriangle	15 (8)	30 (21)	$2.916 * 10^{-6}$
HollowMirroredRightTriangle	12 (4)	27 (19)	$4.398 * 10^{-5}$
InvertedRightTriangle	29 (18)	30 (23)	0.5
HollowInvertedRightTriangle	15 (3)	24 (16)	0.011
InvertedMirroredRightTriangle	30 (24)	30 (25)	1.0
Inv'HollowMirr'RightTriangle	30 (14)	30 (25)	1.0
IsoceleseTriangle	0 (0)	2 (1)	0.25
HollowIsoceleseTriangle	1 (0)	7 (4)	0.024
InvertedIsoceleseTriangle	15 (6)	24 (18)	0.011
HollowInv'IsoceleseTriangle	9 (1)	15 (12)	0.062
RectangleWithEmptyTrapezoid	2 (2)	2 (0)	0.5
Inv'dRect'WithEmptyTrapezoid	0 (0)	7 (2)	0.005
ObtuseTriangle	4 (0)	9 (5)	0.076
HollowObtuseTriangle	10 (2)	16 (11)	0.066
MirroredObtuseTriangle	0 (0)	0 (0)	1.0
MirroredHollowObtuseTriangle	2 (0)	2 (0)	0.5
InvertedObtuseTriangle	0 (0)	1 (0)	0.5
HollowInvertedObtuseTriangle	1 (0)	4 (2)	0.166
InvertedMir'ObtuseTriangle	0 (0)	3 (0)	0.125
HollowMir'Inv'ObtuseTriangle	0 (0)	2 (1)	0.25
VShape	18 (2)	27 (18)	0.006
Trapezoid	0 (0)	2 (1)	0.25

Table 6.3: Full breakdown of all problems in the second corpus, 2D arrays representing images, with success rates for 30 runs of both baseline (GP without TL enabled) and transfer learning system. Two attempts were permitted for both approaches, and success counted if either attempt succeeded. Statistical significance established per problem by Fisher's Exact Test. (n=30)

A similar detailed breakdown of problems from the canvas-based problem set is shown in Table 6.3. These results again demonstrate that this work's approach generally performs better than the baseline, indicating successful inference of which code fragments from solved problems are useful in unsolved ones. The overall success rates are more similar here than the previous problem set (i.e., across the total 30 runs, the number of programs found at least once by each approach), with the more significant difference being the frequency with which the approach finds a solution to a given problem. This suggests the most significant result on this corpus was in its ability to improve the baseline GP on already-findable problems, rather than allowing new problems to be found.

Again, in these results, it can be seen that a significant number of solutions generalise to 1,000 additional input examples, and again this number of generalisable solutions is often significantly higher using this approach. In both problem sets, in fact, both approaches appear to be roughly equivalent in terms of their generalisability as a proportion of their solutions; the fact that the approach generates more solutions overall thereby increases its total number of solutions which generalise.

6.2.1 Computational Cost

Additional, in this section this work considers the overall computation cost of the baseline and the transfer-learning-based approach; although both experiments had the same number of primary GP generations available to solve each problem, this work's approach uses additional steps such as NN training. Measured in compute time, the first problem corpus took 15 hours for the baseline on the hardware employed, while the full transfer learning system took 24 hours. On the second corpus, the baseline took on average approximately 49 hours, while the TL system took 80 hours. The training time of the neural networks was not recorded specifically, but can be assumed to represent a non-negligible proportion of this additional time, as approximately 350 networks were trained each time, one per fragment. It can be also noted that the hardware setup entailed NNs being trained using CPU resources, rather than dedicated high-performance GPU hardware; using GPUs for NN training may offer a significant time saving.

6.2.2 Summary

The results in this section clearly demonstrate the advantages of the transfer learning approach in this context, and the success on two distinct program domains suggests the approach has a degree of generality to it. The kind of program generated by the system is illustrated in Algorithm 2, demonstrating a solution to the 'abs' problem converted to Java code; while some of the program is certainly unusual, it does represent a generalised solution to the problem in question.

In the following two subsections the work examines two specific elements of the approach in more detail, to provide further context to the main results. Firstly this subsection examines how fitness curves over time appear for both the baseline and the system, to help understand how transfer learning has affected GP population fitness behaviour. Then it examines the effect of injecting random large fragments of code into a GP process, to confirm that *NN-selected* fragments are the source of improvements seen in this section, rather than those improvements coming simply from the injection of larger fragments of code in general.

6.2.3 Fitness curves

This section examines how fitness-over-time plays out in both the baseline and the transfer-learning-based system, providing further insight into the particular affect of transferred material. Figure 6.1 shows the average and median fitness for all problems over time for the array-based problem set. Here can be seen a very similar overall shape between the two systems, but with notably higher fitness throughout for the approach (particularly clear in the median).

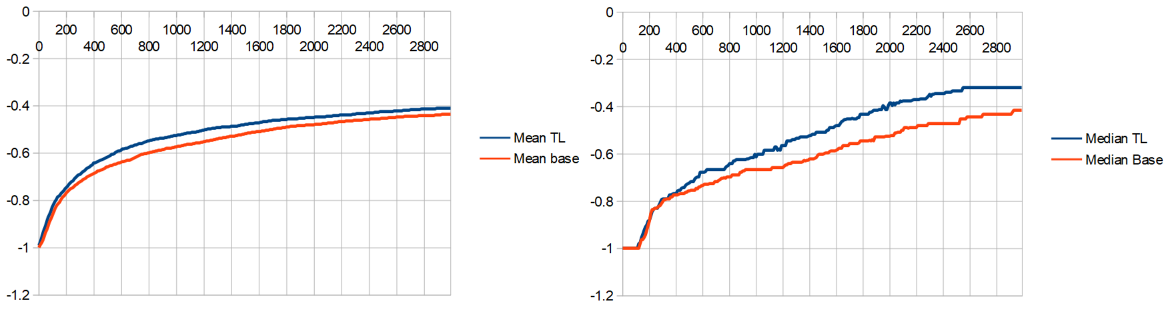


Figure 6.1: Fitness curves for GP and TL systems on the first corpus, with mean on the left and median on the right. TL system in blue (higher curve both times). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=900)

Figure 6.2 shows fitness over time for a specific problem ('Keep Positives') in the array-based problem set. In this particular one, this work's approach on the right shows interesting behaviour in the median graph. The TL approach had a success rate of 100% on this problem, and the median graph shows a dramatic jump around generation 500. This implies that certain programmatic elements were found which allowed fitness to be increased very rapidly. The baseline system, by comparison, does not have this characteristic. The TL system, working on this problem, also has a much stronger early increase than the baseline GP alone, with a high slope from generation 0. This, too, is likely due to genetic material being transferred from

Algorithm 2 A translated example of solution produced to the ‘abs’ problem, generated by the GP process, a problem which gained a large performance increase by code fragment guidance (from 36% to 85%). (Translated from the internal language into Java) In this solution, the GP uses a loop as a conditional (since a loop will only execute its instruction block if the bounding variable is positive). It must iterate through each value, and write the negative of the value to the output array if the input array’s value at that index is negative (thus rendering it positive). The GP accomplishes this by generating a negative, writing it to the output, then erasing it if the value read in was already positive, using a loop as a conditional. Inefficient but an effective solution to the problem as presented to it.

```
static int nArrays = 2;
static int nVars = 12;
public static int[] generatedProgram(
int[] inputArray,int param){
    int[][] arrays = new int[nArrays][];
    arrays[0] = inputArray;
    int[] variables = new int[nVars];
    variables[0] = inputArray.length;
    variables[1] = param;
    arrays[1] = new int[variables[0]]
    for (variables[2]=0;variables[2]<variables[0];variables[2]++){
        variables[3] = arrays[0][2]
        variables[4] = arrays[0][2]
        variables[5] = variables[6] - variables[3]
        arrays[1][i] = variables[5]
        for (variables[7]=0;variables[7]<variables[4];
        variables[7]++){
            arrays[1][i] = variables[4]
        }
    }
}
return arrays[1];
```

previous problems, such as read and writing to and from the input and output arrays. Further work would need to be done to analyse exactly how code changes over time, and fragment usage correlates and corresponds to these fitness changes, but this data is a useful confirmation of the expected overall behaviour.

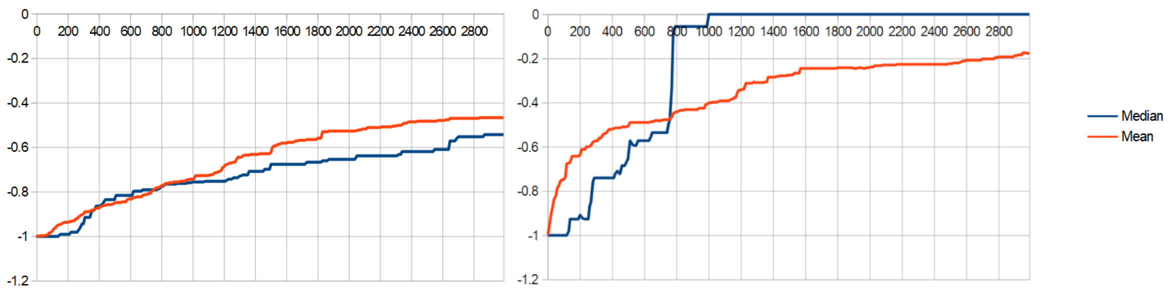


Figure 6.2: Fitness curves on the 21st problem of the 1D Array corpus, 'Keep Positives'. The left graph represents the median and mean from non-TL GP runs, with the right from the TL runs (median in blue, ends lower on the non-TL runs, higher on TL). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=30)

Figure 6.3 shows a problem from the second corpus, the 2D arrays/canvas set representing geometric images. This one also shows a stronger early trend, especially on the median graph, where the GP system spends time with fewer than 50% of the population elite individuals able to write anything usefully to the output array (fitness is at -1, indicating it has not achieved anything better than a program returning an all-zero 2D array would achieve). The TL approach by comparison has clearly provided the GP system with a good starting point. This problem is interesting, however, in that it had lower overall performance with the approach than the baseline GP, and this appears to be visible in the graphs, with the approach achieving a lower fitness towards the end of the run. It could be hypothesised that the approach transferred data which may initially have boosted fitness, but lead to the GP process becoming trapped in a local minimum which did not lead to useful progress towards a 0-fitness solution.

