

Mutation Operator Evolution for EA-Based Neural Networks

Ryan J. Meuth

December 11, 2005

Abstract

In the field of Machine Learning and Computational Intelligence one of the most difficult environments that an algorithm must perform in is that of a continuous, dynamic environment. In this type of situation, an agent must learn the value of an infinite number of states in order to make an informed decision about what action to take. The agent must also be able to adapt to the changing environment. To accomplish this, neural networks are often used to approximate the value function over the state-space. In this paper, a method for evolving mutation operators for Evolutionary Algorithm based neural networks is presented, and the evolved results are compared with existing operators. It is shown that the method described was not able to evolve a significantly distinct operator, possibly due to low selection pressure and experimental setup parameters.

Contents

1	Introduction	3
2	Background	4
3	Problem Statement	4
4	Experimental Setup	4
4.1	Genetic Programming Implementation	5
4.1.1	Representation	5
4.1.2	Initialization	5
4.1.3	Reproduction	5
4.1.4	Parent Selection	6
4.1.5	Survival	6
4.1.6	Fitness Evaluation	6
4.1.7	Termination	7
4.2	Genetic Program Parameters	7
4.3	Evolutionary Algorithm Test Bed	7
4.3.1	Representation	7
4.3.2	Initialization	8
4.3.3	Reproduction	8
4.3.4	Parent Selection	8
4.3.5	Survival	8
4.3.6	Fitness Evaluation	8
4.3.7	Termination	8
4.4	EA Test Bed Parameters	8
5	Results	9
6	Conclusion	10
7	Future Work	14

1 Introduction

In the field of artificial intelligence, one of the hardest things that we can try to make a computer program do is to interact with the real world. In contrast to the well-defined, discrete, simplified world that programs normally operate in, the real world is large, unknown, and complex. In the real world, programs must learn and adapt to new and changing situations in order to be effective. Reinforcement Learning is a set of methods that attempts to bridge this gap between computers and the real world. In Reinforcement learning a program, or agent, acquires information about the state of the environment and chooses an action based on that state. The environment responds to the action by returning a reward, which may be positive or negative, to the agent. The agent then uses the reward to modify what action it takes the next time it is in a similar state. If the agent can visit each state an infinite number of times, the true values for each state can be determined, and the optimal action policy can be found[SuttonBarto1998](Pg.52-85).

Many techniques exist for implementing reinforcement learning, and most center around the construction of a state-value function that records the value of given state. Given a state, the agent then uses the value function to determine the action from that state that will result in the highest reward. In situations where the environment can be represented discretely, or is small and finite, a simple table can represent the value function for every state in the environment. However, many real-world environments are neither discrete or finite, and alternate methods for finding the value function must be used.

Since the environment is continuous, and possibly infinite, it is possible that in a finite amount of time the value function may never be exactly determined. Thus, we must turn to function approximation techniques for practical reinforcement learning implementations. One of the most popular function approximation methods is that of Artificial Neural Networks[SuttonBarto1998](Pg.193-226). Traditionally, ANN's have been used in conjunction with gradient-descent methods and error-backpropagation. Both of these techniques require knowledge of the function to be approximated in order to be successful. Since the exact characteristics of the value function may not be known beforehand, these error-based techniques are inadequate. Also, for problem environments that change over time, such as stock market prediction problems or the location of chairs in a dining room, the characteristics of the value function are highly unpredictable and may change drastically over time. These classes of situations are very difficult to solve with error-based techniques.

In the place of error-based techniques, Evolutionary Algorithms can be used to evolve a set of weights that enables the network to approximate the true value function of the environment. The knowledge-independent properties of EAs make them well suited for value-function approximation, as the characteristics of value function may be unknown beforehand. Traditionally, the mutation operator of choice for evolving neural network weights (which are floating point values) has been the gaussian operator, whereby a value drawn from a gaussian distribution with mean at zero and a small standard deviation is added to the

allele[SalamaHingston1995]. This is analogous to the creep operator of integer representations. Also used is the Uniform mutation operator which is a replacing operator that draws a new weight-value from a distribution. While effective, these may not be the optimal mutation operators for all types of neural network evolution.

