EVOLVING INTELLIGENT MULTIMODAL GAMEPLAY AGENTS AND DECISION MAKERS WITH NEUROEVOLUTION

By

IAN PATRICK FINLEY

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, Oklahoma

2012

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2015

EVOLVING INTELLIGENT MULTIMODAL GAMEPLAY AGENTS AND DECISION MAKERS WITH NEUROEVOLUTION

Thesis Approved:

Dr. Gary G. Yen

Thesis Adviser

Dr. Guoliang Fan

Dr. George Scheets

.

ACKNOWLEDGEMENTS

I want to begin by thanking my advisor Dr. Yen, whom I hold the upmost respect for as a mentor and have gained a lasting friendly relationship through my years of study. I want to thank him for the overwhelming help and support he provided. Without his motivation, direction, and kind heart, my success would not have been possible.

To my grandparents, Sam and Donna Finley, I wish to convey my unlimited gratitude and adoration. I thank them for their lifelong investment in me which is a debt I could never repay. I will forever cherish their wisdom, encouragement, and love.

To my mother, Kelley Jaques, I want to thank for molding me into the person I am. I want thank her for all the books on my birthdays, for "Muzzy", for all the brains that grossed me out, and for providing the constant fuel that kept my curiosity aflame. I want to thank her for her unyielding compassion and devotion as I could not ask for a better mother.

I want to thank my father, Brad Finley, for being my rock, for his guidance, for giving me the shirt off his back, and for teaching me to put everything that I have in whatever I do.

I am also thankful to all of my family and friends who gave me their support and inspiration which include Jim, Ronnie and Jayme Gardner, Tim, Robin, Marshall and Emma Finley, Michelle and Hannah Breeding-Finley, Dylan Finley, Shelley and Michael McMillan, Mike and Pattie Jaques, Renee, Art and Lilia Rousseau, Chris Dinges, Cameron Carroll, Michelle Ondak, Corey Vyhlidal, Derek Milligan, Hoang Nguyen, and Matt Creech.

Finally, I would like to thank Carolina Arbona for being my inspiration, for her care, and for helping me be the best version of myself.

Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: IAN FINLEY

Date of Degree: DECEMBER, 2015

Title of Study: EVOLVING INTELLIGENT MULTIMODAL GAMEPLAY AGENTS AND DECISION MAKERS WITH NEUROEVOLUTION
Major Field: ELECTRICAL ENGINEERING

Abstract: ‘Super Mario Bros’ is a difficult platforming game that requires the use of multiple behavioral modes to complete different gameplay elements such as: collecting coins, dodging enemies and getting to the end of the level. Methods for creating intelligent game playing agents have previously used human designed behavior policy for each gameplay state or by combining gameplay goals into a single task to be learned. This thesis assesses the development and method of training machines to promote multiple modes of behavior within neural network controllers. These controllers utilize the concept of evolution through multi-objective optimization for the test bench platform game system ‘MarioAI’. Artificial neural networks were evolved to exhibit complex and multimodal behavior using multiple sub objectives of the game; and thus overcome the non-linear, noisy, and fractured game environment. Experiments were conducted with the purpose of creating multiple Pareto-optimal solutions of quality with differing behavioral aspects. These solutions were then discerned by a Decision Maker Neural Network Ensemble that had been evolved to pick the best solution according to game level. This Decision Maker Ensemble proved to be able to learn on minimal information and provide the highest overall game score. The results of this thesis show that it’s possible to train agents on sub objectives to teach multiple forms of complex behavior that can then be abstractly chosen by an evolved Decision Maker to provide a better outcome than agents that were trained specifically towards that single solution.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Dr. Michio Kaku said, “Consciousness is the process of creating a model of the world using multiple feedback loops in various parameters (e.g., in temperature, space, time, and in relation to others), in order to accomplish a goal (e.g., finding mates, food, and shelter) [1]. Any of these “goals” will require the combination of multiple behaviors and are integral to completing the task. Therefore a large aspect of intelligence is the ability to exhibit multiple forms of behavior in symphony. Anything that shows this form of consciousness is then to be considered intelligent. It can then be said that to design intelligent machines, these machines must be created to produce multiple forms of behavior.

Computational Intelligence (CI) is the study of using ideals of nature to create intelligent systems that exhibit multiple forms of behavior in a complex and changing environment. Many complex real world CI problems come from trying to discover these behavioral policies: Robotics, Drone Control, and even Non Playable Character (NPC) control in video games. In the real world however, there are a myriad of possibilities to account for and would be too complex to create a system for every possible action *a priori*. These problems also suffer from the issue of being difficult to determine what even is the “correct” and “incorrect” action? This is due to the overall reliance on the developer to have a complete understanding of the environment being studied. Therefore with the complexity and inherent uncertainty of these problems, it would be better for systems to learn multiple forms of behavior, than to develop them explicitly.

1.1 Motivation

CI machines have been used to solve a wide array of problems. Mobile robotics require control laws and mechanical design to overcome difficult terrain [2]. Artificial Intelligent design for simulated car racing requires differing driving strategies for different track types and obstacles [3]. Intelligent opponent design in video games require the need to switch between different play styles to account for aggressive or defensive patterns to create interesting variations and difficulty game play against humans [4]. These problems all require the ability to switch between tasks and thus need to learn and exhibit multimodal behavior. Multimodal behavior describes the ability to exhibit multiple, different, even conflicting forms of behavioral modes.

Modeling these different forms of behavior poses to be a difficult challenge. Many forms of standard behavioral design are not always applicable for modeling real systems like the ones described above, as it can be unclear as to what behavior is required. Additionally it becomes difficult to discern and design when to switch between behaviors. A promising learning approach for this issue is through the use of *Neuroevolution* (NE). This method uses direct interaction with the environment to provide feedback in the form of task objectives (i.e., climbing the most stairs, or getting the highest game score). These objectives then drive behavior learned within neural networks that have been adapted through evolutionary strategies. Many forms of Neuroevolution strategies use only a single main objective making it difficult to learn the multiple modes of behavior necessary in creating a robust machine. Therefore this thesis proposes the learning of multimodal behavior through the use of Multi-Objective Neuroevolution. By using training controllers on multiple conflicting sub-goals instead of the overall goal, multiple complex behaviors can be learned even in difficult environments. As a case study, the controllers will be designed to create intelligent video game playing controllers for the difficult platforming game "Super Mario."

CHAPTER II

REVIEW OF LITEREATURE

The main idea for this thesis is the development of multimodal behavior through multi-objective, Pareto-based Neuroevolution. This idea and its foundations are visited in this chapter as well as the Mario AI test bench and the difficulties that are inherent to the game.

2.1 Genetic Algorithms

This section explains the history and relevance of Genetic Algorithms, which are a subset of Evolutionary Algorithms (EA), for the use as a searching algorithm.

Developed by John Holland in 1962, Genetic Algorithms (GAs) are an adaptive and stochastic population based searching method based on Darwinian's theory of Evolution [5-7]. The objective of GAs is to search a decision space for a set of decision variables that optimize some fitness or performance model in the search space.

GA methods follow four key elements. First, the genetic pool must have variety. This means in order for the population to continue to evolve, there must be some mechanism that drives this variation. Second, the child of two parents will contain some of the genetic variations passed down between the two. Next, in every generation, more offspring are produced than can survive.

And finally, only the strongest are permitted to survive. The individuals within the species, the phenotype, that go on to reproduce or who reproduce the most, are those with the most beneficial genetic variation. This is the idea of "natural selection, and the backbone of GAs. The heuristics of the algorithm can be seen in the flowchart below.

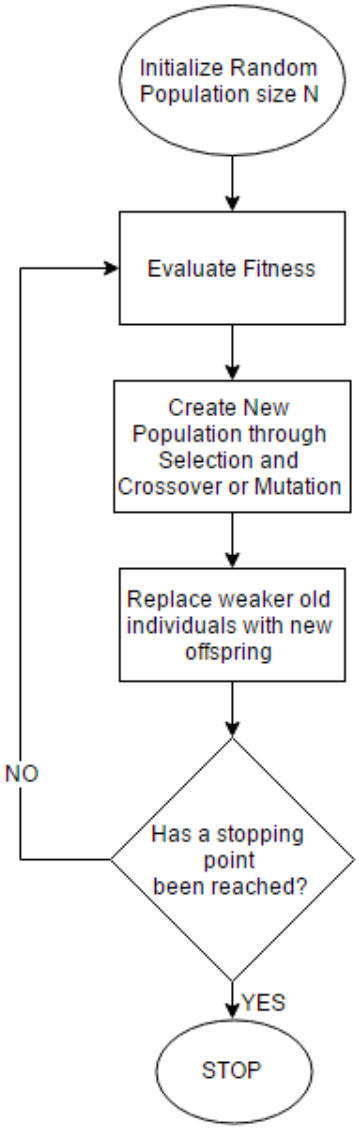


Figure 1: Flow Chart for GAs

The first step is to initialize the population of individuals, where each individual in the population represents a point in the search space. This space is determined by the set of variables, the genotypes that make up the properties of the individual, the phenotype. These variables will have the form of some direct or indirect encoding and are usually represented as a binary string in the case for basic GAs. Other forms of variable encoding can be discrete, tree based, real, or any other form that allows for distinct properties to be separable and searchable in a collection. More will be explained on the types of encoding in later sections. The genotype then must be mapped to the phenotype for evaluation.

Evaluation of the population is based on an optimization schema where the individual that results in the best performance in the system are given the highest fitness value. The fitness function chosen determines the effectiveness of the overall process. Also, for real problems, this becomes the most computationally costly step for the overall process and tweaks and tradeoffs must be found for both a robust and time efficient convergences of solutions.

