# Programs that Program

ksb@cs.utexas.edu, jasonzliang@utexas.edu

**Abstract**

Evolutionary computation allows computers to automatically solve problems that can be cast as optimization problems. Until now, instantiations have been hand designed. We propose to allow computers to automatically design such instantiations by using evolutionary computation itself. To do so, we desire programs that write other programs and thereby explore a search space. In this paper, we demonstrate that neural networks (which can behave as programs or program components) can generate other meaningful neural networks.

## Introduction

The following is a quick overview of the rest of the paper: In section 1, we will go over methods for constructing self-replicating feed-forward neural networks. Next, in section 2, we will discuss experimental results. Finally in the last section, we present conclusions and future work.

## 1 Self-Replicating Feed-forward Neural Networks

To create a self-replicating feed-forward neural network, we first use the pyBrain neural network library [3] to create an initial neural network that has a fixed topology. By fixed topology, we mean that the number of input neurons, output neurons, hidden layers, neurons, and number of connections remain constant. The only parameters we plan to optimize in order to generate self-replicating neural networks are the weights of the connections between neurons. Since a feed-forward neural network can have many variations in its topology, we have chosen a simple structure that makes the most sense.

One major challenge is deciding what kind of input to feed into the neural network and how to interpret the output of the neural network as another neural net. For simplicity, we decided on giving our neural network a N-bit vector of binary numbers as input. If we topologically linearize all the connections in the neural network, we can associate the value of binary vector with a corresponding connection. For example, the binary vector 1001 can be associated with the 5th connection in a small network with 8 connections in total. Accordingly, the output of the neural network is a single value and it represents the weight of the connection specified by the input vector. Thus, by feeding the neural network a set of inputs that map to all the connections in the network, we can generate

weights for each of connections. This means that the network is capable of generating another net exactly like itself in topology, but with possibly different connection weights.

The next step is come up with a method to optimize the connection weights of the original network such that child network generated also has the same connection weight. Given the lack of gradients in this problem, we decide to rely on a black-box optimization approach to optimize the connection weights of the parent network. Thus we will require a fitness function that evaluates how close the the neural network is to generating a child network exactly like itself. To compute a fitness value, as seen in Fig. 1, we use the parent network to compute the connection weights of the child network, determine the error between the new connection weight and its original value in the parent network, and return the sum of all the errors. Now we can try a variety of different black-box optimization methods, including genetic algorithms [1], CMAES [2], and NES [4] to try to minimize the error and maximize the fitness of the neural network. Experimentally, we found that using CMAES resulted the most improvement in fitness every generation.

```
def fitnessFunction(parentConnections):
  error = 0.0
  for index, parentWeight in enumerate(parentConnections):
    input = getBinaryVector(index)
    childWeight =  neuralNetwork.activate(input)
    error += abs(parentWeight - childWeight)
 return -1*error
```

Figure 1: Function for computing fitness of a neural network. For each connection in the parent network, we convert it into a binary vector, feed it as input, and get the connection weight of the same connection in the child network. We compare the difference between the child and parent weights and sum up all differences as error, which is then inverted to become a fitness value.

## 2 Experimental Results

## References


[1] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[2] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003.

[3] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 2010.

[4] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.