2 Background

Current research into the evolution of mutation operators has been sparse up to this point, though some insight into the processes at work can be gleaned from an examination of Evolutionary Strategies. During the development of Evolutionary Strategies, Schwefel and Rechenberg discovered that evolutionary progress only occurs within a very narrow band of mutation step size, called the evolutionary window [Schwefel1975] [Rechenberg1973]. In evolutionary strategies, a mutation step size is paired with each allele in an individual, and this parameter evolves as the individual evolves. When evolving neural networks, the gaussian mutation operator has no dependance on the target allele, and as such no relation to the problem being solved. It may be possible that a problem-wide evolutionary window exists, and the mutation operator itself may be able to be optimized for a given problem.

In 2002 Andreas Koenig reported on a comparison of Gaussian, Cauchy, Adaptive Levy and combined mutation operators for Evolutionary Strategy and Evolutionary Programming methods. His work showed that Cauchy distribution based methods were superior on most benchmark problems, but these problems were solved using evolutionary strategies, which neglected the effects of recombination[Koenig2002].

3 Problem Statement

This project is concerned with determining if mutation operators can be evolved using genetic programming that are better at evolving neural network weights for a general class of problems and neural network architectures.

4 Experimental Setup

For this problem, a two-tiered evolutionary scheme was implemented, where the first tier uses genetic programming to evolve mutation operators which are evaluated by the second tier, a series of evolutionary algorithms that evolve neural networks for different classes of problems such as non-linear approximation and time-series prediction. Figure 1 shows a diagram of this architecture.

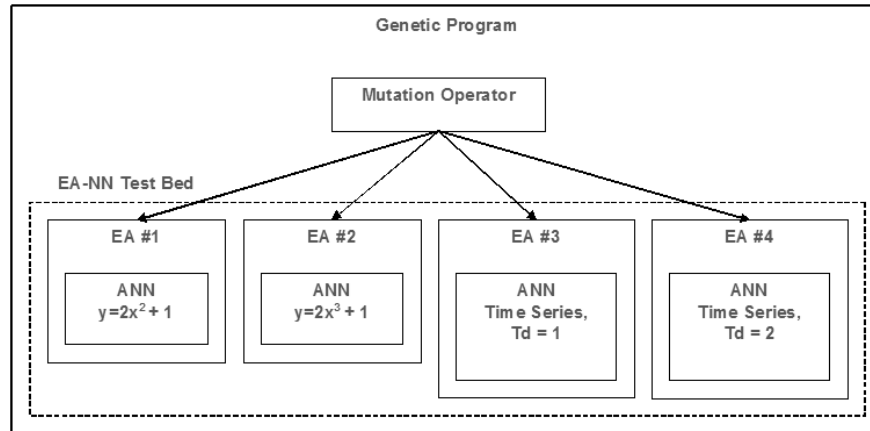


Figure 1: Experimental Setup

4.1 Genetic Programming Implementation

4.1.1 Representation

In Genetic Programming, individuals are represented as parse tree structures that contain functions at non-leaf nodes and constants or variables at leaf nodes [Eiben2003] (Pg.103). Thus, the tree can represent a function or even a program, depending on the function set. For this problem, the function set consists of addition, subtraction, multiplication and division $\{+, -, *, \div\}$. The terminal set includes the value of the weight to be modified, a random variable from a uniform distribution, and random constant values. This combination of terminal types allows both replacing and creep operators to be formed, including the standard mutation operators used with EA-based neural networks.

4.1.2 Initialization

Individuals in the genetic program are initialized as randomly generated trees that are grown from the root using a recursive process. This process is limited by the maximum tree depth.