Next there needs to be a strategy to select and ensure that the best individuals have a better chance of being chosen to reproduce and generate offspring. This selection mechanism will drive the population to convergence. Care must be taken, however, so as to allow lesser individuals to also mate, this will guarantee that the gene pool does not become saturated and stagnant as even the less prominent individuals in the population may hold some advantageous genetic information. A powerful selection strategy is through tournament selection. Some number k of individuals, where k is the tournament size, is randomly selected from the population as contestants. If tournament size is equal to one, then the selection strategy becomes random selection. Conversely, if tournament size is greater than four, then diversity is lost as about 50% of the population is lost in the selection process [8]. Therefore a binary tournament (tournament size of 2) is usually chosen.

The contestants are then compared via their fitness value, and the individual with the best fitness value is deemed the champion and is allowed to mate with another champion of another tournament.

Upon selection of parents, they are then allowed to mate and generate single or multiple offspring. These offspring will then have a new variant of their parent's genotype. This source of variation comes from genetic operators that allow for both exploitation and exploration of the search space through future generations. Exploration is the method of individuals spanning the entire search space in hopes of finding the best solutions, and exploitation is the method of searching around these solutions in hopes of finding the overall best. The sharing and mixing of information between two parents is the exploitation operator and is called the crossover. This allows for the possibility of improving on the selected best individuals. For example, one such crossover is 1-point crossover where each parent swaps a randomly selected point in their genotype with that of the other parent as shown in Figure 2 below.

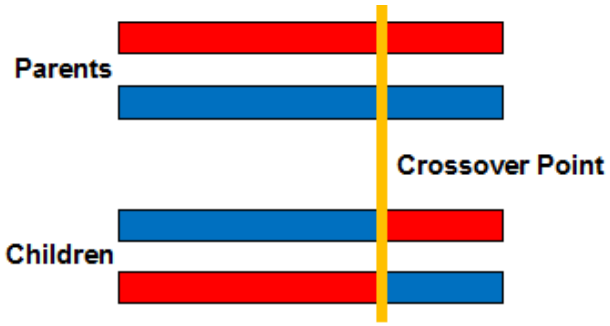


Figure 2: 1-point crossover

This then allows for the population to grow in the direction of the global optimum solution. However, too much exploitation may cause the population to possibly get trapped in a local optimal solution and perform poorly.

Every child developed from mating also has a small probabilistic chance of being a mutant. This is the exploration mechanism specified for escaping local optima and reaching the entire search space. This operator will cause random perturbation through the child's chromosome. For real GAs this can be through Gaussian noise $N(0, \sigma)$, where 0 is the mean value and σ is the standard deviation.

Thus for each parameter, x_i that has been selected for mutation will face some Gaussian perturbation:

$$\dot{x_i} = x_i + N(0, \sigma_i)$$

If the probability of mutation is too high for GAs, then exploration will become too high and the searching algorithm will be reduced to random search without proper convergence.

This selection and generation of children continue until a new intermediate population of children is of size *N*. The new population is then used to compare and replace unfit older individuals. This replacement strategy allows continual injection of new and hopefully improved variations in genetic information and lets the population drift in the direction of global optimal solutions.

After a new population has been formed, a stopping criterion is checked. This is usually in the form of maximum generations or a certain threshold of the fitness function is found. If the criterion is not met then the process reiterates. Otherwise, when the last generation completes, the individual with the best fitness value is presented as the solution when only a single objective is concerned.

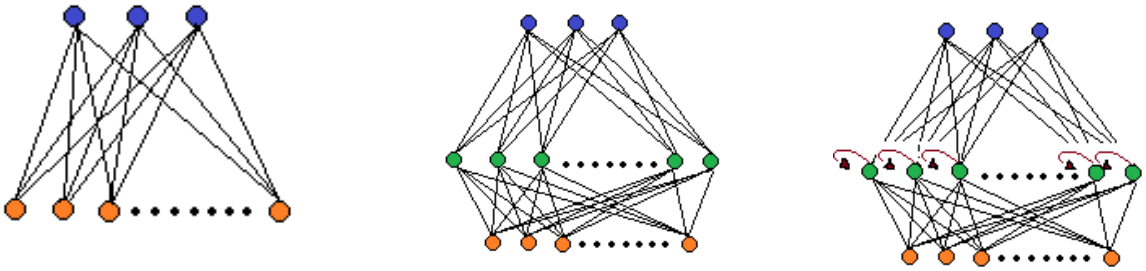
GAs are especially useful within partially observable domains with little knowledge on the system, as GAs don't require predefined sets to train offline or prior analysis of the problem domain. GAs can then be paired with neural networks as a surrogate model to solve Reinforcement Learning (RL) problems, even when the correct output or policy is unknown. Thus, genetic algorithms are a useful method for solving continuous RL problems.

2.2 Artificial Neural Networks

This section will introduce neural networks as a surrogate model for RL problems, as well as some basic history and concepts specific for Neuroevolution.

Neural Networks (NN) are biologically inspired by their real world counterparts. Neural structures are highly interconnected system of parallel electrically excitable cells called neurons that possess three main components: The dendrite, the cell body, and the axon. The dendrite acts as an input receptor that carry signal to the cell body. The cell body then takes the collective sum of all the input signals as a voltage change. If a threshold is met, an action potential signal can then be passed as an output through the axon where synaptic connections are activated upon arrival with other neurons. Following this neural design, neural networks have the possibility to theoretically learn and approximate any function. This makes Artificial Neural Networks (ANN) a great candidate for regression, classification and control and prediction problems [9].

There are a variety of different NN types that differ in size, connection types, etc., but the simplest NN are the feedforward network. These networks have the information as inputs travel forward throughout the system to be manipulated as new outputs. In feedforward networks, there are no synaptic connection that loops back to provide any time dependent information.



(a) SLP **(b) MLP** **(c) RNN**

Figure 3: Artificial Neural Network Architectures: (a) Fully connected Single Layer Perceptron (SLP). (b) Fully connected Multilayer Perceptron (MLP). (c) Recurrent Neural Network (RNN) with recurrent hidden layer Neurons

The simplest form of the feedforward network is the perceptron; see Figure 3(a). This is an ANN with only an input layer that connects directly with an output layer. This type of simple network can only learn to solve linearly separable problems, and therefore is limited to only simple classification and control. Therefore, another ANN architecture, called the Multilayer Perceptron (MLP), must be explored. This type of perceptron has an input layer, a hidden layer, and an output layer; see Figure 3(b). Each layer has as activation called the transfer function, f, which computes the weighted sum of the previous layer's inputs. The transfer function then allows for the combined inputs to be squashed into the range. The most common transfer function used for Neuroevolution is the log-sigmoid transfer function as it allows the inputs to be squashed to a range of 0 to 1 and activation of an output could then be deterministically chosen by some threshold.

Thus the output, **α**, for each neuron, **j**, with an input **p**, in a layer with the Log-Sigmoid transfer function ***f*** is:

$$\alpha_j = f(n_j)$$

$$where\ n_j = \left(\sum_{i} w_{i,j} p_i + b_j\right)$$

$$Then\ \alpha_j = \frac{1}{1 + e^{-(n_j)}}$$

Where *w* is a connection strength between the node i and j and determines the "steepness" of the sigmoid**,** and *b* is the bias that is injected into the system that allows for the sigmoid to be shifted. For this thesis, however, the bias is not used as it injects additional complexity in the search space.

Using this methodology, a MLP can learn problems that are complex and not linearly separable, and in theory, can learn any function with a large enough number of hidden neurons [10].

Another form of ANN architecture is the feedback NN called Recurrent Neural Networks (RNN). These networks have the ability to form cycles that allows information to propagate back into previous layers. Through these cycles, this network can then learn temporal properties of the system. This means that a RNN can learn and preserve memory of past events.

Most classical neural networks train and adapt the weights using a method known as gradient descent using backpropagation [9]. The standard method requires a training set, a testing set, and a validation set for training and evaluation. This type of training is not helpful when dealing with real RL problems. Firstly, collection of good data for each specific stage of this method is expensive and the quality of the data is hard to define. Second, even if the data is proven to be good, it might not lead to the most optimal solution.

Genetic Algorithms, then, provide a strong alternative, or complement, to back-propagation [11]. As explained in the previous section, Genetic Algorithms have a distinct ability to genetically encode most anything to a phenotype. Therefore instead of other learning methods, GAs can easily define several parameters of neural networks as genetic information and evolve the network based on a performance criterion which is more flexible than the definition of an energy or error function. Gradient descent methods also suffer with entrapment in local minima error surface. This problem is addressed with GAs as it is has multiple points samplings on the error surface while also searching different parts of the decision space using the genetic operators. The fitness function of a GA also does not depend on gradient information and thus does not have to be differentiable or even continuous.

GAs can then be seen as an efficient replacement for when gradient decent methods suffer when the error function becomes nondifferentiable and complex, especially when gradient information is too costly to attain. GAs then provide a flexible and robust solution to optimize ANN weights for large, complex, non-differentiable, and multimodal spaces [12]. Given than many real world and RL problems share these aspects, there has been much research in the application of evolving ANN weights and parameters [13].

2.3 Neuroevolution

Neuroevolution (NE) is the combination of the approximation superiority of ANN with the searching power of GAs. With NE, GAs can encode and evolve ANNs to find a set of parameters (either the weights and/or the topology) to solve many complex computational problems in both the real world and the virtual. However, recurrent networks have been shown to have similar results to that of feedforward agents in platform games and are also outside of the scope of this thesis.

Computation Intelligence research in games has been finding more and more traction, as video games provide a new form of real life problems that can be tested through simulation of games inexpensively.