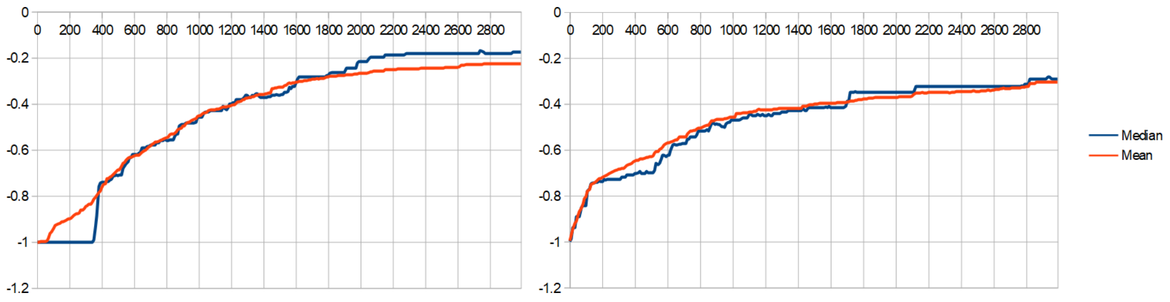


Figure 6.3: Fitness curves on the 6th problem of the 2D Array corpus, 'Mirrored Hollowed Parallelogram'. The left graph represents the median and mean from non-TL GP runs, with the right from the TL runs (median in blue, ends higher on the non-TL runs, TL median has more step-function-like jumps in the curve). Y is the fitness, which reaches 0 when the problem is solved, and is normalised to a fitness of -1 if the sampled program is returning an all-zero array. Penalty value of -10,000 changed to -1 to allow mean to be viewed usefully. X is generation count. (n=30)

Problem Set	**Mean gens. to threshold**	**Mean transfer Ratio**
1D-array	1439.76 (848.74)	0.75 (0.25)
2D-array / canvas	1414.12 (878.64)	0.62 (0.31)

Table 6.4: Average transfer learning metrics of generations-to-threshold and transfer ratio, for each of the two problem sets. Standard Deviation shown in parentheses

Beyond the evidence of learning enhancements shown by fitness curves, existing research into transfer learning has also suggested a set of specific metrics to help understand the effects of transferred information Taylor and Stone (2011). Considering the specific way in which transfer works in this work's approach, the most suitable metric from this set is the 'time to threshold', which reports how many generations it takes for the average fitness of the TL system to exceed the average fitness the baseline reaches after the full 3000 generations. As part of this measure we also report the 'transfer ratio', which computes the average fitness of the TL system divided by the average fitness of the baseline. We show the average values for this metrics across all problems in each problem set in Tables 6.4 (with a full per-problem breakdown provided in Appendix A.7). On both measures we see clear evidence of the benefits of transfer learning; the approach reaches the best fitness of the baseline in less than half the number of generations on average, and shows a clear relative improvement in fitness over the baseline. While the standard deviation is high, the results are replicated sufficiently consistently to form a clear pattern to support the statistically significant results seen previously.

6.2.4 Analysis of Large Mutations

Finally, this section examines whether *NN-suggested* specific fragments are likely to be the real source of improvements, or whether this may be simply a factor of inserting larger fragments of code (up to 4 lines at a time) in general. To do this, a follow-up experiment was run which simulates random large fragments of code being added as insert mutations. These fragments are not recommended by an NN, but instead are generated at random each time the 'fragment injection' mutation is called. They are all of maximum length (to avoid one-line fragments being generated which would be no different than the pre-existing 'inject' mutation which injects as single line of code).

Problem	**Rand Success**	**B' Success**	**Success**	**Fisher Sig.**
Add	46%	47%	83%	0.003
Append	33%	0%	93%	$3.300 * 10^{-6}$
CumulativeAbsoluteSum	0%	7%	40%	$2.522 * 10^{-4}$
CumulativeSum	8%	17%	70%	$3.636 * 10^{-6}$
KeepEvenIndices	8%	23%	100%	$3.536 * 10^{-13}$
ClipToMin	46%	53%	90%	$4.805 * 10^{-4}$
RetainSecondHalf	0%	80%	33%	0.001
Sort	0%	0%	0%	1.0
Subtract	71%	33%	83%	0.166
Abs	46%	37%	73%	0.028
GreaterThan	4%	10%	17%	0.142
IndexParity	100%	97%	100%	1.0
FirstElementOnly	100%	33%	100%	1.0
Identity	100%	97%	100%	1.0
DivergentSequence	4%	40%	93%	$6.202 * 10^{-12}$
Double	79%	47%	100%	0.013
ShiftRight	17%	0%	87%	$2.076 * 10^{-7}$
ShiftRightLossy	21%	60%	100%	$2.314 * 10^{-10}$
ShiftLeft	21%	17%	100%	$2.314 * 10^{-10}$
ShiftLeftZeroPadded	88%	67%	93%	0.333
RetainFirstHalf	13%	0%	57%	$7.547 * 10^{-4}$
LessThan	4%	13%	30%	0.014
Multiply	58%	53%	80%	0.055
Negative	75%	77%	100%	0.005
Pop	46%	20%	100%	$2.252 * 10^{-6}$
KeepPositives	38%	60%	100%	$1.510 * 10^{-7}$
KeepEvens	0%	0%	0%	1.0
ArrayLength	100%	97%	100%	1.0
ArrayToZero	100%	97%	100%	1.0
KeepNegatives	75%	60%	97%	0.023
KeepOdds	0%	0%	3%	1.0
Reverse	54%	43%	80%	0.032
CatToSelf	92%	10%	70%	0.041
CatZerosToSelf	100%	23%	100%	1.0
ClipToMax	42%	43%	90%	$1.688 * 10^{-4}$

Table 6.5: Comparison of random fragments (Rand) generated when the 'inject fragment' mutation is called and with the full TL results. As this is a secondary experiment, note the random fragments only were run for 24 runs. Statistical significance established per problem by Fisher's Exact Test. (n=24, n=30)

As can be seen in Table 6.5, there is a clear statistical difference between the use of randomly generated fragments and deployment of fragments by the NN, which were sourced from previous successes. The random-fragment approach used here achieved an average success rate of 46%, compared to the TL system's 76%; the use of random fragments is, however, slightly better than the performance of the baseline GP. Comparison to the baseline GP indicates that certain problems were benefited by this new mutation strategy. Specifically, a number of those which required the array length to be changed saw large improvements to their performance. This could be hypothesised to be a result of the fitness function, which does not have a shaped landscape towards the correct array length, but rather simply awards a penalty if the output array is of the wrong length. In this case, the fitness function would be unable to guide navigation through problem space, and the much larger jumps performed by this mutation operation appear to lead to success.

Statistical differences between the two were even larger in the second corpus, the 2D arrays representing geometric images, with the new baseline achieving a success rate of 31%, which is lower than those seen in the GP baseline, and far lower than those seen in the TL approach. This suggests that for this corpus the larger mutations were harmful, and that this approach strongly benefited from more targetted fragments being injected. Full results are seen in Tables 6.6, with eight problems demonstrating statistically significant improvements over this secondary baseline by the TL system.

It can be concluded from this that the larger mutations assisted on some problems (and harmed on others), but were not solely contributory towards the success seen on the full end-to-end transfer learning system.

Problem	**Rand**	**Baseline**	**TL**	**Fisher Sig.**
Square	100%	100%	100%	1.0
HollowSquare	100%	100%	100%	1.0
Parallelogram	4%	0%	7%	0.5
HollowParallelogram	0%	3%	7%	0.333
MirroredParallelogram	20%	37%	43%	0.045
MirroredHollowParallelogram	0%	30%	20%	0.020
RightTriangle	100%	100%	100%	1.0
HollowRightTriangle	96%	100%	100%	0.5
MirroredRightTriangle	48%	50%	100%	$3.583 * 10^{-6}$
HollowMirroredRightTriangle	28%	40%	90%	$2.318 * 10^{-6}$
InvertedRightTriangle	96%	97%	100%	0.5
HollowInvertedRightTriangle	28%	50%	80%	$1.147 * 10^{-4}$
InvertedMirroredRightTriangle	100%	100%	100%	1.0
Inv'HollowMirr'RightTriangle	100%	100%	100%	1.0
IsoceleseTriangle	0%	0%	7%	0.333
HollowIsoceleseTriangle	0%	3%	23%	0.010
InvertedIsoceleseTriangle	32%	50%	80%	$3.441 * 10^{-4}$
HollowInv'IsoceleseTriangle	24%	30%	50%	0.033
RectangleWithEmptyTrapezoid	0%	7%	7%	0.333
Inv'dRect'WithEmptyTrapezoid	0%	0%	23%	0.010
ObtuseTriangle	8%	13%	30%	0.037
HollowObtuseTriangle	8%	33%	53%	$3.028 * 10^{-4}$
MirroredObtuseTriangle	4%	0%	0%	0.5
MirroredHollowObtuseTriangle	0%	7%	7%	0.333
InvertedObtuseTriangle	0%	0%	3%	1.0
HollowInvertedObtuseTriangle	0%	3%	13%	0.083
InvertedMir'ObtuseTriangle	0%	0%	10%	0.166
HollowMir'Inv'ObtuseTriangle	0%	0%	7%	0.333
VShape	24%	60%	90%	$5.527 * 10^{-7}$
Trapezoid	0%	0%	7%	0.333

Table 6.6: Comparison of success rate for random fragments (Rand) generated when the 'inject fragment' mutation is called and with the full TL results. As this is a secondary experiment, note the random fragments only were run for 25 runs. Statistical significance established per problem by Fisher's Exact Test. (n=25, n=30)

6.3 Conclusion to Experiments

This chapter has covered the range of experiments which went into the development of a Transfer Learning system for Genetic Programming.

In Experiment 1 it initially assessed the problem of which programs to use to train neural networks. It demonstrated that training corpora for neural networks benefit from being structured. The combined system exploits this by having multiple networks trained on multiple corpora, focused around certain programmatic elements. While experiment 1 uses human-selected requirements, these training corpora derive from automatically extracted code fragments.