4.1.3 Reproduction

Reproduction is accomplished through the use of both recombination and mutation operators. In genetic programming, recombination is accomplished through sub-tree swapping. The mutation operation is dependant on what part of the tree is undergoing mutation. If the node under mutation is a terminal node, creep and replacement operations are performed with equal probability on the constant, along with a probability of addition or deletion of operator nodes and

constant and variable values. For function nodes mutation occurs as a replacement operation, along with a probability of addition or deletion of operator nodes. Such a wide-sweeping mutation operator is intended to promote diversity, but incurs computational expense as the individuals can become large.

4.1.4 Parent Selection

Parent Selection is accomplished through over-selection, whereby individuals are ranked by fitness and then divided into two groups, one containing the top $x\%$ and the other containing the remaining $(100 - x)\%$ of the population. When parents are selected, 80% of them come from the first group, and the remaining 20% come from the second group. The value of x is given by an empirical 'rule of thumb' that depends on the population size. For this problem, the population size is 1000 individuals, so x takes a value of 32% [Eiben2003](Pg. 109). This method is used to maintain a high selection pressure for large populations, and to select individuals as parents that are likely to produce high-fitness offspring. This promotes a high-fitness population, and reduces the likelihood of wasting time evaluating individuals that are low-fitness.

4.1.5 Survival

Survival is accomplished through a steady-state strategy in order to combat the destructive effects of crossover. This is accomplished by ranking the population by fitness, and the top n individuals are kept, while the rest are destroyed, where n is the maximum population size.

4.1.6 Fitness Evaluation

Evaluation for the GP Tier is performed by the EA Test Bed. The fitness function for a trial mutation operator consists of the inverse sum of the four average MSE values from the EA Test bed added to the number of nodes in the trial operator, as follows:

$$fitness_i = \frac{1000}{(\sum AvgMSE_i^j - n_i)}$$

Where i is the individual under evaluation, j is the index of the result from EA-NN evaluations, and n is the number of nodes in the individual's parse tree. This fitness function has the effect of promoting short, efficient mutation operators that result in low MSE values during the evaluation stage.

Individuals in Genetic programming have the tendency to grow over the evolutionary process if left unchecked, so the number of nodes in the individual are taken into account during evaluation, with penalties for each node. This has the effect of promoting computationally efficient individuals, and reducing the computational resources necessary to complete evaluation[Engel2002](Pg.149). Also, a repair function is implemented that checks the tree structure for errors, and limits the size of the tree to a maximum depth of 8, or 256 nodes.

4.1.7 Termination

The GP will execute until the maximum number of generations is reached.

4.2 Genetic Program Parameters

The GP is executed with a population of 1000 individuals, with a mutation rate of 50% and the recombination rate is set at 50%. The GP is evaluated over 7 runs, with 100 generations each. A steady-state method is used with 20 new children produced each generation.

Representation:	Parse Tree
Parent Selection:	Over-Selection from top 32% of pop.
Survival:	Rank-Based
μ :	1000 Individuals
λ :	20 individuals
Type:	Non-Generational
Recombination:	Uniform Crossover (sub-tree exchange)
Mutation:	Replacing
P(Recombination):	50%
P(Mutation):	50%

Table 1: GP Experimental Setup Parameters.

4.3 Evolutionary Algorithm Test Bed

For the second tier, four evolutionary algorithms are implemented to evolve neural network weights using the mutation operators from the GP tier. Each EA evolves a neural network for a different type of problem, with varying network architectures. Two of the EA-NNs are feed-forward networks approximating the following non-linear functions:

$$y = 2x^2 + 1$$
$$y = 2x^3 + 1$$

The remaining two EA-NNs are recurrent networks performing time-series prediction on a short (approximately 100 points), randomly generated distribution. The first recurrent network includes a single time-delay element, while the second contains 2 time delay lines, but they are both attempting to predict the same time-series.

4.3.1 Representation

For each of these EA-NNs, the individual representation takes the form of the weight set for the networks, which is a series of floating point values.

4.3.2 Initialization

Individuals are initialized with random weight sets, in the range of $(-6, 6)$ to prevent saturation from occurring within the neural network.