Neuroevolution is a strong competitor for solving many tasks necessary for games. Game strategies could be learned through Q-learning or TD learning, modeling efficient non-player characters can be used with Support Vector Machines [14]. When in the domain of games, there are specific issues that these methods must specifically handle with significant prejudice to overcome. Video games have a very large state/action space due to the large variety of objects and enemies that require different possible actions to be handled at each time step. It is also very important to be able to not have to check each action at each time step to evaluate an action's value like traditional RL. This could become very computationally complex as the number of actions to check and the number of agents in the game increase. Another issue is the consistency of fast behavioral adaptation and complex behavior learning. Since the evaluation in real world applications also pertain to video game simulation, the time necessary to create intelligent agents through typical RL techniques might be in the range of hours and days. And lastly the ability of creating diverse agents is highly desirable in video games. Many traditional learning methods for RL have guarantee of convergence, but risk convergence to reduce all agents to the same principles and behaviors.

Neuroevolution has been shown to be a strong method for solving all of these problems. They work well in high-dimensional spaces as they can learn to generalize very well and do not need to evaluate each action at each time step. Diversity is maintained in the population of agents through the diversity imposing mechanisms specific to GAs. Neuro-evolved agents also can quickly learn approximations for simple behaviors and will discover more complex behaviors later, allowing for faster learning.

Neuroevolution has been shown to have record beating performances in both control systems and strategy selection in complex decision domains [15] [16]. It is scalable, robust, allows for complexification of neural networks, and allows for open-ended learning through exploration and exploitation. It also has brought about new kinds of games such as “Galactic Arms Race” where space ships are given the ability to evolve its weapons instead of having pre-built designs [17].

For the case of reinforcement learning in video games, Neuroevolution has been used in many forms. One such form is in a game called NERO developed by Stanley, Bryant, and Miikkulainen, where robotic agents were evolved by each player by rewarding positive game task outcomes. These agents were then able to learn how to capture battle points and evolve dedicated game policies [18].

It has been shown that Neuroevolution has multiple applications in games for research, testing, and also opening new fields of gameplay possibilities. Each differentiation between NE methods is based on the role of the NE agent for a particular game problem. When developing NE agents for RL control problems, a large issue is creating intelligent agents to handle multiple forms of states/actions. Many methods use a predetermined policy and reward system for each state. This, however, limits learned behavior to that of the predetermined policy, and therefore restricts the agent from finding the “best” action for each state. These methods also usually train towards a single goal and could miss nuances in an environment that requires differing behavior to complete conflicting sub goals for optimal performance. The “behavior” of an ANN is generally determined by the architecture of the NN and the weights of the connections. Therefore, if multimodal behavior is to be learned, the possibilities of multiple objectives to drive weight optimization for multimodal decision making must be explored.

2.4 Multiple Objective Pareto Optimality

As it can be seen, Neuro-evolved controllers can be used in a multitude of problems but domains might require multimodal behavior due to conflicting objectives. In the case of Hong and Cho, a GA was used to evolve agents to play in a game called Robocode [19]. The agents were evolved to select a set of pre-defined simple state/action strategies based on the different individual behavior of the opponents. However, each agent that was evolved suffered from poor generality. Some agents that were good at offensive objectives, could not determine the correct behavior if the roles were switched from predatory to prey. Therefore a way of dealing with multiple conflicting objectives is necessary. One way to accomplish this is to create a combined fitness as a weighted sum of each n fitness objectives:

$$F(x) = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n, \qquad 0 \leq w_n \leq 1, \qquad \sum_{1}^{n} w_n = 1$$

Where *w,* are the specified weighted tuning parameters for each respective fitness function. This, however, does not guarantee that multimodal behavior will be learned, as even though all objectives may be represented, the tuning parameters have to be very specific as to drive the population to each specific task. Not only that, but a single combined objective will return a single optimal solution from the multi-objective space. A one dimensional objective solution cannot find multiple optimal solutions on non-convex surfaces, but rather points that are tangential to the surface. Therefore another way of handling multiple objectives is necessary and can be found through Pareto optimality.

Pareto optimality is the concept of comparison of points in a decision space represented by their mapped equivalent in the objective space. The comparison between solutions in the objective space is through criteria called dominance denoted as $\succ$.

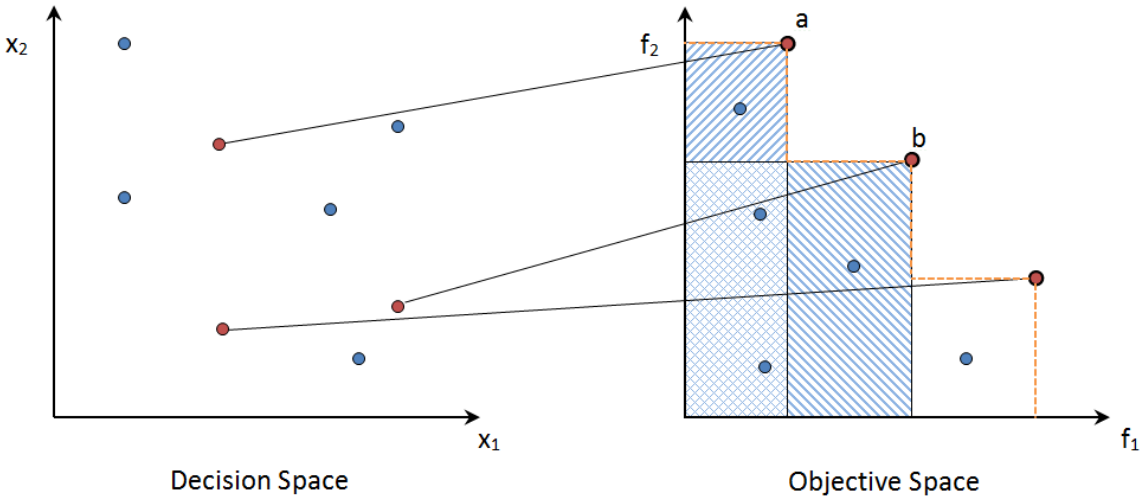


Figure 4: Mapping from decision space to Objective space for multiple objectives

The figure above, Figure 4, is an example of a two dimensional maximization optimization problem where each point in the objective space represents a corresponding fitness value in the first and second objective. Point **a** in the objective space is said to dominate all solutions with the upwards diagonal shaded area beneath it. Similarly, all points within the downward diagonal shaded area are dominated by the point **b**. Any point in the light shaded cross area are considered to be dominated by both **a** and **b.** Any points that are not dominated by any other point are considered to a member of the "Pareto optimal front" and are depicted as the red points connected by the orange dotted line in Figure 4 . These points are considered to be the "best" of the two objectives, as no one point is better than another in terms of their fitness. This can be seen as point **a** has a high value in the f_2 direction, but has a lesser value in the f_1 direction than point **b.** A formal definition of Pareto Dominance can be seen next:

Definition: (Pareto Dominance): *The minimization of n components* $f_k, k = 1, \dots, n$ *of* a *vector function* $\boldsymbol{f}$ *of a vector variable* $\boldsymbol{x}$

$$\boldsymbol{f}(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \dots, f_n(\boldsymbol{x})\big).$$

Vector $\vec{u} = f(x_{u)} = (u_1, u_2, \dots, u_n,)$ *dominates vector* $\vec{v} = f(x_v) = (v_1, v_2, \dots, v_n,), \vec{u} \succ \vec{v}$ *if:*

$$\forall i \in \{1, \dots, n\}, u_i \geq v_i \; and \exists i \in \{1, \dots, n\} | u_i > v_i$$

Then a set is Pareto optimal if it contains all the points *x* that are not dominated by any other points *y*. This set makes up the Pareto front and represents the best solutions found by the searching algorithm, where each subsequent front after that is the next best front. A popular and powerful searching algorithm that utilizes this optimality is the Non-Dominated Sorting Genetic Algorithm II.

2.5 Non-Dominated Sorting Genetic Algorithm II

The Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [20] is the successor to the multi-objective optimization algorithm NSGA [21]. The preceding multi-objective algorithm NSGA imposed many helpful aspects for searching the multi-criteria objective space. It allowed for the assignment of fitness based on non-dominance and created progression in the direction of the Pareto optimal region, one front at a time. It also imposed a sharing hyper parameter σ**,** that allowed for a diverse solution in the parameter or objective space. This however came with disadvantages that NSGA-II improves upon. Firstly, NSGA was computationally expensive in the sorting stage, where the complexity was of $O(MN^3)$ (where M is the number of objectives and N is the population size.) Therefore, NSGA could not efficiently handle large population sizes. Next, NSGA did not have any form of built in elitism, where elitism is the preservation of strong past solution set so as to not lose the best solution to future generations. Most forms of single

objective GAs have a separate elite set for which offspring are compared to which helps drive convergence to the global optima. The lack of elitism in NSGA slowed the performance of the algorithm as well as resulted in poor performance of solutions. Lastly, the implementation of the fixed sharing parameter resulted in the need to highly tune for performance as the algorithm was highly sensitive to this parameter

NSGA-II improves upon all of these problems as it reduces the time complexity of the rank and sorting of non-dominated solutions to $O(MN^2)$, which again for real world applications is negligible as the time complexity is based on the simulation time but still improves upon the overall calculation time. The lack of elitism is handled as NSGA-II contains population storage of $(\mu + \lambda)$, where $\boldsymbol{\mu}$ is the parent population of size μ, and $\boldsymbol{\lambda}$ is the offspring population of size λ and for this thesis and most NSGA-II problems $\mu = \lambda$. More on this inherent elitism strategy and selection will be explained later. Lastly, the diversity of the population is handled dynamically through comparisons crowding distances as opposed to trying to determine the best sharing parameter value a priori through rigorous testing.

When converting from the single objective to the proposed multiple objective optimization algorithm, NSGA-II, the heuristics for the GA must change and can be seen in the flowchart on the following page.

