Evolving Artificial Neural Networks with FINCH

Amit Benbassat
Dept. of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
amitbenb@cs.bgu.ac.il

Moshe Sipper
Dept. of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
sipper@cs.bgu.ac.il

ABSTRACT

We present work with the FINCH automatic evolutionary programming tool to evolve code that generates Artificial Neural Networks (ANNs) that perform desired tasks. We show how FINCH can be used to evolve code that generates an ANN that performs a simple classifying task with proficiency.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.5.1 [**Pattern Recognition**]: Models—*Neural Networks*

General Terms

Algorithms, Languages

Keywords

NeuroEvolution; FINCH

1. INTRODUCTION

The FINCH system developed by Orlov and Sipper [2] is an automated programming system that evolves unrestricted Java bytecode programs. The system receives a seed method written in Java, translates it to Java bytecode, and uses this seed to kick-start an evolutionary process that generates programs that successfully solve a given problem. Orlov and Sipper [2] used FINCH to evolve solutions for some classical GP benchmarks such as symbolic regression, trail navigation, and the intertwined spiral problem explored by Koza [1]. FINCH achieved consistent repeatable success in evolving code that solved these benchmark problems.

In this work we try to look at FINCH from a different angle. Unrestricted, automatically generated code can potentially prove clumsy, buggy or otherwise problematic in the long run. If one wants to use the product of automatic programming in the long run one would want assurance of its stability. One way to gain such assurance is to rely on handwritten, verified code. Herein we use an indirect encoding scheme in conjunction with FINCH to evolve code that interacts with handwritten code that implements an *Artificial Neural Network* (ANN) interface. The evolved function, rather then being evaluated directly, builds an ANN that is in then evaluated according to its proficiency in a given task.

Copyright is held by the author/owner(s).
GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.

2. ANN INTERFACE

We have tried various ANN implementations, finally settling on a simple implementation of an acyclic ANN in which links can be added one by one using an interface. The ANN object is given its number of inputs, outputs, and additional neurons at the time of generation. It accepts input vectors that contain values from $\{1, -1\}$.

We defined a simple interface that allows FINCH to alter the structure of the ANN by adding links or changing existing ones. The interface also allows for a node-adding option which we have not yet implemented. It is easy to add more methods to the interface, and during our work we added two more, called *lock* and *unlock*, in an attempt to use them to control some undesired FINCH behaviors.

Our current ANN FINCH interface includes the follwing methods:

- addLink(int,int,double) —Add a new link to the network. The method recives two integer indices for the two sides of the link, and a floating point value for the link's weight.
- addNode(int) —Add a new artificial neuron node to the network. Currently this method is unimplemented in all of our ANN classes.
- lock() —Add a lock to the network. The idea here is algorithm control. In our implemented classes locked networks cannot run, and this allows us to put limitations on the number of calls to methods from the evolved method.
- unlock() —Remove a lock from the network. This is complementary to the previous method.

3. THE TASKS

We focus on simple tasks, which we call "signal intensity tasks", where the expected output of the ANN depends only on the number of 1s in the input vector. These include:

1. The intensity/size classifier —This type of network should return 1 iff at least half of the inputs are 1s.
2. Counting network —This type of network should return an output with the same number of 1s as the input vector.

In order to evolve code in FINCH a seed Java method is required. As we did not want to inject our knowledge of the problems into FINCH we decided to use randomly generated code as the seed individual. Initially the code comprised of *addLink()* method calls that used randomly generated parameters. We abandoned this approach because it caused technical difficulties in the form of

Draft

crossover failures and exceptions during runs. Instead we opted for evolving a method that manipulated 40 Java variables (the seed was again randomly generated) and then called another method which added links to the ANN in a way depending on these same variables. For example, see Equation 1 that contains an example of a line of code from the seed method, and Equation 2 that contains a line of code from the ANN building method that uses some of the same variables.

$$v_i6 = 689 + v_i7; \tag{1}$$

$$ann.addLink(v_i6 + 599, v_i5 + 720, v_d7 + -2.706); \tag{2}$$

In Figure 1 we see a small neural network. Our system allows for links to have any hidden or output neuron as their destination, but makes sure the source comes before the destination in a predecided node ordering.

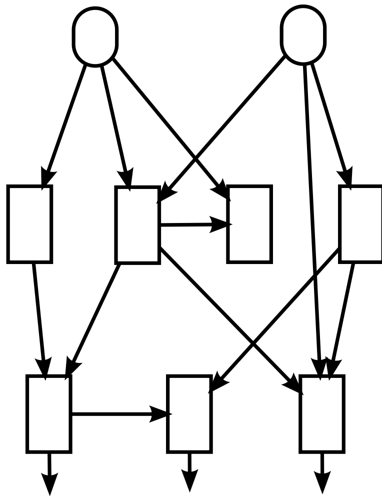

Figure 1: A small example of an artificial neural network that has 2 inputs, 4 hidden neurons and 3 output neurons. Flow in the network is from top to bottom. Links in the network, represented here by arrows, can either go down to a lower level, or go right in the same level.

4. RESULTS

4.1 Intensity/Size Classifier

It is not difficult to handcraft a relatively simple ANN for this task. It is however more difficult for FINCH to find an effective ANN without any prior knowledge of the suitable ANN structure. In our intensity classifier experiments we used the following run parameters:

- ANNs with 16 inputs, 3 outputs, and 6 hidden nodes (25 nodes total).
- Population size 300.
- Generation limit of 20.
- Tournament selection with a tournament size of 4.
- Uniform crossover with xo probability of 0.9.
- fitness is evaluated over 2000 random inputs.

This setup routinely resulted in networks that responded correctly for over 80% of inputs.

4.2 Counting Network

In this task the network should match the number of 1s in the output to the number of 1s in the input. This task is solved by a trivial network if the size of the input and output arrays is the same, but it is trickier when there are less outputs than there are inputs (in this case we make sure only to test on inputs with less 1s than the total number of outputs). Using a small number of outputs allows us to turn counting into a more managable task. In our counting network experiments we used the following run parameters:

- ANNs with 16 inputs, 3 outputs, and 6 hidden nodes (25 nodes total).
- Population size 600.
- Generation limit of 100.
- Tournament selection with a tournament size of 4.
- Uniform crossover with xo probability of 0.9.
- fitness is evaluated over 2000 random inputs.

This setup managed to occasionally hit upon the right solution in this more difficult task (compared with intensity classification) and get a score of 100%. In other runs FINCH achieved substantial progress over the initial seed individual. We see this as a sign that our system shows promise though it probably still requires some work.

5. CONCLUSIONS

We have shown that FINCH shows promise in the previously unexplored avenue of evolving code that builds useful computational entities. This indirect encoding approach can lead to interesting results in the future.

More work is still required to make FINCH a competative flexible tool for neuroevolution but the potential is there. With more work FINCH can be used as the basis for a developmental evolutionary algorithm to evolve programs that build computational entities such as ANNs.

6. ACKNOWLEDGMENTS

Amit Benbassat is partially supported by the Lynn and William Frankel Center for Computer Sciences. This research was supported by the Israel Science Foundation (grant no. 123/11).

7. REFERENCES

[1] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[2] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.