4.3.3 Reproduction

Reproduction is accomplished through recombination and mutation operations, where the mutation operation is provided by the GP tier, and the recombination operation is discrete, in the form of uniform crossover, in order to maintain the importance of the mutation operator for the creation of new weight values. Weight values for each allele are bounded to $(-6, 6)$ to prevent saturation from occurring in the network.

4.3.4 Parent Selection

Parent Selection in the EA-NNs is implemented as Ranking Selection, where individuals in the population are ranked and have a linear relationship with rank to probability of being selected as a parent. The slope of this relationship is a constant selected by hand.

4.3.5 Survival

Survival is performed through a non-generational scheme whereby parents compete with children for survival. This is accomplished by ranking the population by fitness, and the top n individuals are kept, while the rest are destroyed, where n is the maximum population size.

4.3.6 Fitness Evaluation

The fitness function for each EA-NN is the Mean-Squared Error of the network over the respective target function.

4.3.7 Termination

Each EA-NN terminates when it reaches the maximum generation limit for the EA test bed.

4.4 EA Test Bed Parameters

Each EA-NN consists of a population of 50 individuals, with mutation and recombination rates held at 50%. Every EA-NN is run 50 times for 50 generations each and the Population Average MSE from each run is averaged to produce an average MSE for the EA-NN test bed under the current mutation operator.

Representation:	Floating Point Vector
Parent Selection:	Rank Proportional
Survival:	Rank-Based
μ :	10 Individuals
λ :	00 individuals
Type:	Non-Generational
Recombination:	Uniform Crossover
Mutation:	Provided by GP
P(Recombination):	50%
P(Mutation):	50%

Table 2: EA-NN Experimental Setup Parameters.

5 Results

Table 3 shows a comparison of fitnesses between existing mutation operators and the average and best evolved operators. We can see that the average and best evolved operators have significantly higher fitnesses, suggesting superior operators.

Figure 2 shows the average best fitnesses across all 7 runs of the algorithm. We can see that the average does not climb much over 100 generations, showing that population improvement was slow, indicating a low selection pressure.

Table 4 shows the structure of the evolved operators and their fitnesses. Notice that the maximum length of the operators is 7 nodes, well below the maximum of 256.

Figures 3 through 9 show the distributions of existing operators and a selection of evolved operators. Figures 5, 6, and 7 illustrate the ability of the algorithm to evolve diverse distributions. We can see in Figures 8 and 9, the highest fitness operators, that their distributions are very similar to that of the creep operator.

The Two-Sample Kolmogorov-Smirnov Distribution test was performed between the top three operators, shown in figures 7, 8, and 9, and the Creep Uniform Random Variable operator. The results of these tests show that none of high-fitness evolved operators were significantly different than the Creep Uniform Random Variable operator.

Mutation Operator	Avg. Fitness
Creep Uniform Random Variable	2571.409
Replacing Uniform Random Variable	1952.391
Average Evolved Best Fitness	4252.331
Best Evolved Fitness	7705.722

Table 3: GP-EA-NN Results

Run	Mutation Operator	Fitness
1	$((WEIGHT \div 0.481581) - (WEIGHT + RAND))$	7705.721709
2	$(RAND - (WEIGHT \div -0.802181))$	5679.963097
3	$(RAND + (WEIGHT + (RAND * 0.697299)))$	5522.531617
4	$(RAND + (WEIGHT - (WEIGHT * RAND)))$	2901.297335
5	$((RAND + WEIGHT) \div 0.786494)$	2845.060655
6	$((WEIGHT - RAND) \div 0.835222)$	2666.738725
7	$((0.913753 \div RAND) + WEIGHT)$	2445.021015