In Experiment 2 it then demonstrated genetic programming was an effective tool for producing human useful programs, with a strong baseline performance. This suggested that rather than attempt a pure neural solution, a hybrid approach may be the superior option. As a result, focus shifted towards how to combine the two, with a good candidate being the use of neural networks to guide Genetic Programming. This appeared to be a good direction as it leveraged the power of neural networks to estimate very high level features of the program (equivalent to the function calls seen in DeepCoder) and the navigational properties of GP which allowed large program spaces to be traversed. It also demonstrated that using evolved programs to train neural networks was superior than random sampling of program space, and as such the later work employed the GP's successful outputs as starting points for corpus generation, building around them by mutation and crossover to create the training corpora used for the neural networks.

In Experiment 3 a preliminary investigation was undertaken into how transfer learning could be accomplished. This started with the fundamentals, evaluating on a number of programs how a Genetic Programming algorithm could be provided with guidance, and assessing the degree of performance gain which could be obtained by the particular approach used in that experiment in as systematic a fashion as computational resources allowed. The capabilities of neural networks were then assessed, to gauge their reliability and abilities, and they were found to have use as estimators in many but not all cases. After that, the GP guidance and neural network predictive capabilities were tested in conjunction, and once shown successful and end-to-end system designed. This system showed positive results, but had a core limitation of only deploying a single code fragment, reducing the transfer learning abilities.

Finally, Experiment 4 allowed an analysis of a full multi-donor transfer-learning system. This combined the most successful elements of all previous experiments, and differed from the previous in that it allowed arbitrary numbers of code fragments to be employed by the Genetic Programming Algorithm. It demonstrated to a high degree of statistical significance that this TL-based approach out-performed a GP baseline, and that this was due to the TL component, not simply due to a more complex mutation structure.

Taken together, these experiments seem to provide convincing evidence that neither Neural Code Synthesis nor pure Genetic Programming should be pursued in isolation, but rather that a combined approach might be the more promising direction in the immediate future for general Turing-complete code synthesis.

Chapter 7

Conclusion

7.1 Introduction to Conclusion

This chapter aims to recapitulate the above work, and evaluate it in terms of its objectives, its successes and its limitations. Firstly, the research questions will be revisited, to answer the questions they posed in the introduction. After this, the future directions this work can be taken are assessed. The work's successes point towards many promising directions which could improve performance still further, and the architecture forms a foundation on which future architectures can be built to explore how hybrid neural-genetic systems can tackle the challenging problem of code synthesis. Finally, the work has some concluding remarks, to summarise the lessons learnt through this project.

7.2 Research Questions

Four research questions were posed at the start of this work. They are re-iterated below.

- What is the best way to generate artificial training data for a neural network for this particular field?
- Which specific form of hints, suggestions or assistance could a GP process be provided with to improve its performance?
- What performance gains could be achieved by an end-to-end system, which automatically trains NNs and uses them to guide a GP without need for human guidance?

7.2.1 What is the best way to generate artificial training data for a neural network for this particular field?

This work studied the question of how to best train a neural network using a synthetic dataset. Uniform sampling of program space is shown to be inferior to a more structured approach.

It was seen in Experiment 1 that adding structure to a synthetic dataset, by forcing it to contain certain proportions of desirable features (in that Experiment's case flow control operators), was a major advantage to neural network performance. It was seen, however, that guidance decisions were not equal, and certain choices, such as attempting to boost the proportion of longer programs in the training corpus, were not as effective as other approaches. This is in contrast to the simpler approach initially used in earlier papers of the literature which would draw from all of program space with equal likelihood, which could be hypothesised to not give sufficient focus to the more difficult to learn sections while wasting training time on the parts the network can easily learn.

In Experiment 2, the need for human guidance was removed as much as possible. This indicated that it is possible to automatically identify programs which would be useful to neural network training (with enough accuracy to be efficient). A neural discriminator was employed and trained in an adversarial fashion similar to that used in Generative Adversarial Networks (Goodfellow et al. (2014)), and lead to the creation of programs which produced complex structured outputs. The source code of these programs were seen to have a strong degree of similarity to those chosen by human designers in the previous example, leading confidence to the hypothesis that these features are representative to the 'human useful' programs targetted by this work.

7.2.2 Which specific form of hints, suggestions or assistance could a GP process be provided with to improve its performance?

This work attempted to answer this question in as great a depth as possible. The range of possible hints and assistance which could be given to a Genetic Process is vast, and exhaustive evaluation to systematically evaluate even one particular approach took months of computational time. Nevertheless, within the scope of these limitations, this work evaluate how providing certain fragments of the correct solution could guide a Genetic Programming algorithm, and what degree of success could be achieved. Certain patterns were seen, indicating that fragments containing literals for example were more effective at guiding the GP process than fragments containing loops.

To be useful, these fragments would need to be deployed automatically. It was shown that a neural network was able to estimate to some degree the types of programs which contain these potentially useful fragments, based purely on their input-output behaviour. The NN's alternate approach to solving the problem of program synthesis can allow two 'perspectives' on any problem to be employed. The fragment system chosen would therefore seem to be a strong addition to existing GP processes, and can serve to boost the GP's success rates reliably, with low probabilities of reduction in performance.

7.2.3 What performance gains could be achieved by an end-to-end system, which automatically trains NNs and uses them to guide a GP without need for human guidance?

The results achieved by the approach studied in this thesis suggest that strong performance gains, up to doubling the success rate, can be achieved. These experiments employed were intended to represent two distinct problem sets with as few assumptions made as possible. One core assumption is that the system will be exposed to multiple problems in a sequential fashion, and that these problems will have a degree of commonality between them (specifically it must be able to re-use lines of code directly). This assumption is not true for all use-cases of a code synthesis system, but would appear to be one which holds true for a good proportion of conceivable industrial uses. In cases in which individual users of the system may not require more than one or two programs synthesised per year, such as non-expert users employing the capability in a spreadsheet software package, these users could leverage cloud processing and data-sharing. In that situation, each user's successfully completed program could be anonymised and sent back to a centralised server, which would generate training corpora and train the network, which could then be distributed to all users during periodic updates. Since the program can be used to extract code-fragments the user's personal data need never be shared with the central server, protecting anonymity and data-privacy.

If these assumptions are met, the advantage granted by this approach is increased find rates. This took two forms, firstly the increase performance on problems which the GP had a non-zero but not perfect find rate; and secondly on problems the GP never produced a functional solution to. Not only were the problems found faster and more reliably, but certain problems which were 'unsolvable' as far as experimental precision was able to determine became solvable.

The use of two corpora with different input-output styles and different data-type usages was intended to verify the system's generality, and its performance was similar on the two domains. This suggests the approach is indeed broadly applicable, and could perform well in yet further domains with differing data types. Naturally, these might require changes to the neural network's featurisation of the input-output examples, but the overall architecture would remain unchanged.

7.3 Limitations

While the work was as exhaustive as possible, limits of computation resources, especially relative to the large bodies of work needed for a clear picture, leave room for future work. Many tasks were computationally extremely expensive, for instance the need for combinatorial breakdowns of every way to present fragments to a GP. Further work on different types of guidance for the GP, and a more in-depth analysis of how each type of fragment affected the GP's performance would have been beneficial. However, the existing work consumed months of computational resources, and additional time was not available.

More work should also have been done to establish a solid statistical footing for the results. Statistical significance testing was only done in the final experiment, and loss of data means that previous chapters could not be re-done. This limits the usefulness of these findings, and their reliability. However, as they did not lead to demonstrably statistically significant results, and were not the End-to-End system in and of themselves, they could be considered extended preliminary testing. They were of sufficient quality to meet peer review standards, despite this lack of statistical analysis.

The work also would benefit from being compared against the programs in the General Program Synthesis Benchmark Suite, to allow better evaluation of its performance in context. This is discussed further below.

7.4 Future Work

A range of directions suggest themselves based on the work presented in this thesis. The architecture proposed is not intended to be a completed system which cannot be improved upon, but rather to be an exploration of what is possible by combining multiple code synthesis techniques into a single algorithm. As such, it welcomes future expansion.

7.4.1 Alternate Guidance

Currently the neural networks are able to provide guidance to the genetic programming algorithm by means of a library of mutations, derived from previously generated programs. Each of these mutations has a weighting associated to it, based on the neural network's estimate of the probability the associated code fragment is present in a solution to the problem it is faced with. It would seem logical, therefore, to expand the system to include neural-network generated weighting for all mutations, not simply the code-fragment-injection mutations. Of specific interest would be the code insertion operations, and the single-element mutation operations. These could be supplied a non-uniform probability mapping across the various operators which could be inserted, for example to bias away from multiplication and division and towards addition if the neural networks consider this an advantageous bias.

This would seem a logical expansion, as it would simply be applying a technique which has been shown to be effective, specifically biasing which code elements are inserted by the code-fragment-insertion mutation function, to all insertion mutations, not simply those associated with the Transfer Learning element.

7.4.2 Expansion of Language

A very core element missing from the language as it was implemented is the ability to generate new constants for the programs to use as literal values. The literal values which were able to be accessed were the integers -1,0,1,2. Every other numeric value which the system required had to be generated by arithmetic functions on these values.

To expand on this, the system would need the ability to store literal values into the program's source code, to mutate these values, and possibly to extract them from the input-output examples. Literal values would potentially need to be able to cover not just integers, but a much wider range of data-types, with two immediately obvious inclusions being floating point values and character strings.

Other than literal values, the ability to deal with floating points and string operations would appear to be a major missing feature, whose inclusion could allow far greater comparison of the system against existing works, as it would allow it to face the exact same problems they did, to evaluate comparative performance.

7.4.3 Functional Encapsulation

Programs these days are far more complex than those of previous decades, but this is not due to human programmers having greater productivity in terms of lines of code manually written, but rather due to leveraging libraries of functions and exploiting higher-level languages. In transferring learnt capabilities from one problem to another, it may be beneficial to allow machines to do the same. To accomplish this goal the system would need the ability to isolate code fragments which are entirely self contained, which it already attempts to, which are usefully re-usable as immutable functions, which they currently may not be.