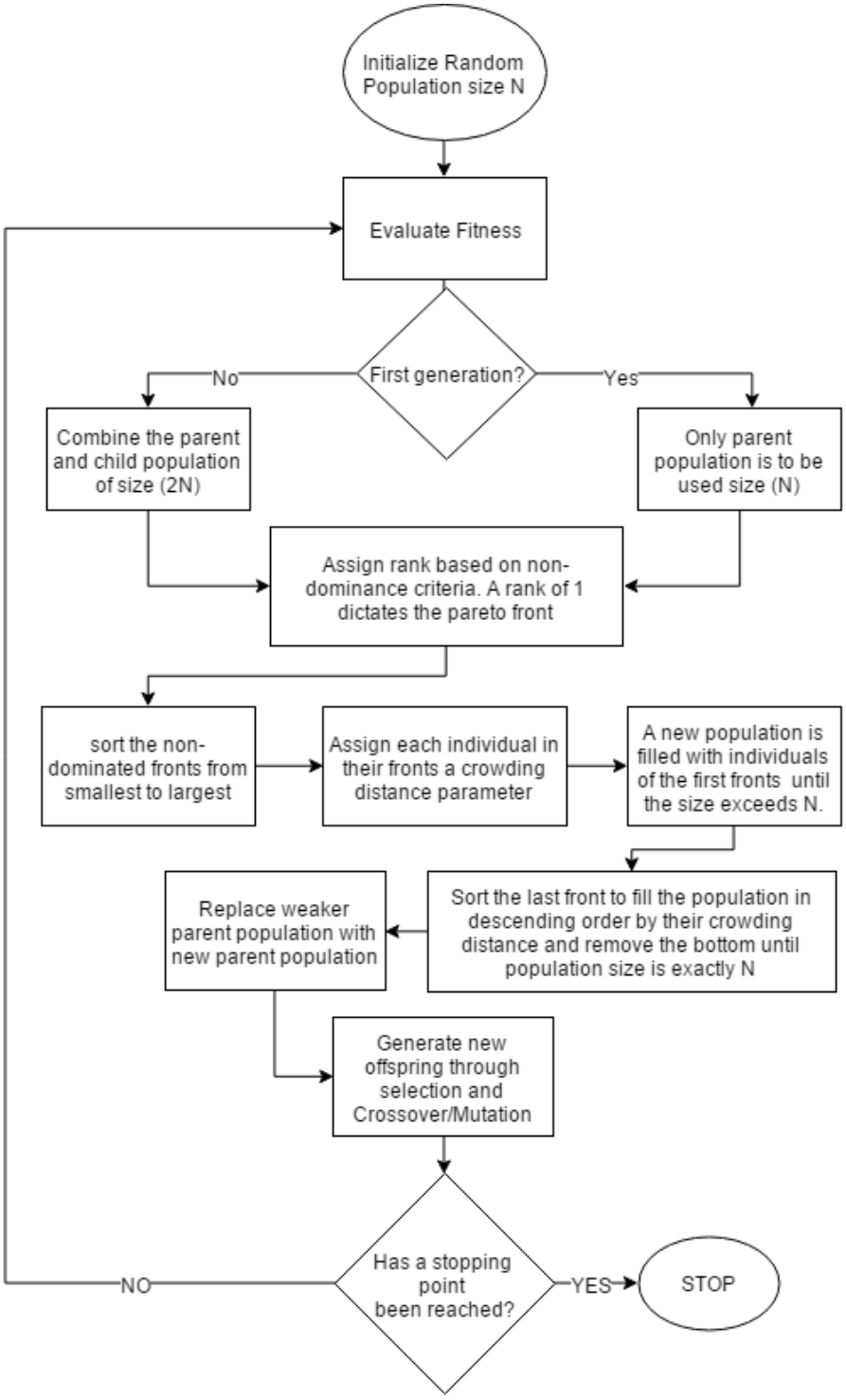


Figure 5: NSGAII Flowchart

2.5.1 Non-Dominated-Sort

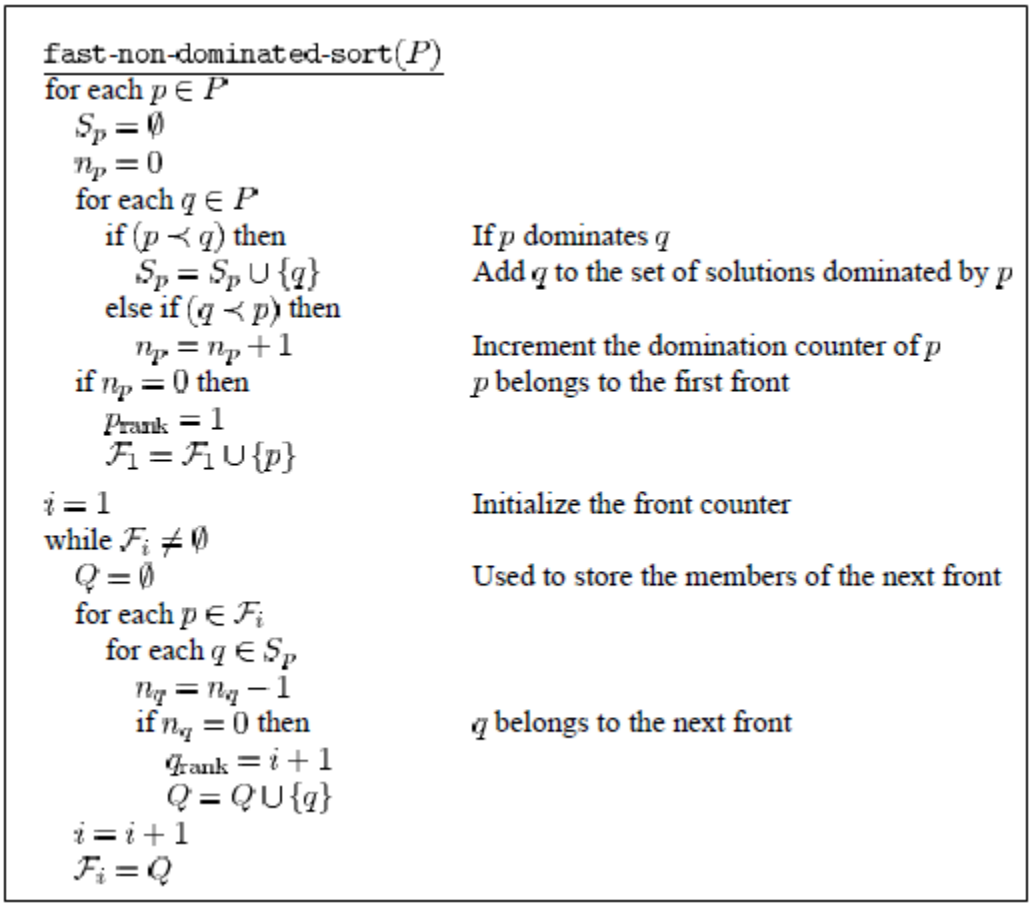


Figure 6 fast-non-dominated-sorting of NSGA-II: [20]

The approach to identifying the best solutions, when multiple objective are concerned, is through the ranking and sorting of individuals by their non-dominance. The formal algorithm proposed by ([20] Deb et al. 2002) for the "fast-Non-Dominate-Sorting" method can be seen above. Each individual, *p*, in the combined population will be compared to all the other individuals, *q*, within the population *P***.** If *p* dominates *q* then *q* is added to a domination set of solutions dominated by *p*. However if *q* dominates *p*, then a domination counter will be incremented. This reiterates until each individual *q* within *P* is compared against. If the domination counter for *p* is zero, then that individual is a Pareto optimal solution and can be placed within the first front.

This continues until every individual is compared to each other throughout the population and all the non-dominated solutions are placed within the Pareto front. The individuals are then sorted into each subsequent front by searching through the remaining individuals in the set of dominated solutions and decrementing their domination counter. When their counter goes to zero, they are then placed in the next available front. This continues until there are no more individuals to place into a front. Measures can be taken to reduce this sorting so as to only take in *N* individuals. This then allows the individuals to convert from a multidimensional comparison to a single dimensional sorted ranking scheme to be used for selection.

2.5.2 Crowding Distance Assignment, Sorting, and Preservation.

After the population is ranked, each individual is then assigned a crowding distance value. This value is used as a niching scheme for the selection process to promote the spread of individuals. To dynamically assign this crowding distance, an individual of a specific front in objective space calculates their Euclidean distance to that of neighboring individuals within the same front. This distance then represents an estimation of the density of individuals surrounding that particular individual. The formal algorithm for assigning the crowding distance can be seen below.

crowding-distance-assignment($\mathcal{I}$)

$l = |\mathcal{I}|$ — number of solutions in $\mathcal{I}$

for each i, set $\mathcal{I}[i]_{\text{distance}} = 0$ — initialize distance

for each objective m

  $\mathcal{I} = \text{sort}(\mathcal{I}, m)$ — sort using each objective value

  $\mathcal{I}[1]_{\text{distance}} = \mathcal{I}[l]_{\text{distance}} = \infty$ — so that boundary points are always selected

  for $i = 2$ to $(l - 1)$ — for all other points

    $\mathcal{I}[i]_{\text{distance}} = \mathcal{I}[i]_{\text{distance}} + (\mathcal{I}[i+1].m - \mathcal{I}[i-1].m)/(f_m^{\max} - f_m^{\min})$

Figure 7 crowding-distance-assignment of NSGA-II: [20]