Table 4: Evolved Operators

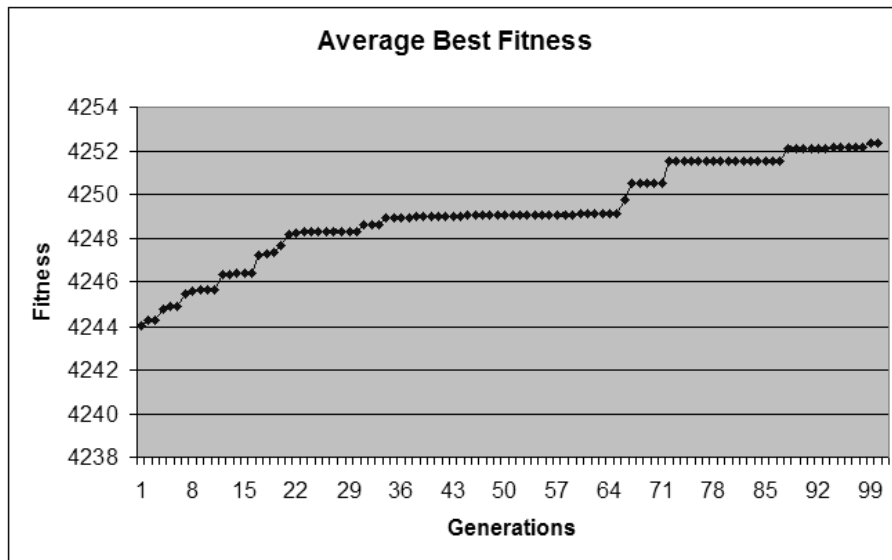


Figure 2: Average Best Fitness over 7 runs.

6 Conclusion

A hierarchical structure for the evolution of EA-NN mutation operators was presented, with results indicating that the existing creep uniform random variable operator is still the best operator on average for evolving neural network weights. The high-fitness results of the experiment could be attributed to anomalies introduced by pseudo-random number generation, lack of data or poor choice of experimental setup parameters.

