Neurally Guided Transfer Learning for Genetic Programming

Alexander Wild
a.wild3@lancaster.ac.uk
School of Computing and Communications,
Lancaster University
Lancaster, UK

Barry Porter
b.f.porter@lancaster.ac.uk
School of Computing and Communications,
Lancaster University
Lancaster, UK

ABSTRACT

A key challenge in GP is how to learn from the past, so that the successful synthesis of simple programs can feed in to more challenging unsolved programs. In this work we present a transfer learning (TL) mechanism for GP which extracts fragments from programs it synthesises, then employs deep neural networks to identify new problems to deploy them into, to boost performance. This end-to-end system requires no human identification of which programs to use as donors for TL, the system only needs IO examples.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Genetic programming**.

KEYWORDS

Genetic Programming, Transfer Learning, Neural Networks

ACM Reference Format:
Alexander Wild and Barry Porter. 2021. Neurally Guided Transfer Learning for Genetic Programming. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3449726.3459511

1 INTRODUCTION

This works aims to improve performance of Genetic Programming by leveraging the power of modern Deep Learning techniques to recognise high-level code properties, using it to power an end-to-end fully automated Transfer-Learning (TL) system. The work demonstrates an approach to a step missing in the literature of TL for GP, automated identification of donor program. Our system takes only IO examples of desired behaviour, and exploits its previous successes to solve future problems. We demonstrate a gain in both probability of solving a problem and number of problems for which at least one run produced a solution.

Existing work in Transfer Learning for GP. Transfer Learning (TL) has been studied to some extent in the field of Genetic Programming, including the effects of transferring sub-trees between symbolic regression problems and how to select trees for best results [1–3]. Our work's primary contribution is the use of machine learning, specifically deep neural networks to identify donor programs from already-solved problems. We present a fully automated approach to selecting which prior solved problem, and associated code fragments, to use as a basis for new unsolved problems – rather than relying on human-aided selection. As such, we require no designer knowledge of the problem space in selecting useful genetic material for transfer in to new problems.

2 METHODOLOGY

Algorithm 1 An overview of the process we envisage as a fully automated end-to-end approach to Transfer Learning between problems within a corpus

```
while unsolved_problems_exist do
    for all stored_fragments do
        present problem's IO to fragment's associated NN
        if NN estimate of probability > 0.5 then
            $S = arbitary\_ranking\_heuristic$
        end if
    end for
    run GP on problem with fragment with highest $S$
    if GP returns valid solution then
        extract all code fragments from solution
        for all extracted_fragments do
            generate 10,000 program training set $P$ wherein all programs contain given fragment
            generate 10,000 program training set $N$ wherein all programs do not contain given fragment
            train an NN to discriminate between $P$ and $N$
            evaluated trained NN on existing found solutions
            if NN accuracy on found solutions > 0.5 then
                store NN + fragment for future deployment
            end if
        end for
    end if
end while
```

Our approach employs a language designed for linear genetic programming, employing the open-source Turing-Complete language used in [4]. It initially aims to solve problems using standard GP techniques, from a user-supplied list of IO example set problem specifications. If it solves any, it breaks down the newly synthesised program into 'fragments' of code. These are any sequence of four or fewer lines of code which do not depend on any previous lines, and may contain flow control operators such as loops or conditionals. We define a line as depending on a previous one if the line reads from a variable written by a previous one, and we consider a line previous to another if its operation could have been executed before that line, either due to occurring earlier in a program or due to a loop both lines are contained in.

For every fragment extracted, two corpora of 10,000 programs are generated: one in which every program has the given fragment, one in which none do. These programs are generated by first taking all GP-synthesised programs which fit the given corpus' requirements (either having or lacking the fragment), and adding them to the corpus. At this point, both corpora will feature only the seed programs, and have a size of at least one but far beneath the desired 10,000. They are then expanded by iterative child generation and addition. While the corpus is below target size, two parent programs from it are chosen uniformly at random, a crossover mutation applied, then they are mutated using the same process employed by our GP. Programs are then only accepted if they are demonstrated to have different behaviours on a fixed testing input set.