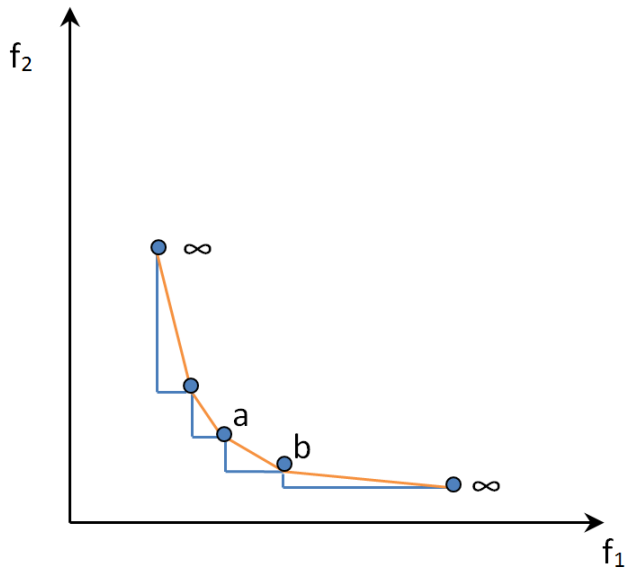


Figure 8: Crowding Distance Assignment. Crowding distance is assigned front wise. The individual solutions represent the maximum value of f_2 and f_1 are the boundary points of the front and will be set as infinity as there can't be a more diverse/ spread point that the boundary points. All individuals within the boundaries of the front will then be assigned a crowding distance based on the sum of Euclidean distances (the orange vectors) between neighboring individuals. It can be seen then that point **a** will have a lower crowding distance than point **b** as there is both visually a higher density of nearest neighbors as mathematically having shorter combined Euclidean distances to the nearest neighbors**.** A higher crowding distance is better

Upon assigning the crowing distance to all the individuals within the population, a new set will be created to reduce the combined population of size *2N* to that of size *N*. This new set to fill from the lowest front first to the last front until the size exceeds that of *N*. The last front that was used to fill the population will then sorted based on decreasing crowding distance. The individuals will incrementally be removed from the bottom until the population is exactly size *N*.

This population will then be preserved and will take the place of the older parent population. Even though this will replace the older population, since the previous child and parent population were combined and sorted, the best individuals will still be preserved; promoting elitism.

2.5.3 Binary Tournament Selection

Once the new parent population has been created, it will then be used for the selection and mating of new offspring. This selection process, as described in earlier sections, is known as binary tournament selection. Instead of comparing contestant's fitness value, as was described for single objective selection, the selection niching parameters are now the rank and crowding distance values. Two individuals are chosen at random and are allowed to compare ranking values. The individual with the lowest ranking value is selected as the champion and is permitted to mate. If, however, the two contestants share the same front, their individual crowding distances are compared. The one with the highest crowding distance is then chosen to preserve the spread of the population. Once a mating pool has been created, the generation of offspring commences.

2.5.4 Simulated Binary Crossover

Simulated Binary Crossover (SBX) has been shown to work very well with real coded GAs [22] and is an effective form of crossover for multimodal and multi-objective problems [23].

It has been argued that evolving neural networks may not be a simple task [24]. This is because, when dealing with multi-objective solutions, the population may contain individuals that produce the same behavior but the individuals may consist of entirely different genotypes. This means that two individuals could contain two genotypes corresponding to totally different areas of the decision space. Crossover could then destroy the working structure of the hidden layer and produce poorer fitness through offspring as can be seen in Figure 9.

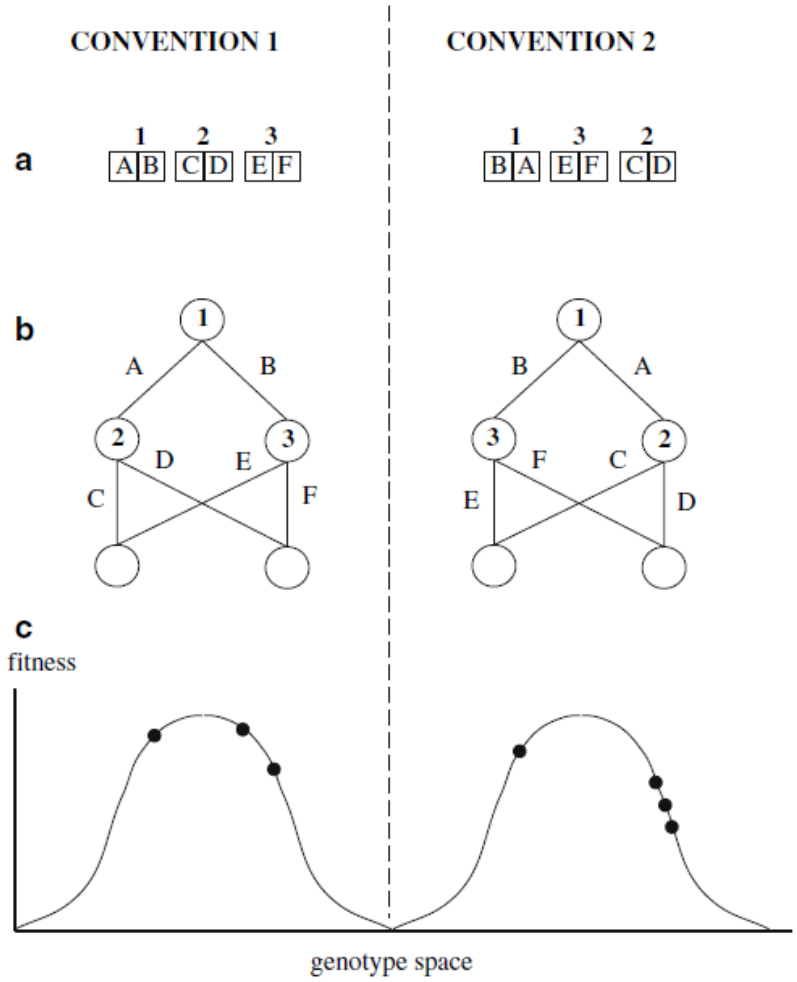


Figure 9 Competing conventions: [24]

It can also be argued, however, that with SBX combined with the inherent preservation of NSGA-II, the searching algorithm is able to overcome this issue by the means of turning the negative into a positive. The preserved elite can act as a vantage point and anchor for the new offspring to explore the surrounding local region. That way, individuals can search the region for the possibilities of a higher front to jump into. Therefore SBX is a good approach for the crossover of ANN weights.

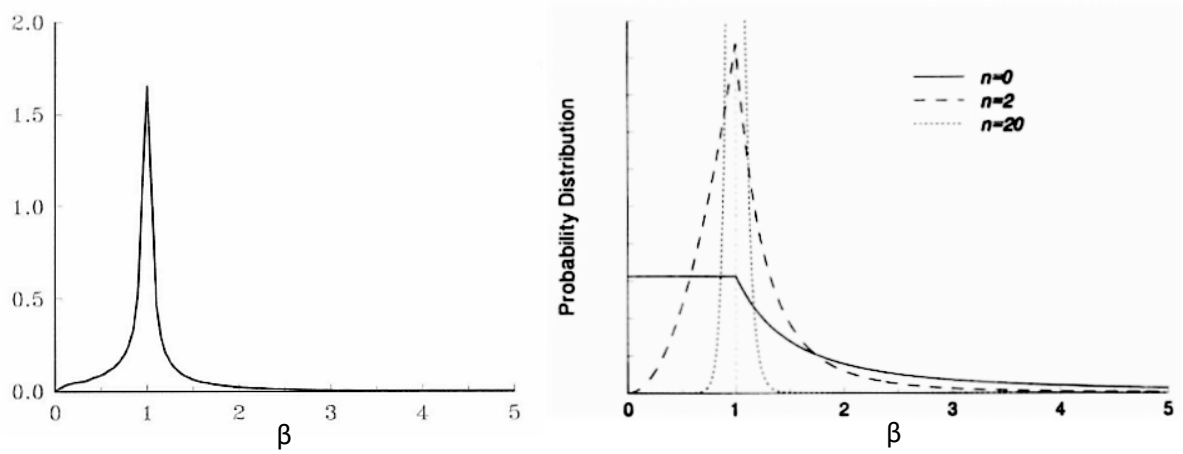


Figure 10 Probability Distributions of Binary GAs and Simulated Binary Real coded GAs: The left figure is the probability distribution of contracting and expanding crossovers for a pair of random binary strings of length 15. The right figure is the representation of the binary distribution as a polynomial distribution for real-coded GAs. For SBX, *n* controls the spread of the probability model. When n = 0, the distribution is close to a uniform distribution. When *n* = 2, the distribution is close to the original binary crossover distribution on the left. [25]

SBX was designed with respect to the one-point crossover in binary-coded GAs. To simulate the operation of a single-point binary crossover directly to real variables, a probability distribution function is used to act similar to the probability distribution in the Binary-coded scheme (See Figure 10) and will be explained in the next section.

When dealing with binary-coded GAs, the offspring may lie inside or outside the region bounded by the parents based on the crossover point or the specific strings of binary crossed over. To represent this differentiation between child points and adult points, the spread factor β, is to be used as the ratio of spread of child points to parent points:

$$\beta = \left|\frac{c_1 - c_2}{p_1 - p_2}\right|$$

If the child points lie outside the bounds of the parent points, then absolute difference in child points is greater than the parent points. This means the spread factor $\beta > 1$ and the crossover is considered to be expanding. If the child points lie within the bounds of the parents, the absolute difference of the children are less than the parents and the spread factor $\beta < 1$. This would then be

a contracting crossover. Lastly if the absolute difference of child and parent points are equal, then the crossover is stationary and the spread factor $\beta = 1$.