In individuality, encapsulated functions are less powerful than full code fragment injection, as they cannot be mutated by future genetic programming operators and cannot have their local variables read by lines of code to adapt their functionality in ways which a human designer might not consider them 'intended for'. This lead to this particular work avoiding that line of approach, and preferring the greater flexibility of simply injecting full code fragments into existing programs in a fashion which allows them to be later edited and mutated and all intermediary variables available for use by later additions. Taken together, however, a library of functions could reduce the number of mutations required to reach the target program, and greatly increase the power of the language.

7.4.4 Data Preprocessing

Neural networks have powerful high-level processing capabilities, but they are only capable of imitating Turing-Completeness, to a limit defined by their capacity and quantity and coverage of training data. Certain functions are trivial in general purpose programming languages which would prove difficult to replicate reliably in neural networks. For example, normalising an array such that its highest value is 1.0 and its lowest 0. This function can readily be written and relied upon to work up to extremely high values, or values extremely close to 0. Conversely, neural networks degrade as the values they encounter exceed those seen in training data, and the ranges of values possible in double-precision floating point may require a large proportion of the network's capacity to effectively handle.

Such a function may reveal useful properties about the input-output examples, but a neural network may find it difficult to allocate portions of its capacity to replicating the functionality then exploiting the information revealed. Instead, it may be beneficial to attempt re-training of neural networks in a fashion which concatenates additional information to the representation the neural network receives. This would be done by pre-processing the values in both the input and output examples, passing them through genetic-programming generated functions, and providing their outputs to the neural networks as additional information.

This was attempted and seen to be of moderate effectiveness during preliminary experiments surrounding this work's Experiment 1, but was not included due to the risk that the perception would be that the human-useful corpus had been chosen specifically for this outcome to be possible, reducing confidence in the neural network's general synthesis capabilities. Further work would need a better human-useful corpus to be selected, ideally one which was already accepted by the community, such as the General Program Synthesis Benchmark Suite Helmuth and Spector (2015).

7.4.5 Applications of Code Synthesis to the field of machine learning

One interesting avenue of investigation is how code synthesis can support machine learning. Code synthesis is, from one perspective, a way to generate a symbolic model of a system's behaviour. That system is the problem, and its behaviour is the transformation of input to output. While in this case the system the code synthesis system is attempting to model is another program, it does not necessarily need to be that way.

If code synthesis were deployed on data from other sources, such as economic or environmental data, the system would attempt to build a symbolic representation of the behaviour it sees. Representing behaviour in a symbolic fashion, as opposed to a connectionistic fashion as neural networks do, has the key advantage that the model can be easily examined, humanly interpreted, and compared easily to other generated programs.

It is in this comparison to other generated programs that the advantage comes to light. Systems which may appear extremely different, in terms of their nature, in terms of the scales they exist on or in terms of their physical appearance, may have behaviours which appear very similar once described in code. For example, the phases of the moon, a pendulum and a vibrating crystal all can be modelled by a simple periodic sine wave. Despite being physically extremely different, and existing on vastly different time scales (differing by many orders of magnitude), if a code synthesis system were deployed to model them, and their respective models examined, a commonality could easily be seen in the structure of the program used to model them. The programs would differ in terms of constants used, not in terms of operators or program flow.

As such, second-order evaluation of systems, by firstly modelling them and then evaluating them based on this generated model, could allow far higher order behaviours to be made possible in the realm of machine learning. The example above is one of extreme invariance to scale, as a classifier could be trained on the programs used to model smaller time-scale systems and readily recognise the code for a lunar calendar as being extremely similar, but it also represents a form of reasoning by analogy. No physical descriptions need to be considered, only how their behaviour is best represented.

This is an intriguing area for potential future investigation.

7.5 Concluding Comments

This thesis has been an exploration of the field of code synthesis. Its goal was to evaluate the separate sub-fields, and attempt to bring them together into a single whole which takes the best from all. It succeeded in producing a hybrid system, which drew from two of the most effective sub-fields, those of genetic programming and of neural code synthesis, but failed to incorporate elements of deductive solver systems. In its approach to hybridisation, while it proved effective, it is by no means the only way in which these two tools could be combined, and the work concludes with the hope of being the first of many such approaches, some of which may differ quite majorly from the approach outlined above. Code synthesis is an important yet exceedingly difficult problem, and it would be unwise for any approaching it to ignore potential assistance from other works in the literature, even if those works appear to be employing entirely different technologies.

Appendix A

Appendix: Additional Tables

A.1 Problems from Experiment 2

These are the problems employed in Experiment 2

Table A.1: Human useful programs, specified by user and presented to the system as a set of I/O mappings only.

Program	**Function**
Absolute Values	Returns an array of equal size to the input array, where all values are the absolute values in the input array
Array Length	Returns a 1-length array, whose value is the length of the input array
Array to Zero	Returns an array of equal length to the input array, filled with zeroes
Cumulative Absolute Sum	Returns an array of equal length to the input array, filled with the cumulative absolute values. That is to say cell $output_i = \sum_{n=0}^{i} \lvert input_n \rvert$
Cumulative Sum	Returns an array of equal length to the input array, filled with the cumulative values. That is to say cell $output_i = \sum_{n=0}^{i} input_n$
Divergent Sequence	Returns an array of equal length to the input array, filled with values such that if i is even $output_i = i/2$, otherwise $output_i = -(i/2)$, e.g. [0,0,1,-1,2,-2...]
First Element Only	Returns an array of length 1, whose content is $input_0$

Table A.2: Continuation of table A.1

Identity	Returns a new array with length and contents matching the input array
Index Parity	Returns a new array with length equal to that of the input array, filled with alternating 0s and 1s, e.g. [0,1,0,1...]
Iterative Difference	Returns a new array with length equal to that of the input array, filled with the difference between the mapped input array cell and the next, such that $output_i = input_i - input_{i-1}$, with $output_0 = input_0$
Keep Evens	Returns an array of equal length to that of the input array. If $input_i$ is even, $output_i = input_i$ else $output_i = 0$
Keep Negatives	Returns an array of equal length to that of the input array. If $input_i < 0$, $output_i = input_i$ else $output_i = 0$
Keep Odds	Returns an array of equal length to that of the input array. If $input_i$ is odd, $output_i = input_i$ else $output_i = 0$
Keep Positives	Returns an array of equal length to that of the input array. If $input_i > 0$, $output_i = input_i$ else $output_i = 0$
Negative	Returns the input array multiplied by -1. $output_i = -input_i$
Pop from Array	Returns an array one element shorter than the input. Values are the input array's values without the last.
Reduce Length by Half	Returns an array half the length of the input array. Values in it are the first values of the input array.

Table A.3: Continuation of table A.1

Program	**Function**
Reverse	Returns a copy of the input array, but with the elements in reverse order
Shift Left	Returns an array of length one less than the input array. Values are shifted, such that $output_i = input_{(i+1)}$. The first input value is therefore not replicated in the output.
Shift Left Zero Padded	Returns an array of length equal to the input array. Values are shifted, such that $output_i = input_{(i+1)}$. The last value of the output array is set to zero.
Shift Right	Returns an array of length one greater than that of the input array. Values are the values in the input array preceded by a zero.
Shift Right Lossy	Returns an array of length equal to that of the input array. Values are such that $output_i = input_{(i-1)}$, with the first output value being zero.
Shuffle Zeros to Back	Returns an array of length equal to that of the input array. Values are the the same elements as in the input array, with the exception that all zero-values are moved to the end of the sequence.
Sign of	Returns an array of length equal to that of the input array. Values in output -1, 0 or 1. If $input_i > 0$ then $output_i = 1$, if $input_i < 0$ then $output_i = -1$, else $output_i = 0$
Sort	Returns a copy of the input array, sorted in ascending order.
Square Values	Returns an array of length equal to that of the input array. Values are such that $output_i = (input_i)^2$
To Iterator	Returns an array of length equal to that of the input array. Values are such that $output_i = i$

Table A.4: Continuation of table A.1. The programs in this table are those which use both the input array and the input integer.

Program	Function
Add	Returns an array of length equal to that of the input array. Values are such that $output_i = input_i + inputInteger$
Append	Returns an array of length one greater than that of the input array. Values are those of the input array, followed by the input integer.
Clip to Max	Returns an array of length equal to that of the input array. Values are such that if $input_i > inputInteger$ then $output_i = input_i$ else $output_i = inputInteger$
Clip to Min	Returns an array of length equal to that of the input array. Values are such that if $input_i < inputInteger$ then $output_i = input_i$ else $output_i = inputInteger$
Constant Addition	Returns an array of length equal to that of the input array. Values are such that $output_i = input_i + (i * inputInteger)$

Table A.5: Continuation of table A.1. The programs in this table are those which use both the input array and the input integer.

Fill Array	Returns an array of length equal to that of the input array. Values are such that $output_i = inputInteger$
Greater Than	Returns an array of length equal to that of the input array. Values are such that if $input_i > inputInteger$ then $output_i = 1$ else $output_i = -1$
Iterate from Start	Returns an array of length equal to that of the input array. Values are such that $output_i = i + inputInteger$
Less Than	Returns an array of length equal to that of the input array. Values are such that if $input_i < inputInteger$ then $output_i = 1$ else $output_i = -1$
Multiples of	Returns an array of length equal to that of the input array. Values are such that $output_i = i * inputInteger$
Subtract	Returns an array of length equal to that of the input array. Values are such that $output_i = input_i - inputInteger$

A.2 Examples of 2nd Corpus Programs

```
Parallelogram                Rectangle with Empty Trapezoid
00000000     ________        11110000     ####____
00010000     ___#____        11100000     ###_____
00110000     __##____        11000000     ##______
01110000     _###____        10000000     #_______
11110000     ####____        10000000     #_______
11100000     ###_____        11000000     ##______
11000000     ##______        11100000     ###_____
10000000     #_______        11110000     ####____

Hollow Isosceles Triangle    Obtuse Triangle
00001000     ____#___        10000000     #_______
00011000     ___##___        10000000     #_______
00101000     __#_#___        11000000     ##______
01001000     _#__#___        11000000     ##______
10001000     #___#___        01100000     _##_____
01001000     _#__#___        00100000     __#_____
00101000     __#_#___        00010000     ___#____
00011000     ___##___        00000000     ________
```

Figure A.1: Examples from the second, 2D boolean, corpus. Each program is first illustrated as the numeric outputs in the array, and then in an ASCII art rendering to better illustrate the shape being generated.