Once these two corpora of programs have been produced, a neural network (NN) is trained to recognise Input-Output (IO) mappings generated by feeding randomly generated inputs into programs from both corpora. This aims to produce a network able to recognise which programs have the given fragment and which do not, based purely on an IO example set. Each fragment therefore has its own associated NN, which is trained to recognise it.

When the system moves onto the next unsolved user-provided IO example set, it can run every trained NN and receive a probability estimate for presence of the NN's associated fragment. A heuristic is employed to decide which fragment to select, from the set of all those with a NN returning an estimated presence probability > 0.5. We employ a rarity heuristic, which selects fragments which have a lower sum probability estimate across all user provided IO, that is to say the ones which would we deployed on fewer problems. This selects for 'specialist' fragments, rather than broadly applicable ones such as simply 'possesses a loop'.

The NN architecture employed is a simple feed-forward network, which takes a fixed-length encoding of the IO examples (with padding to support the randomly chosen array lengths), and passes it through 4 densely connected layers of 128 neurons, before outputting via single sigmoid-range neuron (serving as a probability estimate for the presence of the code fragment).

With a fragment selected, we run a normal GP process, but statically force all programs in the populations to include the given fragment in their source code (initial generation is additive mutations of the fragment itself). We allow the fragment to exist solely as non-functional intron code, which allows it to be excluded from the functionality of the GP process if it is not beneficial, or to remain ready for re-activation by subsequent mutations as the program develops over the GPs' generations.

An overview of the approach is given in Algorithm 1, showing the main loop of the process.

3 RESULTS AND DISCUSSION

Complete list of problems and results from this section are available in attached appendix.

We evaluate our system on a set of 35 problems, all of which take the form of a function which receives both an integer and an array of integers, and returns an array of integers. These programs range for easy tasks, such as copying the input array to the output, to sorting; concatenation; returning the absolute of the input; and conditionals relating to the integer and to the values in the input array. We allow the system two passes through the problem corpus, to allow it to potentially transfer from programs lower down in the corpus to problems earlier (our baseline comparison is similarly allowed two runs to maintain equivalent CPU expenditure).

Approach	Find Rate	Problems Solved
Baseline (using NS)	42.6%	27
GP Transfer Learning	54.1%	32

Table 1: Success of the Transfer Learning system on a corpus of 35 integer and integer-array problems compared to the GP process without TL. (n=20)

Our results are displayed in table 1. We see that not only does the probability of finding an arbitrary program increase, some problems which had a find rate too low to measure with 20 repeats began to be found. While 8 problems remained unsolved, including sorting, this indicates that the technique is useful for boosting the complexity of problems which can be solved, rather than simply boosting reliability on problems the GP can already handle.

We saw largest gains in the problems which required an output array of differing length to the input one. This is likely due to our fitness function being very course with regards to output length, with a flat penalty for any length difference, rather than a shaped landscape. As such, the GP itself struggled to assemble multiple lines which would correctly generate the desired output array length. Our preliminary experiments indicate that an NN is able to spot the need for array length change code with an accuracy of 73%, and can therefore deploy the code in a fashion which preserves it until subsequent mutations can successfully build upon it.

Conversely, no problem was majorly degraded. The code fragments are not obligated to form a functional part of the eventual program, their operators could be rendered inert (i.e. by writing to variables which are never read).

4 ACKNOWLEDGEMENTS

This work was partly supported by the UK Leverhulme Trust via the *Self-Aware Datacentre* project, grant RPG-2017-166.

REFERENCES

[1] 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. https://doi.org/10.1109/TCYB.2020.2969689

[2] 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. https://doi.org/10.1145/3319619.3322072

[3] 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. https://doi.org/10.1109/CEC.2017.7969453

[4] 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. https://doi.org/10.1007/978-3-030-27455-9_7