The idea then for binary GAs is that for each contracting case with a spread factor β**,** there is also an expanding case with spread factor $1/\beta$. The relationship of the probability distribution with β for contracting cases in binary GAs is:

$$C(\beta) = \frac{\sigma}{1-\beta}$$

where σ is some constant term [25]. (This portion of the distribution is in the range of $\{0 \leq \beta < 1\}$ and can be seen on the left figure of Figure 10). Since the overall probability for contracting cases are equal to the expanding cases, a relationship was found:

$$E(\beta) = \frac{1}{\beta^2} C(\frac{1}{\beta})$$

Therefore, given that the sum of both crossover probabilities is equal to overall probability, and the overall probability of either crossover types are equal, then each case must have an equal probability of 0.5. Knowing this, the contracting probability distribution can be substituted into the above equation to obtain the probability distribution for the expanding crossover:

$$E(\beta) = \frac{\sigma}{\beta(\beta-1)}$$

(This portion of the distribution is in the range of $\{\beta > 1\}$ and can be seen on the left figure of Figure 10) Thus a polynomial distribution was proposed for each real case for representing binary the distribution functions:

$$\begin{cases} c(\beta) = 0.5(n_c+1)\beta^{n_c}, & \beta \leq 1 : Contracting \\ c(\beta) = 0.5(n_c+1)\dfrac{1}{\beta^{n_c+2}}, & \beta > 1 : Expanding \end{cases}$$

The nonnegative real number, n_c, is the distribution index used to differentiate the polynomial model as can be seen in right hand side of Figure 10. For small values of this index, points far away from the parents would be chosen for the children. Oppositely, for larger values the children would become restricted by the distribution range and only points close to the parents will be chosen. Finding a proper distribution index value is then very important as it has a high impact on convergence speed.

For multi-variable problems in binary coded GAs, a crossover point is found to determine the split and swap of parents. This split can also be performed as well for real-coded GAs, however it creates a positional bias as the probability of transmission of genetic material will be highly dependent on the position within the chromosome [26]. Therefore, to simplify and create a uniform probability of crossing over any variable, the probability of a variable being crossed over is 0.5.

Once a variable has been chosen for crossover, a new random spread factor $\bar{\beta}$, must be calculated for the real SBX crossover operator. Since there is equal probability of both contracting and expanding cases a uniform random number between 0 and 1 will be selected (denoted as *u*). This value *u* will then be used to calculate $\bar{\beta}$ by equating the area under the curve equal to *u*. This was found to be:

$$\bar{\beta} = \begin{cases} 2u^{\left(\frac{1}{n_c+1}\right)} & : if\ u \leq 0.5 \\ \dfrac{1}{2(1-u)}^{\left(\frac{1}{n_c+1}\right)} & : otherwise \end{cases}$$

Now that a representation of the spread factor has been determined the offspring solution c_1 and c_2 can be calculated:

$$c_1 = 0.5\left[(p_1 + p_2) - \bar{\beta}|p_2 - p_1|\right] \qquad c_2 = 0.5\left[(p_1 + p_2) + \bar{\beta}|p_2 - p_1|\right]$$

The children that are created are then symmetric about the parents p_1 and p_2 to avoid bias towards any one parent as well as retaining the average value of both parents like in binary coded GAs.

2.5.5 Parameter Based Mutation

If the crossover operator is not applied after reproduction, a small amount of offspring with probability p_m will be allowed to mutate. The mutation used for real and continuous values is known as parameter based mutation [27]. Parameter based mutation uses a polynomial probability distribution to create an offspring solution around the vicinity of a parent solution. This approach is extremely similar to that of the SBX operator. To calculate the new mutated parameter a uniform random number between 0 and 1 is created. This is used to calculate the perturbation factor δ:

$$\delta = \begin{cases} [2u^{\left(\frac{1}{n_m+1}\right)}] - 1 & : if\ u < 0.5 \\ 1 - [2(1-u)]^{\left(\frac{1}{n_m+1}\right)} & : if\ u \geq 0.5 \end{cases}$$

This perturbation factor is specific to the mutation distribution index n_m, as large n_m create mutation closer to the parent and small n_m creates a new child far from the parent. This factor is then used to calculate the mutate value as:

$$c = p + \delta$$

All of the offspring that are created by the SBX operator or the parameter based mutation operator are added to a new child population. Once the population is of size *N*, the selection process is over and the offspring are allowed to be evaluated. Each generation allows for new and better controllers as the population moves towards the Pareto optimal front. This process reiterates until a stopping criterion has been met.

2.7 Mario AI as a Test bench

This section will explain the importance of using platform games like Mario AI to test intelligent controllers in a highly complex environment.

Mario AI is a tool developed by Juilian Togelius and Sergey Karakovkiy to be used as a new way to test complex learning algorithms and policies. This tool has since been used in competition at the Mario AI Championship for multiple NE methods [28]. A content creation track was implemented that allowed a group to develop a procedural level generator that replicates styles of a human level designers [29]. A study by Ortega et al used the Mario AI framework to let a Neuro-evolved controller learn and eventually mimic the behavior of human players [30]. A game play track was also created to promote contestants to create intelligent agents to solve the difficult task of learning how to play multiple levels of Mario. One interesting study implemented a game play agent that used RL to develop a Neuro-evolved controller using a single combined weighted fitness [31].

Video games have a smooth learning curve in the form of level difficulty, thus making them useful for continual learning. Super Mario AI is a continuous, high dimensional state-space, partially observable environment. It contains many different level types with different enemies and thus has high dimensional observations. But what is most useful is the goal of the game requires execution of multiple different behaviors to complete the game [32].

Figure 11 MarioAI: The MariAI test bench provides an overlay that displays the current level information, Mario score, and the controller's name and its action at each time step.

Super Mario Bros' gameplay consists of a game character (Mario) starting on the far left of a level where he must traverse obstacles to make it to the end of the level at the far right. This is a simplification of the problem as the character can move through this 2-D world by jumping over pitfalls, collecting coins, avoiding a multitude of enemies like turtles and angry bullets, shooting fire from his hands, eating power-ups, and the list goes on. Getting to the end of the level is the main goal but the overall score is a combination of sub goals: collecting coins, collecting power ups, maximizing distance in level, killing enemies, reducing the amount of times Mario gets hit, and maximizing time remaining in a level.

Mario has three stages of power-ups in which he starts the game at the highest stage. In the highest stage, Mario has the power to shoot fireballs which can kill almost any enemy within the game. This power is linked to the run button, so the agent must learn to differentiate between shooting and running. If Mario gets hit while in this "Fire Flower" stage, he is reduced to regular

Mario. In this form he still retains his size, but he no longer has the ability to shoot. This means that the agent has to differentiate between stages so that it knows it can no longer kill enemies by shooting/running at them and must find a new way to destroy or avoid them. The last stage, and Mario's last chance, is small Mario. In this form Mario has lost his height and now looks like a baby. This is both a negative and a positive, as even though Mario has only one hit left his hitbox has been reduced to fit his model size. Therefore he can pass through more difficult areas with deft, cunning, and luck to bypass obstacles.

The enemies that are injected into a level are generated with special behaviors, properties, and types based on the level type and difficulty. An enemy type that performed one way at a simpler level, could gain wings and jump at more difficult levels. As the difficulty of the game increases, so do the amount and variety of obstacles within the level that Mario must overcome.

The generation itself can become noisy with the level seeding property of the generator. A single level at a specific difficulty can become completely different when generated on different seed values. Thus an agent that performs well on one particular difficulty and level type must be able to generalize or it won't be able to handle the change. By randomly selecting seeds for each level, the agent would train on a different generated model than the one before. This makes it near impossible to create a deterministic model of the gameplay space, and is why NNs are typically used as a surrogate model to approximate the data, as opposed to trying to learn and adapt through every possible level. When a human plays Mario, they have no trouble understanding this overwhelming amount of information and use it to learn different complex actions for winning, but to a learning agent this becomes an extremely complex problem to solve as it requires a switching of learned behaviors quickly and efficiently. This means then, that the Mario AI domain is fractured.

A fractured domain is one where tasks change wildly and infrequently when entering different state spaces. For many standard problems in RL and control, state space is smooth and has low variety in action over time. But for video games like “Super Mario Bros”, different tasks within the game could switch and vary after every time step.

One example of a game with fractured task space is “Keepaway Soccer” [33]. In this game, Kohl showed that the strategies necessary to win within his Markov decision process (MDP) modeled task space were fractured and difficult for a Neuro-evolved controller to learn without complexification of NN architecture and learning methods. This proved especially difficult for NNs that evolved network topologies.

Another example of a fractured space is that of Mrs. Pacman, where the tasks for dealing with threatening ghosts and ghosts that can be eaten made it difficult to split tasks [34]. Schrum was able to overcome the behavior split by using multi-objective optimization to drive state/action partially observable Markov decision process (POMDP) selections.

Therefore, an interesting method for solving the fractured Mario AI problem is to use multi-objective optimization to develop multi-modal behavior to overcome the fractured task selection. Since the state space for Mario AI is so large and the resulting action space (where the controller can output: down, left, right, jump, and shoot at each time step) yields $2^5 = 32$ total possible actions, it would be too ineffective to create a MDP modeled controller. Therefore a direct action selection controller is used instead. The quantitative measure of fracture is hard to define in a non-Markovian Domain where state/action and reward are predefined. Therefore any mention of fracture within MarioAI is through qualitative measurements.

It can then be seen that Mario AI is a powerful, robust and diverse benchmarking tool. For the case of this study, the gameplay track will be used to develop an intelligent, multimodal, game solving agent.

CHAPTER III

METHODOLOGY

3.1 Encoding and NN representation

For real world optimization problems there are many ways that a GA can be encoded. Varying encoding schemes can affect significantly the convergence time for solutions as well as the quality of solutions. A good form of encoding with evolved neural networks is direct encoding, as shown through extensive analysis of genetic operators and real-coded chromosomes [35]. Braun and Weisbrod also state that the use of indirect encoding requires a "detailed understanding of both the neural network and genetic algorithms inlayed mechanisms. [36]" This means indirect encoding relies on the developer to create a quality mapping schema from the GAs genotype to a phenotype that can be efficient in evolving the neural network. Therefore in this study, direct encoding was chosen to represent the genotype.