A.3 Experiment 3: Full Fragment Guidance Breakdown

This section presents the results of every fragment presented to every problem tested in Experiment 3.1

Line	Operator	As Code
1	Literal	variables[6] = 1;
2	Add	variables[7] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[7]]
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Read	variables[5] = arrays[0][variables[2]];
6	Write	arrays[1][variables[2]] = variables[5];
7	Endloop	
8	Write	arrays[1][variables[2]] = variables[1];

Fragment	Success Rate
1	3%
1, 2	27%
1, 4	10%
4	0%
4, 5	0%
4, 6	0%
4, 8	0%

Table A.6: Fragments assessed from program “Append". Program’s code listed, in C-like format, with operators listed ahead of each line for ease of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make Array	arrays[1] = new int[vars[0]]
2	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
3	Literal	variables[5] = -1;
4	Read	variables[3] = arrays[0][variables[2]];
5	Condition	if (variables[3]>0)
6	Else	else
7	Multiply	variables[3] = variables[3] * variables[5];
8	Endloop	
9	Add	variables[4] = variables[4] + variables[3];
10	Write	arrays[1][variables[2]] = variables[4];
11	Endloop	

Fragment	Success Rate
1	0%
1, 2	0%
1, 3	0%
1, 5	0%
1, 6	0%
2	0%
2, 3	0%
2, 4	3%
2, 5	3%
2, 6	0%
3	0%
3, 5	0%
3, 6	0%
3, 7	0%
5	0%
5, 6	3%
6	0%

Table A.7: Fragments assessed from program “Cumulative Absolute Sum". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[4] = 2;
2	Make Array	arrays[1] = new int[vars[0]]
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Read	variables[3] = arrays[0][variables[2]];
5	Modulo	variables[5] = variables[3] % variables[4];
6	Condition	if (variables[5]==variables[6])
7	Write	arrays[1][variables[2]] = variables[3];
8	Endloop	
9	Endloop	

Fragment	Success Rate
1	0%
1, 2	6%
1, 3	3%
1, 5	3%
2	3%
2, 3	0%
3	0%
3, 4	0%
3, 7	0%

Table A.8: Fragments assessed from program “Keep Evens". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 2;
2	Divide	variables[3] = variables[0] / variables[6];
3	Make Array	arrays[1] = new int[vars[3]]
4	Loop	for (variables[2]=0;variables[2]<variables[3];variables[2]++)
5	Read	variables[5] = arrays[0][variables[2]];
6	Write	arrays[1][variables[2]] = variables[5];
7	Endloop	

Fragment	Success Rate
1	0%
1, 2	13%

Table A.9: Fragments assessed from program “Retain First Half". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[7] = 2;
2	Make Array	arrays[1] = new int[vars[0]]
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Subtract	variables[6] = variables[0] - variables[2];
5	Subtract	variables[6] = variables[6] - variables[7];
6	Read	variables[5] = arrays[0][variables[6]];
7	Write	arrays[1][variables[2]] = variables[5];
8	Endloop	

Fragment	Success Rate
1	63%
1, 2	80%
1, 3	77%
2	73%
2, 3	60%
3	63%
3, 4	80%
3, 7	77%

Table A.10: Fragments assessed from program “Reverse". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 1;
2	Add	variables[8] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[8]]
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Add	variables[7] = variables[2] + variables[6];
6	Read	variables[5] = arrays[0][variables[2]];
7	Write	arrays[1][variables[7]] = variables[5];
8	Endloop	

Fragment	Success Rate
1	3%
1, 2	13%
1, 4	20%
4	0%
4, 6	0%

Table A.11: Fragments assessed from program “Shift Right". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Literal	variables[6] = 2;
2	Add	variables[8] = variables[0] + variables[6];
3	Make Array	arrays[1] = new int[vars[0]]
4	Subtract	variables[9] = variables[0] - variables[6];
5	Loop	for (variables[2]=0;variables[2]<variables[9];variables[2]++)
6	Add	variables[7] = variables[2] + variables[6];
7	Read	variables[5] = arrays[0][variables[2]];
8	Write	arrays[1][variables[7]] = variables[5];
9	Endloop	

Fragment	Success Rate
1	80%
1, 2	73%
1, 3	63%
1, 4	63%
3	67%

Table A.12: Fragments assessed from program “Shift Right Lossy". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	**Operator**	**As Code**
1	Literal	variables[5] = 1;
2	Subtract	variables[1] = variables[0] - variables[5];
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Loop	for (variables[3]=0;variables[3]<variables[1];variables[3]++)
5	Add	variables[6] = variables[3] + variables[5];
6	Read	variables[4] = arrays[0][variables[3]];
7	Read	variables[7] = arrays[0][variables[6]];
8	Subtract	variables[8] = variables[4] - variables[7];
9	Condition	if (variables[8]>0)
10	Write	arrays[0][variables[6]] = variables[4];
11	Write	arrays[0][variables[3]] = variables[7];
12	Endloop	
13	Endloop	
14	Endloop	
15	Make Array	arrays[1] = new int[vars[0]]
16	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
17	Read	variables[5] = arrays[0][variables[2]];
18	Write	arrays[1][variables[2]] = variables[5];
19	Endloop	

Fragment	**Success Rate**
1	0%
1, 2	0%
1, 3	0%
1, 8	0%
1, 15	0%
1, 16	0%
3	0%
3, 8	0%
3, 15	0%
3, 16	0%
3, 17	0%
4	0%
4, 6	0%
8	0%
8, 9	0%
8, 15	0%
8, 16	0%
15	0%
15, 16	0%
16	0%

Table A.13: Fragments assessed from program “Sort". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	**Operator**	**As Code**
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[6] = 2;
3	Divide	variables[4] = variables[0] / variables[6];
4	Loop	for (variables[2]=0;variables[2]<variables[4];variables[2]++)
5	Loop	for (variables[3]=0;variables[3]<variables[4];variables[3]++)
6	Add	variables[7] = variables[2] + variables[3];
7	Write to 2D	array[variables[7][variables[3]]=1;
8	Endloop	
9	Endloop	

Fragment	**Success Rate**
2	23%
2, 3	30%

Table A.14: Fragments assessed from program "Mirrored Parallelogram". Program's code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[6] = 2;
3	Divide	variables[4] = variables[0] / variables[6];
4	Loop	for (variables[2]=0;variables[2]<variables[4];variables[2]++)
5	Add	variables[5] = variables[2] + variables[4];
6	Write to 2D	array[variables[5][variables[10]]=1;
7	Write to 2D	array[variables[2][variables[4]]=1;
8	Subtract	variables[6] = variables[4] - variables[2];
9	Write to 2D	array[variables[2][variables[6]]=1;
10	Write to 2D	array[variables[5][variables[6]]=1;
11	Endloop	
12	Literal	variables[8] = 1;
13	Subtract	variables[7] = variables[0] - variables[8];
14	Write to 2D	array[variables[7][variables[10]]=1;

Fragment	Success Rate
2	13%
2, 3	60%
2, 12	10%
12	13%
12, 13	40%

Table A.15: Fragments assessed from program “Mirrored Hollow Parallelogram". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[1] = 1;
3	Subtract	variables[4] = variables[0] - variables[1];
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Write to 2D	array[variables[2][variables[4]]=1;
6	Write to 2D	array[variables[5][variables[2]]=1;
7	Write to 2D	array[variables[2][variables[2]]=1;
8	Endloop	

Fragment	Success Rate
2	80%
2, 3	90%
2, 4	80%
4	90%
4, 6	63%
4, 7	87%

Table A.16: Fragments assessed from program “Hollow Right Triangle". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[3] = 1;
3	Subtract	variables[4] = variables[0] - variables[3];
4	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
5	Write to 2D	array[variables[2][variables[4]]=1;
6	Write to 2D	array[variables[4][variables[2]]=1;
7	Subtract	variables[5] = variables[0] - variables[2];
8	Subtract	variables[5] = variables[5] - variables[3];
9	Write to 2D	array[variables[2][variables[5]]=1;
10	Endloop	

Fragment	Success Rate
2	67%
2, 3	93%
2, 4	80%
4	67%
4, 7	80%

Table A.17: Fragments assessed from program “Hollow Mirrored Right Triangle". Program's code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[4] = 2;
3	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
4	Multiply	variables[6] = variables[2] * variables[4];
5	Subtract	variables[5] = variables[0] - variables[6];
6	Loop	for (variables[3]=0;variables[3]<variables[5];variables[3]++)
7	Add	variables[7] = variables[3] + variables[2];
8	Write to 2D	array[variables[7][variables[2]]=1;
9	Endloop	
10	Endloop	

Fragment	Success Rate
2	23%
2, 3	20%
3	20%

Table A.18: Fragments assessed from program “Inverted Isoceles Triangle". Program’s code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

Line	Operator	As Code
1	Make 2D Array	new 2DArray(size=variables[0]);
2	Literal	variables[7] = 1;
3	Literal	variables[4] = 2;
4	Divide	variables[5] = variables[0] / variables[4];
5	Loop	for (variables[2]=0;variables[2]<variables[0];variables[2]++)
6	Loop	for (variables[3]=0;variables[3]<variables[5];variables[3]++)
7	Subtract	variables[8] = variables[0] - variables[5];
8	Divide	variables[8] = variables[8] / variables[4];
9	Subtract	variables[8] = variables[8] - variables[3];
10	Add	variables[9] = variables[8] + variables[7];
11	Condition	if (variables[9]>0)
12	Subtract	variables[9] = variables[2] - variables[8];
13	Condition	if (variables[9]>0)
14	Subtract	variables[9] = variables[0] - variables[8];
15	Subtract	variables[9] = variables[9] - variables[2];
16	Condition	if (variables[9]>0)
17	Write to 2D	array[variables[2][variables[3]]=1;
18	Endloop	
19	Endloop	
20	Endloop	
21	Endloop	
22	Endloop	

Fragment	Success Rate
2	10%
2, 3	10%
2, 5	10%
3	3%
3, 4	0%
3, 5	10%
5	3%

Table A.19: Fragments assessed from program "Trapezoid". Program's code listed, in C-like format, with operators listed ahead of each line for each of readability. Fragments then described, in reference to lines used followed by success rate using fragment as GP guidance (n=30)

A.4 Full results of NN-selected guidance for GP from Experiment 3

Tables A.20,A.21,A.22 shows the success rate of the GP, if provided with hints by the neural network sets. Two sets of experiments are done on the array to array corpus, one on the canvas corpus. Most problems showed a success increase, including a number from both corpora which increased in success chance from 0% to a non-zero value. Two problems were made unfindable by the less-effective uniform fragment selection process, Iterative Difference and Trapezoid. As the baseline find rate was low, this does not represent a major drop in success, and may potentially simply be due to insufficient samples to determine the true success probability. We do not however reject the possibility that our approach has a negative effect on find-rates for certain problems. It was seen in Experiment 1 that some fragments, known to be present in the ground-truth implementation, decreased find-rates. It is possible that the neural networks correctly identified fragment presence, but that these degraded the GP's performance. It is, of course, also possible that the NN incorrectly estimated that a fragment was present when it was not, and that this erroneous hint harmed the GP.