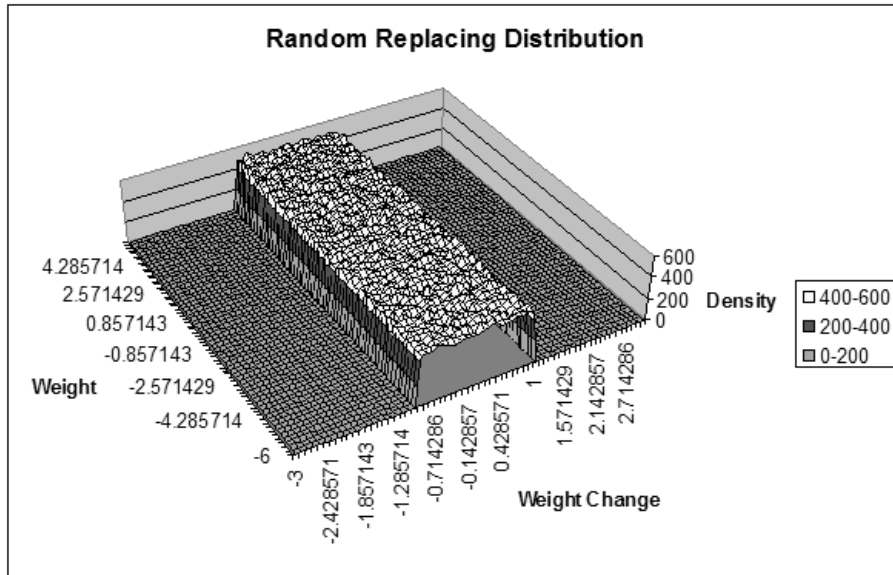


Figure 3: Replacing Uniform Random Variable Distribution

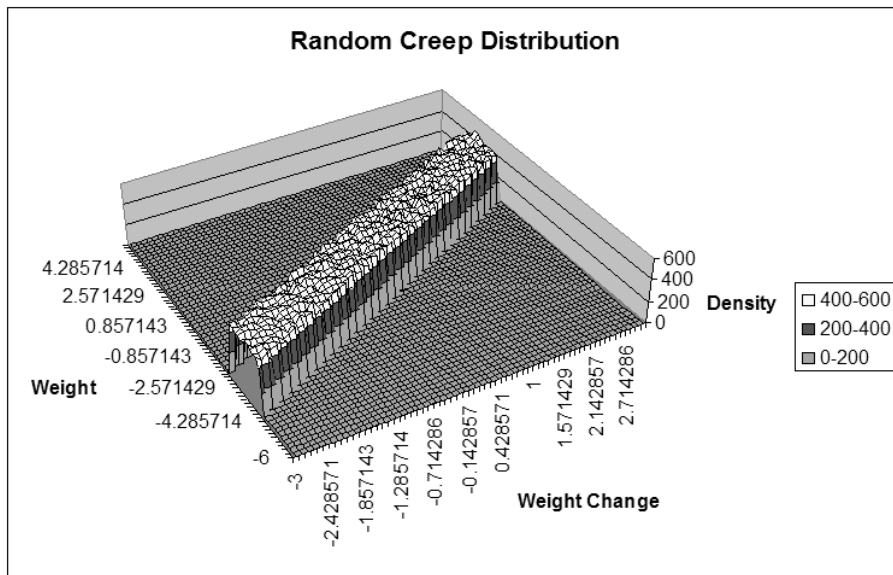


Figure 4: Creep Uniform Random Variable Distribution

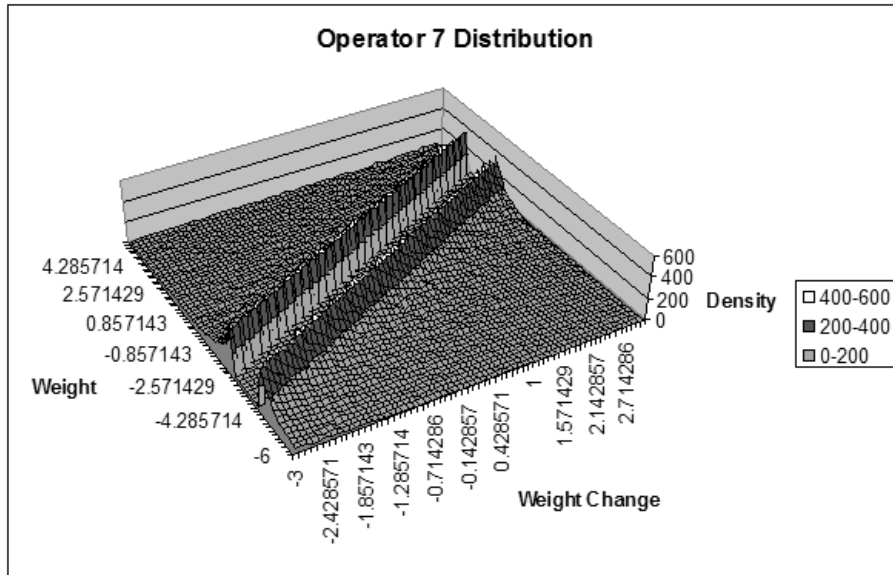


Figure 5: Evolved Operator 7 Distribution

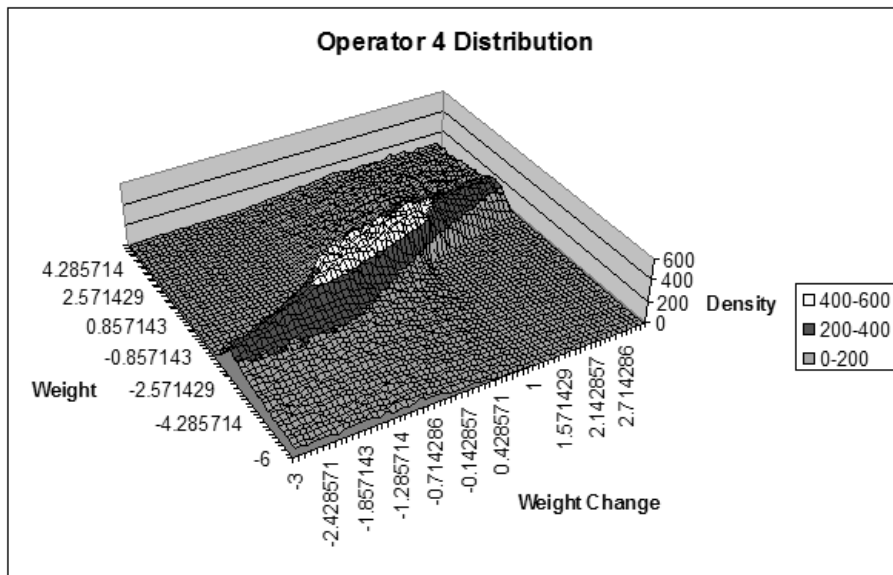


Figure 6: Evolved Operator 4 Distribution

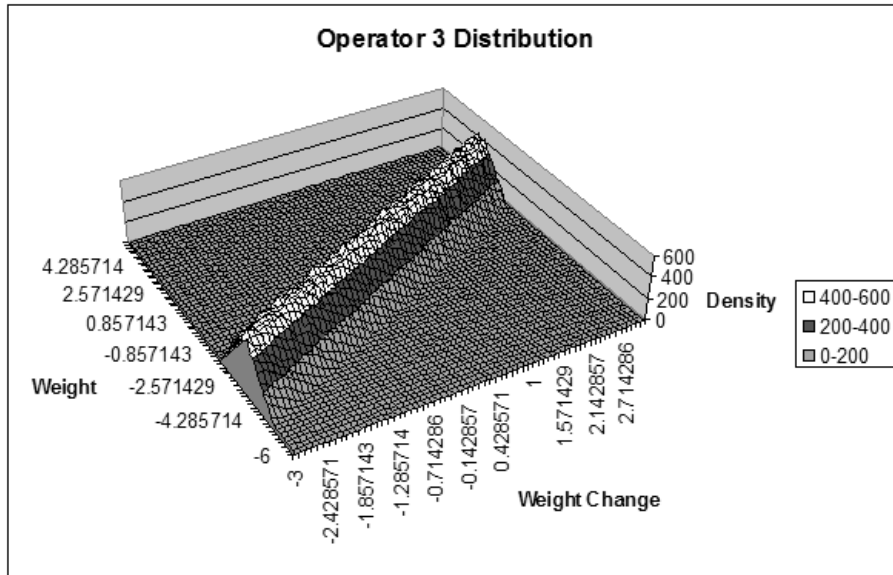


Figure 7: Evolved Operator 3 Distribution

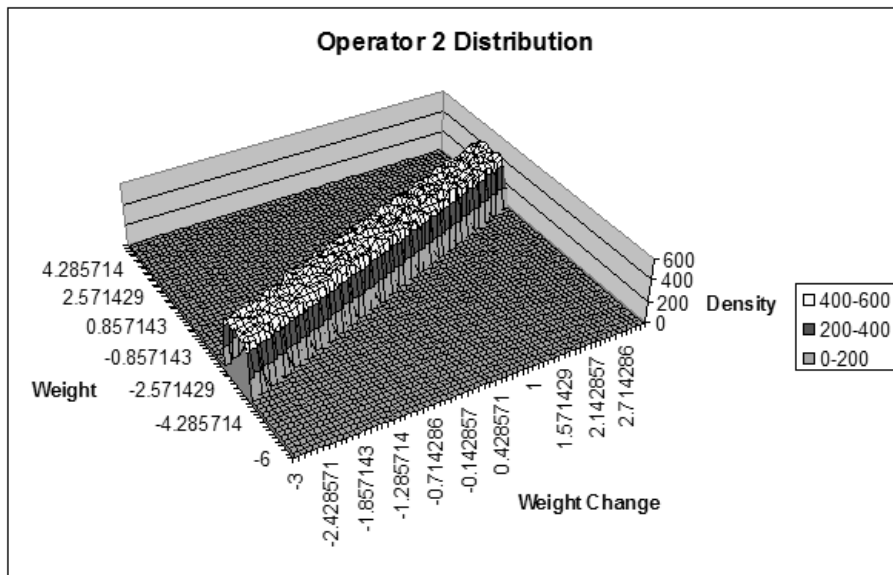


Figure 8: Evolved Operator 2 Distribution

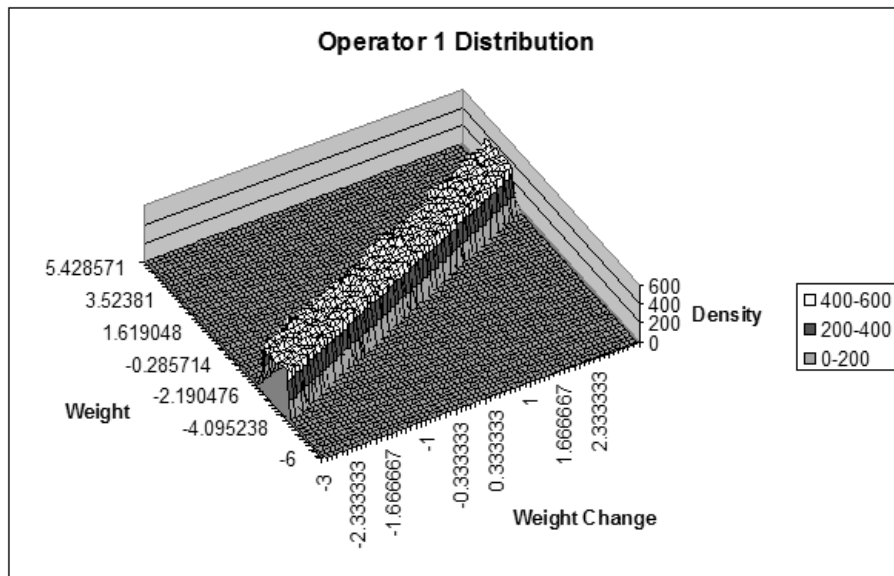


Figure 9: Evolved Operator One Distribution

7 Future Work

Future work can include additional runs to find the true likelihood of evolving a better operator, and an increase in selection pressure to promote evolution