The neural weights can be conceived of a finite-dimensional parameter space directly within the genotype. Usually the formulations of this parameter space for GAs are through binary encoding. Binary encoding is fairly straightforward but requires a significant amount of bits to represent any real world decimal number to a useful degree of accuracy. Combine that with the large parameter space of a neural network, and the storage complexity of the system grows rather large. Instead of binary, real floating point numbers can be used to encode the system.

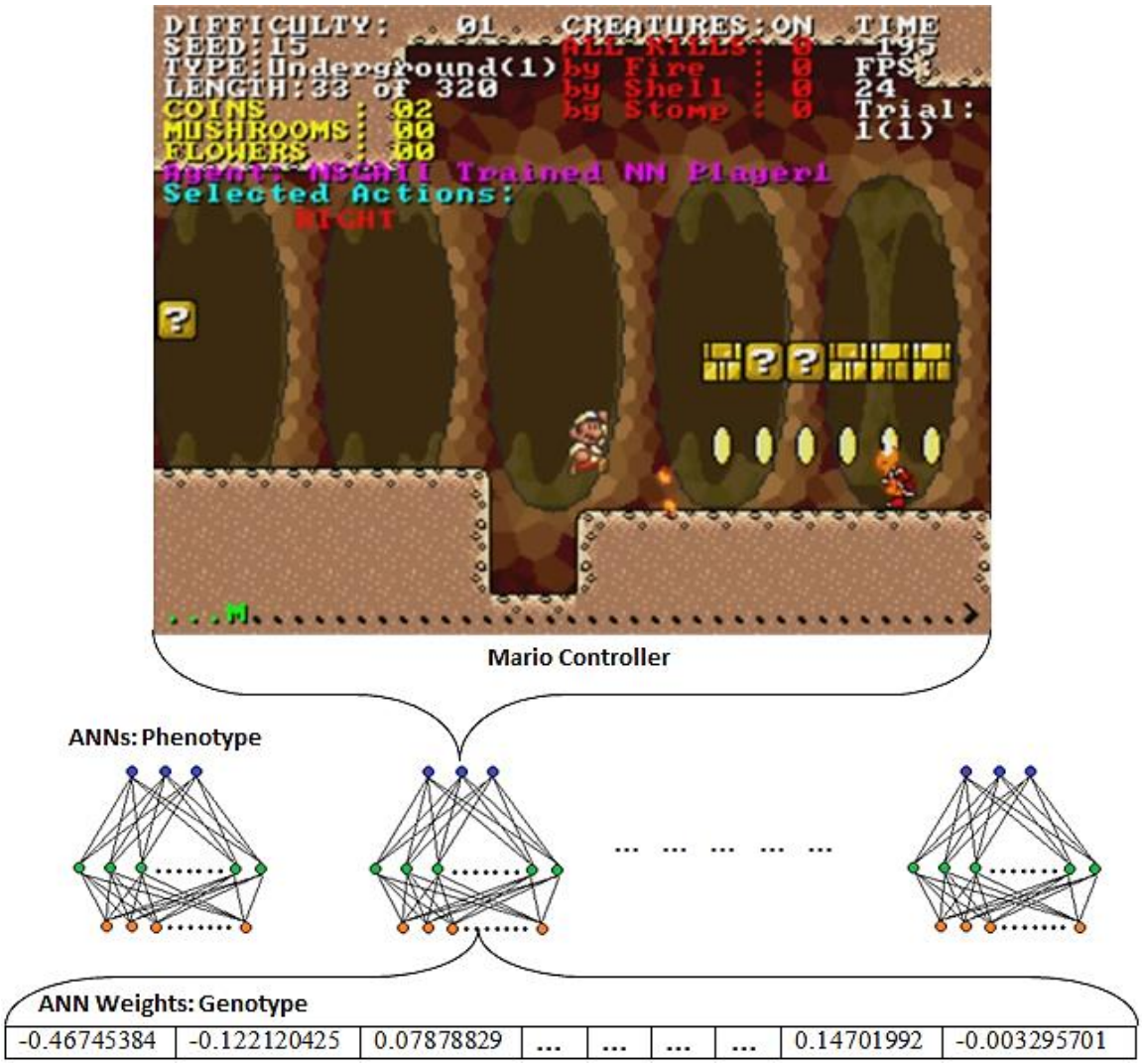


Figure 12 One-to-One Mapping of real encoded Genotype to Neural Network Phenotypes: The One-to-One mapping of real numbers as parameters to the ANNs is simplistic and effective.

Work with real encoding in GAs has been found very effective in continuous domains [37]. Since all problems of weight manipulation in this study are defined in the real space, the use of real-coded GAs with the SBX searching operator is more suitable than Binary GAs. Using real numbers allows for simple one-to-one mapping from genotype to the phenotype as can be seen in Figure 12. Since the choice of encoding drastically affects the search space of solutions, it is a pivotal aspect of the design of an NE system

3.2 Input representation

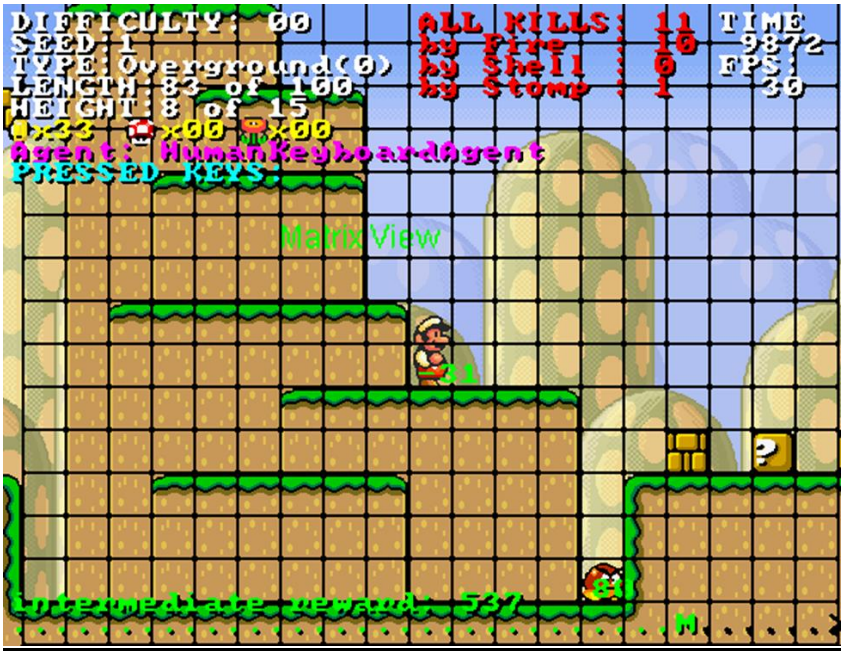


Figure 13 Mario's Visible Environment: Mario's visible 22x22 environment with Mario residing at point 11:11. Bitmap key data is held within the grid space for each rendered object. For example, the enemy at grid space 15:16 is returning the value: 80.

Mario's visual representation of the world is through a 22x22 grid of squares with Mario in the center at block 11:11. Each square contains the raw data bitmap keys of the level environment at every time step. This influx of data is supposed to provide Mario with the same amount of information as that of a human observer playing the game. This means, however, that the total observable space that Mario can account for is 484 blocks of input data. This high-dimensional space is extremely difficult to analyze and is known as the curse of dimensionality. For creating a controller for MarioAI, training on a direct input space of 484 shows to be impossible. This is true even if the raw data types are normalized between [0-1] for easier search-ability and the controller is allowed to play for more than 300 generations (an average total run of: 300,000 levels played).

There are many approaches for reducing the input space. One possibility is to just reduce the visual input around Mario. Attempts of this method using a reduced visual size of 9, 25, and 46 blocks of visibility resulted in a Mario controller that could handle lower levels with minimal input variety [30]. As the level difficulty increased the addition of more enemies, enemy types and difficult obstacles like gaps were added. The controller could not efficiently differentiate and account for the new variables with such a limited field of view. As the field of view was increased, so did the learning curve as the input space dimension became large. The best input state size for that experiment was found to be between 14 -21 total inputs before any learning ability started to be hindered.

Another method is through the use of angle and positional inputs. These are effective inputs for 3D environments like in the video game Quake [38], or the car racing simulator developed by Togelius and Lucas [39]. These state spaces are represented in a spherical plane, yet MarioAI is a 2D grid with x and y block coordinates. This is an important distinction as the type of input representation can both reduce the fracture problem and bias the GA's search toward finding useful multimodal behavior. To put it plainly, solving a problem in a grid space becomes much more difficult if the state space is represented as polar coordinates as opposed to Cartesian.

Other forms of input representation use a user defined pathfinding state/action representation. This usually is of the form of some hierarchical determined algorithm to provide both direction and policy. It is arguable, then, that the driving mechanism for multimodal behavior is predetermined. This type of bias towards learned behavior is not applicable to this thesis. Therefore, a mixed input representation of feature selection and straight line sensing is proposed.

Within the MarioAI domain, there are three main sections of the level environment that Mario must account for: Obstacles, Enemies, and Collectables. Information about these items must be available to Mario in a way that is descriptive without being overly large and complex. A form of

feature abstraction is then used to describe the world for each of these sections. It is also important to mention that each of these sections make up the overall input visualization of the Mario agent. However, to show each individual environmental abstraction in detail, each section is split to better inform the reader on how each environment feature is determined and selected.

The main necessity for Mario to win is to have a basic understanding of passable terrain and hindering obstacles. For this, seven types of inputs are proposed and can be seen in Table 1.