Problem	Success Rate	Baseline
Abs	73%	7%
ArrayLength	100%	100%
ArrayToZero	100%	100%
CumulativeAbsoluteSum	0%	0%
CumulativeSum	33%	3%
DivergentSequence	63%	57%
FirstElementOnly	100%	27%
Identity	100%	100%
IndexParity	100%	100%
IterativeDifference	3%	3%
KeepEvens	0%	0%
KeepNegatives	27%	0%
KeepOdds	10%	0%
KeepPositives	47%	60%
Negative	67%	67%
Pop	100%	30%
RemoveFirstElement	83%	10%
RetainFirstHalf	0%	0%
Reverse	77%	57%
ShiftLeft	80%	10%
ShiftLeftZeroPadded	83%	40%
ShiftRight	17%	0%
ShiftRightLossy	77%	83%
ShuffleZerosToBack	80%	100%
Signum	10%	0%
Sort	3%	0%
SquareValues	77%	70%
ToIterator	100%	100%
Add	37%	23%
Append	47%	0%
ClipToMax	43%	17%
ClipToMin	70%	3%
ConstantAddition	7%	0%
FillArray	100%	100%
GreaterThan	23%	10%
IterateFromStart	100%	97%
LessThan	17%	7%
MultiplesOf	87%	90%
Multiply	30%	20%
Subtract	43%	17%

Table A.20: Success rates for problems of the 1st corpus, using the rarest-first guidance strategy, with the aggregate estimates from the feed-forward architecture network (n=30)

Problem	**Success Rate**	**Baseline**
Abs	27%	7%
ArrayLength	100%	100%
ArrayToZero	100%	100%
CumulativeAbsoluteSum	0%	0%
CumulativeSum	20%	3%
DivergentSequence	83%	57%
FirstElementOnly	83%	27%
Identity	100%	100%
IndexParity	100%	100%
IterativeDifference	0%	3%
KeepEvens	0%	0%
KeepNegatives	53%	0%
KeepOdds	10%	0%
KeepPositives	40%	60%
Negative	63%	67%
Pop	73%	30%
RemoveFirstElement	57%	10%
RetainFirstHalf	0%	0%
Reverse	53%	57%
ShiftLeft	43%	10%
ShiftLeftZeroPadded	63%	40%
ShiftRight	0%	0%
ShiftRightLossy	53%	83%
ShuffleZerosToBack	77%	100%
Signum	10%	0%
Sort	0%	0%
SquareValues	77%	70%
ToIterator	100%	100%
Add	23%	23%
Append	7%	0%
ClipToMax	33%	17%
ClipToMin	20%	3%
ConstantAddition	3%	0%
FillArray	100%	100%
GreaterThan	13%	10%
IterateFromStart	97%	97%
LessThan	30%	7%
MultiplesOf	93%	90%
Multiply	30%	20%
Subtract	17%	17%

Table A.21: Success rates for problems of the 1st corpus, using the uniform guidance strategy, with the aggregate estimates from the feed-forward architecture network (n=30)

Square	90%	97%
HollowSquare	100%	100%
Parallelogram	23%	0%
HollowParallelogram	7%	0%
MirroredParallelogram	43%	7%
MirroredHollowParallelogram	17%	13%
RightTriangle	93%	97%
HollowRightTriangle	97%	87%
MirroredRightTriangle	50%	60%
HollowMirroredRightTriangle	60%	63%
InvertedRightTriangle	67%	60%
HollowInvertedRightTriangle	57%	83%
InvertedMirroredRightTriangle	97%	100%
InvertedHollowMirroredRightTriangle	100%	100%
IsoceleseTriangle	10%	0%
HollowIsoceleseTriangle	43%	13%
InvertedIsoceleseTriangle	37%	47%
HollowInvertedIsoceleseTriangle	33%	50%
RectangleWithEmptyTrapezoid	3%	3%
InvertedRectangle	10%	3%
obtuseTriangle	23%	3%
hollowObtuseTriangle	53%	27%
mirroredObtuseTriangle	0%	0%
mirroredHollowObtuseTriangle	0%	0%
invertedObtuseTriangle	0%	0%
hollowInvertedObtuseTriangle	17%	10%
invertedMirroredObtuseTriangle	7%	0%
hollowMirroredInvertedObtuseTriangle	7%	3%
VShape	40%	47%
Trapezoid	0%	7%

Table A.22: Success rates for problems of the 2nd corpus, using the uniform guidance strategy, with the aggregate estimates from the convolutional architecture network (n=30)

A.5 Experiment 3 Baseline Success Rates

This section details the find rates of programs using the GP process, and the same process with its Diversity Boosting Repetition Penalty component disabled. These results are used to guide the Experiment in section 5.1.9

Note that these baselines are used for Sections 5.1.8,5.1.9,5.1.10 only, 6 employs a newly computed baseline.

Problem	**Baseline Success Rate**	**GP without NS**
Array Length	100 %	100 %
Array to Zero	100 %	100 %
First Element Only	40 %	50 %
Identity	95 %	100 %
Negative	55 %	45 %
Double	55 %	45 %
Add	15 %	35 %
Multiply	50 %	30 %
Subtract	20 %	20 %
Absolute	35 %	20 %
Keep Negatives	35 %	55 %
Keep Positives	30 %	40 %
Keep Evens	0 %	0 %
Keep Odds	0 %	0 %
Index Parity	95 %	95 %
Keep Even Indices	30 %	10 %
Greater Than	15 %	10 %
Less Than	5 %	10 %
Clip to Max	25 %	35 %
Clip to Min	40 %	15 %
Sort	0 %	0 %
Shift Left	5 %	10 %
Shift Left Zero Padded	30 %	45 %
Shift Right	0 %	0 %
Shift Right Lossy	60 %	15 %
Reverse	25 %	40 %
Pop	10 %	15 %
Concatenate To Self	15 %	30 %
Concatenate Equal Length Zeros	10 %	25 %
Retain First Half	0 %	0 %
Retain Second Half	0 %	0 %
Append	0 %	0 %
Cumulative Absolute Sum	0 %	0 %
Cumulative Sum	5 %	15 %
Divergent Sequence	50 %	10 %

Table A.23: Success rates for a Genetic Algorithm on the problems of the first corpus, a set of functions which take an array of integers and an integer variable and return an array of integers. The first pass is the standard configuration used throughout the paper, using a diversity boosting implementation, the second is the same GP without the diversity boosting component to test the effects of NS on success rates. n=20

Problem	**Baseline Success Rate**	**GP without NS**
Square	100 %	100 %
Hollow Square	100 %	100 %
Parallelogram	0 %	0 %
Hollow Parallelogram	0 %	0 %
Mirrored Parallelogram	0 %	11 %
Mirrored Hollow Parallelogram	30 %	11 %
Right Triangle	100 %	89 %
Hollow Right Triangle	90 %	100 %
Mirrored Right Triangle	30 %	33 %
Hollow Mirrored Right Triangle	25 %	25 %
Inverted Right Triangle	60 %	88 %
Hollow Inverted Right Triangle	35 %	13 %
Inverted Mirrored Right Triangle	95 %	100 %
Inverted Hollow Mirrored Right Triangle	100 %	88 %
Isocelese Triangle	0 %	0 %
Hollow Isocelese Triangle	5 %	0 %
Inverted Isocelese Triangle	40 %	13 %
Hollow Inverted Isocelese Triangle	15 %	0 %
Rectangle With Empty Trapezoid	0 %	0 %
Inverted Rectangle With Empty Trapezoid	10 %	0 %
Obtuse Triangle	10 %	0 %
Hollow Obtuse Triangle	15 %	0 %
Mirrored Obtuse Triangle	0 %	0 %
Mirrored Hollow Obtuse Triangle	0 %	0 %
Inverted Obtuse Triangle	0 %	0 %
Hollow Inverted Obtuse Triangle	10 %	0 %
Inverted Mirrored Obtuse Triangle	0 %	14 %
Hollow Mirrored Inverted Obtuse Triangle	0 %	0 %
V Shape	45 %	67 %
Trapezoid	0 %	0 %

Table A.24: Success rates for a Genetic Algorithm on the problems of the second corpus, the set of functions which take an integer size for the returned canvas, and must return the shape specified. The first pass is the standard configuration used throughout the thesis, using a diversity boosting implementation, the second is the same GP without the diversity boosting component to test the effects of NS on find rates. n=20

A.6 Fragments evaluated for NN recognisability in Experiment 2

The tablesA.25A.26 describe the fragments (some of which contain requirements about variable dependencies) used in experiment 2.