of fitter individuals. Also, the use of standard benchmark problems, such as those presented by Koenig(2002) would lend credibility to the results of the search. The inclusion of trigonometric functions and constants in the functional and terminal set of the GP would better facilitate the construction of new distributions, resulting in the inclusion of Cauchy, Levy and Gaussian distributions in the GP solution space, in addition to the uniform random distributions.

References

- [Eiben2003] A.E. Eiben and J.E. Smith *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin Heidelberg, 2003, ISBN 3-540-40184-9.
- [Engel2002] Andrias P. Engelbrecht *Computational Intelligence: An Introduction*. Wiley, 2005, ISBN 0-470-84870-7.
- [Schwefel1975] H-P Schwefel *Evolutionary Strategies and Numerical Optimization*. PhD Thesis, Technical University, Berlin, 1975.
- [Rechenberg1973] I Rechenberg *Evolutionary Strategies: Optimizing technical systems with Principles of Biological Evolution*. Frammann-Holzboog Verlag, Stuttgart, 1973.
- [SuttonBarto1998] Richard S. Sutton and Andrew G. Barto *Reinforcement Learning: An Introduction*. MIT Press, 1998, ISBN 0-262-19398-1.
- [SalamaHingston1995] *Evolving Neural Network Controllers*. Evolutionary Computation, 1995., IEEE International Conference on Volume 2, 29 Nov.-1 Dec. 1995 Page(s):579 - 583 vol.2 Digital Object Identifier 10.1109/ICEC.1995.487448
- [Koenig2002] Andreas Koenig *A Study of Mutation Methods for Evolutionary Algorithms* CS447 Semester Project, Fall Semester 2002. <http://web.umn.edu/tauritzd/courses/ec/fs2002/project/Koenig.pdf>