Fragment	Description
Add	Simple addition. Requires the program to at some point contain an addition operation
+1 Offset Loop	Three Line Fragment The first sets $Var1$ to 1 The second is a loop operator The third requires an addition operator such that $Var2 = loop\_iterator + Var1$
Length -1 Loop	Three Line Fragment The first sets $Var1$ to 1 The second requires an addition operator such that $Var2 = input\_array\_size + Var1$ The third is a loop operator bounded to $Var2$
Literal (2)	Requires a variable to be set to 2
Loop	Requires the program to contain a loop
Loop Conditional	Two line fragment. The first line requires a loop The second line requires a conditional ($var > 0$)
Loop Read	Two line fragment. The first line requires a loop The second line requires a read operation such that the index read is the loop's iterator
Nonstandard Array	One line fragment. Requires the output array to be created, with a size $Var1$ such that $Var1$ is not the variable defaulting to input array size
Read	Requires the program to read from the input array
Add	Simple subtraction. Requires the program to at some point contain a subtract operation

Table A.25: Fragments used in experiment 2, corpus 1, to determine whether the NN can recognise their presence in a program's source code based on its behaviour. If multiple lines are required they are required to exist in order, but not necessarily consecutively. Variable numbering is not reflective of source-code implementation and for descriptive purposes only.

Fragment	Description
Add	Simple addition. Requires the program to at some point contain an addition operation
Conditional	Greater than 0 operator. The program must contain an operator which executes a non-empty code block if a variable is greater than zero
Half	Two line fragment. The first sets a variable $Var1$ to the literal 2, The second assigns a variable to $desired\_output\_size/Var1$
Half Loop	Three line fragment. The first sets a variable $Var1$ to the literal 2, The second assigns a variable $Var2$ to $desired\_output\_size/Var1$ The third defines a loop operator which runs from 0 to $Var2$
Half Loop Depends	Four line fragment. The first sets a variable $Var1$ to the literal 2, The second assigns a variable $Var2$ to $desired\_output\_size/Var1$ The third defines a loop operator which runs from 0 to $Var2$ The fourth defines an operation setting a value on the 2D canvas, with the requirement that the X position of the point be logically dependent on the loop iterator
Loop Conditional	Two line fragment. The first requires a loop operator The second requires a conditional ($var > 0$) operator
Loop Draw	Two line fragment. The first requires a loop operator The second requires a 2D array write operation in which the X position drawn to depends logically on the loop's iterator
Loop Loop	Two line fragment. The program must have two loops (not necessarily nested)
Loop Loop Subtract	Three line fragment. The program must have two loops (not necessarily nested) It must then have a subtract operator
Draw Draw	Two line fragment. The program must have two draw-to-2D-array operators

Table A.26: Fragments used in experiment 2, corpus 2, to determine whether the NN can recognise their presence in a program's source code based on its behaviour. If multiple lines are required they are required to exist in order, but not necessarily consecutively. Variable numbering is not reflective of source-code implementation and for descriptive purposes only.

A.7 Transfer learning metrics in detail

In Section 6.2.3 we reported averaged metrics for transfer learning; here we report the per-problem metrics for time-to-threshold and transfer ratio, in Tables A.27 and A.28.

Problem	**Generations to threshold**	**Transfer Ratio**
Add	2388.8	0.89
Append	0.0	0.95
CumulativeAbsoluteSum	2346.4	1.00
CumulativeSum	2205.2	0.89
KeepEvenIndices	1888.8	0.66
ClipToMin	1932.4	0.70
RetainSecondHalf	0.0	0.93
Sort	2277.6	0.98
Subtract	1996.8	0.71
Abs	1855.2	0.76
GreaterThan	2362.4	1.39
IndexParity	774.8	0.46
FirstElementOnly	0.0	0.79
Identity	604.4	0.42
DivergentSequence	1846.8	0.72
Double	1322.8	0.55
ShiftRight	0.0	0.92
ShiftRightLossy	1404.0	0.55
ShiftLeft	948.8	0.52
ShiftLeftZeroPadded	1825.2	0.76
RetainFirstHalf	0.0	0.92
LessThan	2718.8	1.35
Multiply	1818.8	0.69
Negative	1386.8	0.49
Pop	1257.6	0.49
KeepPositives	1499.2	0.63
KeepEvens	2314.4	0.92
ArrayLength	0.0	0.24
ArrayToZero	0.0	0.22
KeepNegatives	1856.4	0.74
KeepOdds	1783.6	0.94
Reverse	1504.8	0.58
CatToSelf	2111.2	0.81
CatZerosToSelf	2242.4	0.80
ClipToMax	1917.2	0.86

Table A.27: Transfer Learning metrics for 25 runs of the TL system compared to the baseline for the first corpus. Generations to Threshold represents the average number of generations take for the TL system to outperform the baseline's asymptotic performance (TL will gain performance after this point). Transfer Ratio is the average fitness of the TL system divided by the average fitness of the baseline (fitnesses less than -1 set to -1).(n=25)

Problem	**Generations to threshold**	**Transfer Ratio**
Square	211.0	0.12
HollowSquare	1060.9	0.27
Parallelogram	1158.3	0.57
HollowParallelogram	2129.4	0.87
MirroredParallelogram	1830.6	0.75
MirroredHollowParallelogram	1968.1	0.72
RightTriangle	152.0	0.07
HollowRightTriangle	1003.0	0.18
MirroredRightTriangle	787.5	0.40
HollowMirroredRightTriangle	1605.4	0.59
InvertedRightTriangle	271.0	0.14
HollowInvertedRightTriangle	2458.2	0.80
InvertedMirroredRightTriangle	149.0	0.06
Inv'HollowMirr'RightTriangle	792.7	0.24
IsoceleseTriangle	0.0	0.89
HollowIsoceleseTriangle	2662.2	0.91
InvertedIsoceleseTriangle	1338.5	0.52
HollowInv'IsoceleseTriangle	1851.3	0.64
RectangleWithEmptyTrapezoid	2693.9	1.20
Inv'dRect'WithEmptyTrapezoid	1952.4	0.95
ObtuseTriangle	1832.9	0.74
HollowObtuseTriangle	2556.0	0.96
MirroredObtuseTriangle	465.3	0.48
MirroredHollowObtuseTriangle	2140.6	0.78
InvertedObtuseTriangle	0.0	0.95
HollowInvertedObtuseTriangle	2513.9	0.93
InvertedMir'ObtuseTriangle	1981.7	0.95
HollowMir'Inv'ObtuseTriangle	2273.9	0.95
VShape	2022.1	0.63
Trapezoid	561.8	0.45

Table A.28: Transfer Learning metrics for 25 runs of the TL system compared to the baseline for the second corpus. Generations to Threshold represents the average number of generations take for the TL system to outperform the baseline's asymptotic performance (TL will gain performance after this point). Transfer Ratio is the average fitness of the TL system divided by the average fitness of the baseline (fitnesses less than -1 set to -1).(n=25)

Bibliography

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: Results and Analysis. *Electronic Proceedings in Theoretical Computer Science* 229 (11 2016), 178–202.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. *2008 IEEE Congress on Evolutionary Computation, CEC 2008* (2008), 162–168. `https://doi.org/10.1109/CEC.2008.4630793`

Douglas Adriano Augusto and Helio JC Barbosa. 2000. Symbolic regression via genetic programming. In *Proceedings. Vol. 1. Sixth Brazilian Symposium on Neural Networks*. IEEE, 173–178.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17* (proceedings of iclr'17 ed.).

Shumeet Baluja and Rich Caruana. 1995. *Removing the Genetics from the Standard Genetic Algorithm.* Technical Report. USA.

Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014a. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 306–317.

Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014b. The plastic surgery hypothesis. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014* (2014), 306–317. `https://doi.org/10.1145/2635868.2635898`

Peter L. Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. 2019. Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks. *Journal of Machine Learning Research* 20, 63 (2019), 1–17.

Yonatan Belinkov and James Glass. 2019. Analysis Methods in Neural Language Processing: A Survey. *Transactions of the Association for Computational Linguistics* 7 (March 2019), 49–72.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680 [cs.LG]

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* 60, 2 (2018), 223–311.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H.S. Torr, and Pushmeet Kohli. 2017. Learning to superoptimize programs. In *ICLR*.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276* (2018).

Biman Chakraborty and Probal Chaudhuri. 2003. On The Use of Genetic Algorithm with Elitism in Robust and Nonparametric Multivariate Analysis. 32 (01 2003).

Qi Chen, Bing Xue, and Mengjie Zhang. 2020. Genetic Programming for Instance Transfer Learning in Symbolic Regression. *IEEE Transactions on Cybernetics* PP (02 2020), 1–14.

X. Chen, C. Liu, and D. Song. 2019. Execution-Guided Neural Program Synthesis. In *ICLR*.

X *et al* Chen. 2017. Towards Synthesizing Complex Programs from Input-Output Examples. *ICLR* (2017), 1–31. arXiv:1706.01284

Brendan Cody-kenny, Edgar Galván-lópez, and Stephen Barrett. 2015. locoGP : Improving Performance by Genetic Programming Java Source Code. *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015), 811–818. `https://doi.org/10.1145/2739482.2768419`

Cristina David and Daniel Kroening. 2017. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society of London Series A* 375, 2104, Article 20150403 (Sept. 2017), 20150403 pages.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017a. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 990–998.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017b. RobustFill: Neural Program Learning Under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 990–998.

Dinabandhubhandari, Chaitra Murthy, and Sankar Pal. 2012. Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence* 10 (04 2012).

Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. 2019. Novelty Search: A Theoretical Perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 99–106.

K. Ellis, Maxwell Nye, Y. Pu, Felix Sosa, J. Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *NeurIPS*.

Tohid Erfani and S. Utyuzhnikov. 2011. Directed search domain: a method for even generation of the Pareto frontier in multiobjective optimization. *Engineering Optimization* 43 (2011), 467 – 484.

Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. 2000. Genetic Programming and Simulated Annealing: A Hybrid Method to Evolve Decision Trees, Vol. 1802. 294–303.

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Qu&#233;bec, Canada) *(GECCO '09)*. ACM, New York, NY, USA, 947–954. `https://doi.org/10.1145/1569901.1570031`

Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. 262–277.

S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill. 2018. Towards Understanding and Refining the General Program Synthesis Benchmark Suite with Genetic Programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*. 1–6.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (01 Oct 2016), 471–476.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*.

Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL meets ML. In *APLAS 2017* (aplas 2017 ed.). Springer.

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages.

Saemundur O Haraldsson and John R Woodward. 2014. Automated Design of Algorithms and Genetic Improvement : Contrast and Commonalities. *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion* (2014), 1373–1380. `https://doi.org/10.1145/2598394.2609874`

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2018. Program Synthesis Using Uniform Mutation by Addition and Deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Kyoto, Japan) *(GECCO '18)*. Association for Computing Machinery, New York, NY, USA, 1127–1134.

Thomas Helmuth, Edward Pantridge, Grace Woolson, and Lee Spector. 2020. Genetic Source Sensitivity and Transfer Learning in Genetic Programming *(ALIFE 2021: The 2021 Conference on Artificial Life, Vol. ALIFE 2020: The 2020 Conference on Artificial Life)*. 303–311.

Thomas Helmuth and Lee Spector. 2015. General program synthesis benchmark suite. *GECCO 2015 - Proceedings of the 2015 Genetic and Evolutionary Computation Conference* (2015), 1039–1046.

Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On Domain Knowledge and Novelty to Improve Program Synthesis Performance with Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1039–1046.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. (2015), 1–9. arXiv:1503.02531

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2018. Densely Connected Convolutional Networks. arXiv:1608.06993 [cs.CV]

Hitoshi Iba, Ji Feng, and Hossein Izadi Rad. 2018. GP-RVM: Genetic Programing-Based Symbolic Regression Using Relevance Vector Machine. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 255–262.

Wojciech Jaskowski, Krzysztof Krawiec, and Bartosz Wieloch. 2007. Knowledge Reuse in Genetic Programming Applied to Visual Learning. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (London, England) *(GECCO '07)*. Association for Computing Machinery, New York, NY, USA, 1790–1797.

Kenji Kawaguchi and Yoshua Bengio. 2018. Depth with Nonlinearity Creates No Bad Local Minima in ResNets. (2018), 1–14. arXiv:1810.09038

Maarten Keijzer, Conor Ryan, and Mike Cattolico. 2004. Run Transferable Libraries — Learning Functional Bias in Problem Domains. In *Genetic and Evolutionary Computation – GECCO 2004*, Kalyanmoy Deb (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–542.

Stephen Kelly and Malcolm Heywood. 2015. Knowledge Transfer from Keepaway Soccer to Half-field Offense through Program Symbiosis. 1143–1150.

Stephen Kelly and Malcolm I. Heywood. 2018. Discovering Agent Behaviors Through Code Reuse: Examples From Half-Field Offense and Ms. Pac-Man. *IEEE Transactions on Games* 10, 2 (2018), 195–208.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-Normalizing Neural Networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 972–981.

Abdullah Konak, David W. Coit, and Alice E. Smith. 2006. Multi-objective optimization using genetic algorithms: A Tutorial. *Reliability Engineering and System Safety* 91, 9 (2006), 992–1007. Special Issue - Genetic Algorithms and Reliability.

Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. Removing the Kitchen Sink from Software. *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15* (2015), 833–838. `https://doi.org/10.1145/2739482.2768424`

W. B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb 2015), 118–135. `https://doi.org/10.1109/TEVC.2013.2281544`

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551.

Joel Lehman and Kenneth Stanley. 2010. Efficiently evolving programs through the search for novelty. *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, 837–844.

Paweł Liskowski, Krzysztof Krawiec, Nihat Engin Toklu, and Jerry Swan. 2020. Program Synthesis as Latent Continuous Optimization: Evolutionary Search in Neural Embeddings. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancún, Mexico) *(GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 359–367.

Shantanu Mandal, Todd A. Anderson, Javier S. Turek, Justin Gottschlich, Shengtian Zhou, and Abdullah Muzahid. 2019. Learning Fitness Functions for Machine Programming. `https://doi.org/10.48550/ARXIV.1908.08783`

B. Miller and D. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Syst.* 9 (1995).

Vojtech Mrazek, Zdenek Vasicek, and Lukas Sekanina. 2015. Evolutionary Approximation of Software for Embedded Systems: Median Function. *Genetic Improvement 2015 Workshop* 1 (2015), 795–801. `https://doi.org/10.1145/2739482.2768416`

Brandon Muller, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. 2019. Transfer learning: a building block selection mechanism in genetic programming for symbolic regression. 350–351.

Luis Muñoz, Leonardo Trujillo, and Sara Silva. 2020. Transfer learning in constructive induction with Genetic Programming. *Genetic Programming and Evolvable Machines* 21 (12 2020).

Andrew Y Ng. 2004. Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*. 78.

AkshatKumar Nigam, Pascal Friederich, Mario Krenn, and Alan Aspuru-Guzik. 2020. Augmenting Genetic Algorithms with Deep Neural Networks for Exploring the Chemical Space. In *International Conference on Learning Representations*.

Damien O'Neill, Harith Al-Sahaf, Bing Xue, and Mengjie Zhang. 2017. Common subtrees in related problems: A novel transfer learning approach for genetic programming. 1287–1294.

Michael Orlov. 2017. Evolving Software Building Blocks with {FINCH}. *Gi-2017* (2017). `https://doi.org/doi:10.1145/3067695.3082521`

Edward Pantridge and Lee Spector. 2020. Code building genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. 994–1002.

Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *ICLR*.

Justyna Petke, Saemundur Haraldsson, Mark Harman, william Langdon, David White, and John Woodward. 2017. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* c (2017), 1–1. `https://doi.org/10.1109/TEVC.2017.2693219`

Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432.

Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. *CoRR* abs/1802.04335 (2018). arXiv:1802.04335

Scott Reed and Nando de Freitas. 2016. NEURAL PROGRAMMER-INTERPRETERS. In *ICLR*.

Joseph Renzullo, Westley Weimer, Melanie Moses, and Stephanie Forrest. 2018. Neutrality and Epistasis in Program Space. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop* (Gothenburg, Sweden) *(GI '18)*. Association for Computing Machinery, New York, NY, USA, 1–8.

Luis Rios and Nikolaos Sahinidis. 2009. Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization* 56 (11 2009).

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (Oct. 1986), 533–536.

Conor Ryan, J. J. Collins, and Michael O'Neill. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings of the First European Workshop on Genetic Programming (LNCS, Vol. 1391)*, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.). Springer-Verlag, Paris, 83–96.

Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. 2013. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2451116.2451151

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (Jan. 2019), 29 pages.

Eui Chul Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc.

Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019a. Program Synthesis and Semantic Parsing with Learned Code Idioms. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10824–10834.

Richard Shin, Neel Kant, Kavi Gupta, Christopher Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. 2019b. synthetic datasets for neural program synthesis. In *ICLR*.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. *Pldi* (2015), 43–54. https://doi.org/10.1145/2737924.2737988

H.T. Siegelmann and E.D. Sontag. 1995. On the Computational Power of Neural Nets. *J. Comput. System Sci.* 50, 1 (1995), 132–150.

Rishabh Singh. 2016. BlinkFill: Semi-Supervised Programming by Example for Syntactic String Transformations. *Proc. VLDB Endow.* 9, 10 (June 2016), 816–827.

Robert J. Smith and Malcolm I. Heywood. 2019. Evolving Dota 2 Shadow Fiend Bots Using Genetic Programming with External Memory. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 179–187.

Sunbeom So and Hakjoo Oh. 2018. Synthesizing Pattern Programs from Examples. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 1618–1624.

Léo Françoso Dal Piccol Sotto and Franz Rothlauf. 2019. On the Role of Non-Effective Code in Linear Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Prague, Czech Republic) *(GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1075–1083.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010a. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 313–326.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010b. From Program Verification to Program Synthesis. *SIGPLAN Not.* 45, 1 (Jan. 2010), 313–326.

Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. 2018a. Neural Program Synthesis from Diverse Demonstration Videos. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 4790–4799.

Shao Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph J. Lim. 2018b. Supplementary material of neural program synthesis from diverse demonstration videos. *35th International Conference on Machine Learning, ICML 2018* 11 (2018), 7624–7626.

Matthew E. Taylor and Peter Stone. 2011. An Introduction to Inter-task Transfer for Reinforcement Learning. *AI Magazine* 32, 1 (2011), 15–34.

Alan M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265.

A. M. Turing. 1950. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind* LIX, 236 (10 1950), 433–460.

Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I McKay, and Edgar Galván-López. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (2011), 91–119.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. HOUDINI: Lifelong Learning as Program Synthesis. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 8701–8712.

V. N. Vapnik and A. Ya. Chervonenkis. 2015. *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities.* Springer International Publishing, Cham, 11–30.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 5045–5054.

Jordan Wick, Erik Hemberg, and Una-May O'Reilly. 2021. Getting a Head Start on Program Synthesis with Genetic Programming. *EuroGP* (2021).

Alexander Wild and Barry Porter. 2019. General Program Synthesis using Guided Corpus Generation and Automatic Refactoring. In *Search-Based Software Engineering (Lecture Notes in Computer Science)*, Shiva Nejati and Gregory Gay (Eds.). Springer-Verlag, 89–104.

Alexander Wild and Barry Porter. 2021. Neurally Guided Transfer Learning for Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 267–268. https://doi.org/10.1145/3449726.3459511

Alexander Wild and Barry Porter. 2022. Multi-Donor Neural Transfer Learning for Genetic Programming. *ACM Trans. Evol. Learn. Optim.* 2, 4, Article 12 (nov 2022), 40 pages. https://doi.org/10.1145/3563043

Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.* 8, 3–4 (may 1992), 229–256.

Songtao Wu, Shenghua Zhong, and Yan Liu. 2017. Deep residual learning for image steganalysis. *Multimedia Tools and Applications* (2017), 1–17. arXiv:1512.03385

Jinghui Zhong, Liang Feng, Wentong Cai, and Yew-Soon Ong. 2018. Multifactorial genetic programming for symbolic regression problems. *IEEE transactions on systems, man, and cybernetics: systems* 50, 11 (2018), 4492–4505.

Amit Zohar and Lior Wolf. 2018a. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2094–2103.

Amit Zohar and Lior Wolf. 2018b. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems* (Montr&#233;al, Canada) *(NIPS'18)*. Curran Associates Inc., USA, 2098–2107.