Input Name	Details
Obstacle Above Distance	Returns the relative distance of any obstacle three blocks above Mario. Returns 1 otherwise.
Obstacle Below Distance	Returns the relative distance of any obstacle three blocks below Mario. Returns 1 otherwise.
Obstacle in Front Distance	Returns the relative distance of the closest obstacle five blocks ahead in line of Mario. Returns 1 otherwise.
Obstacle in Front Size	Returns the size in blocks of any obstacles in front of Mario. Returns 0 otherwise.
Gap Distance in Front	Returns the relative distance of the closest gap to Mario. Returns 1 otherwise.
Find Edge of Gap	Returns the relative distance of any edge in front of Mario. Returns 1 otherwise.

Table 1 Obstacle Input Sensing:

These inputs are designed to pick out specific environment devices that cause Mario issues in completing the game. Obstacle detection above and below Mario is used to provide both floor and sky information to Mario. This is by providing feedback between three grid values above and three grid values below Mario. A relative distance value scaled to [0, 1] is returned if certain terrain patters are found. This will give Mario an understanding of if he is on a flat surface or in the air, as well as visuals for terrain that can be jumped up into like hills. The line sensing input in front of Mario returns the relative distance of the closest impassible obstacle five grid spaces in front of him.

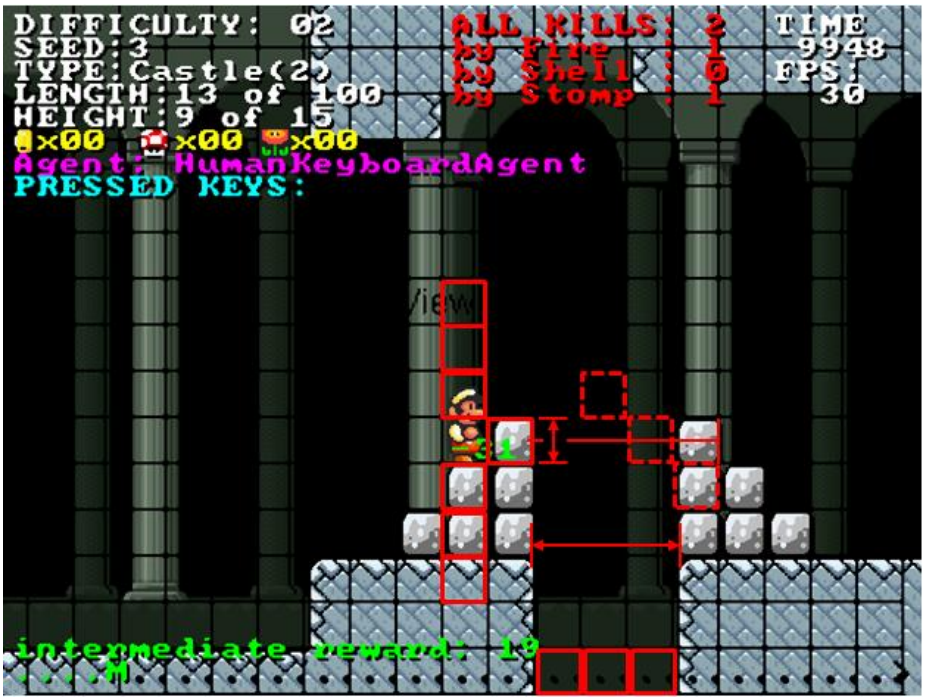


Figure 14 Obstacle input information: Input abstraction selection determines impassable obstacles relative position by looking three blocks above Mario, three blocks below Mario, and line sensing obstacles directly in front of Mario. For handling gaps, the bottom of the screen is monitored to search for a lack of terrain to the right of Mario. To handle the stairs problem as in this figure, an edge detecting sensor (depicted as the red dashed squares) is used to look ahead and determined if there is any place safe to land. Gap and objects in front of Mario also have a height value returned to tell Mario how far to jump.

The most difficult danger that can befall Mario is the gap. The gap can quickly end a good run as it instantly kills Mario and ends the level regardless of Mario's current health. Not only is this difficult to learn to overcome, it is also difficult for the agent to learn that it needs to be overcome. Some Mario agents might grow to fear the gap and choose never to jump but just stand at the edge until the time runs out. Therefore, not only does gap distance need to be addressed, but also the size of the gap. This way there is some information about when and how far Mario must stay airborne to bypass the pitfall. Lastly, the input for specifically dealing with edge problems was created for the problem of stairs as shown in Figure 14. These white blocks create a dilemma for Mario that he can't usually overcome with normal obstacle detection alone.

Typically Mario will see an obstacle like this and jump right over it only to lead to another simple block obstacle.

He will then crawl all the way to the top, realize there is a gap, and attempt to clear it. Without understanding where the landing edge is, Mario will undershoot the jump and hit the opposite descending staircase on the other side. Therefore, a primitive "landing" edge detecting input is necessary to handle the complicated stair problem.

The next sets of inputs are dedicated to handling the different enemies within the environment and can be seen in Table 2 below.

Input Name	Details
First Enemy Position	Returns the x and y coordinates of the closest enemy to Mario. Returns 1 otherwise.
Second Enemy Position	Returns the x and y coordinates of the second closest enemy to Mario. Returns 1 otherwise.
First Spike Enemy Position	Returns the x and y coordinates of the first closest spiked enemy to Mario. Returns 1 otherwise.
Second Spike Enemy Position	Returns the x and y coordinates of the second closest spiked enemy to Mario. Returns 1 otherwise

Table 2 Enemy Input Sensing

These inputs are designed to both reduce the total input space and give concise positional direction information about the on screen enemies. One issue Mario faces is the possibility of attack from multiple directions. Instead of using relative position detection like that of obstacle detection, a type of feature selection is used. The proposed method for finding this information within the state space is by searching through the visible environment for the bitmap key data specific to enemies. There are many enemy types within the MarioAI environment but to reduce the state space complexity, all enemies were split into two categories: those that can be killed by jumping on top of them, and those that cannot.

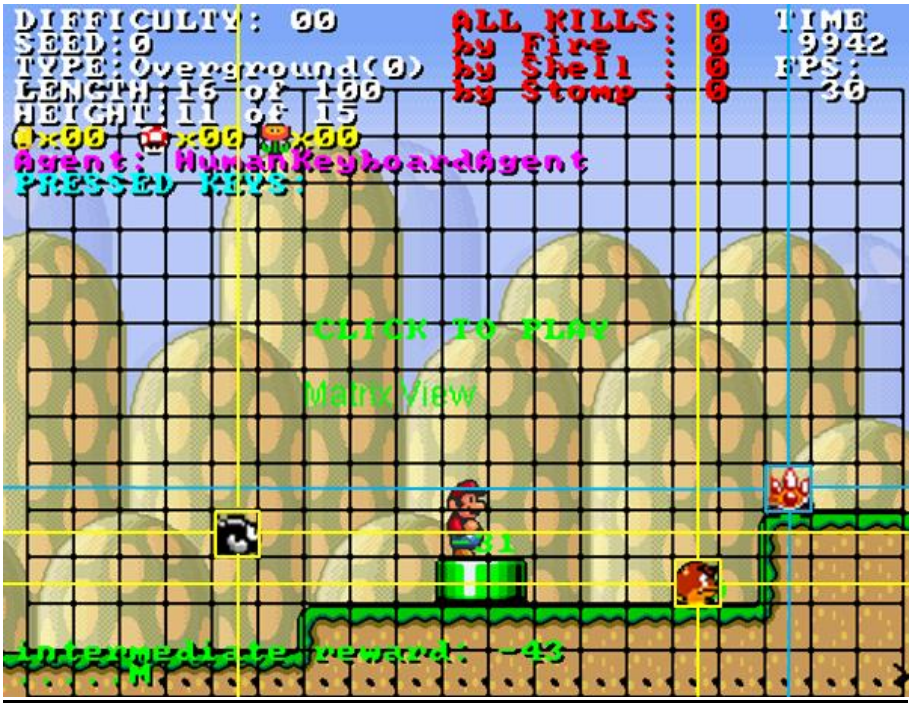


Figure 15 Enemy input information: Input feature extraction for enemies search the visible grid and returns the enemy's x and y coordinates. Enemies that Mario can jump on are represented by the yellow square (the Bullet and the Goomba) and enemies that Mario cannot jump on are represented by the blue square (the Spiny Shell)

This distinction will afford Mario enough understanding of the environment to overcome most issues without imposing too much bias from the designer. There are a total of four inputs for each category of enemy corresponding to the closest enemy's position and the second closest enemy's position. This representation gives Mario both direct positional awareness, in the form of Cartesian coordinates, and indirect relative position, in in the form of closest and second closest enemy. This method of feature selection reduces the need for omnidirectional or quadrant based enemy detection and can be observed in Figure 15.

The last section necessary of attention is the collectables. These assorted items throughout the game can both aide in the completion of the level through power-ups, and add to the overall high-score through coins. For the original MarioAI competition, the coins were deemed not necessary as a form of measurement and relied only on level completion as the determining metric.

For this thesis however, the coin bitmap information was added back in. The inputs for handling feature space extraction for collectibles are below in

Input Name	Details
First Collectable Position	Returns the x and y coordinates of the closest collectable to Mario. Returns 1 otherwise.
Second Collectable Position	Returns the x and y coordinates of the closest collectable to Mario. Returns 1 otherwise.
Closest Block Position	Returns the relative distance of the closest breakable or hidden block to Mario. Returns 1 otherwise.

Table 3 Collectible Input Sensing

The collectable sensing feature extraction uses the same method that was described earlier for that of the enemy detection. The collectable categories are split into two types: pick-ups and blocks. The pick-up category accounts for coins and power-ups whereas the block category is specific to the breakable and hidden blocks that Mario can open to see if any more collectibles are inside. The pick-up category has four inputs dedicated to the (**x, y**) pair of the first and second closest coin or power-up, allocating both direct and indirect positional awareness to Mario. The block input requires only one (**x, y**) pair as it is not as detrimental to Mario's overall success. This selection visualization can be seen in Figure 16 on the following page.

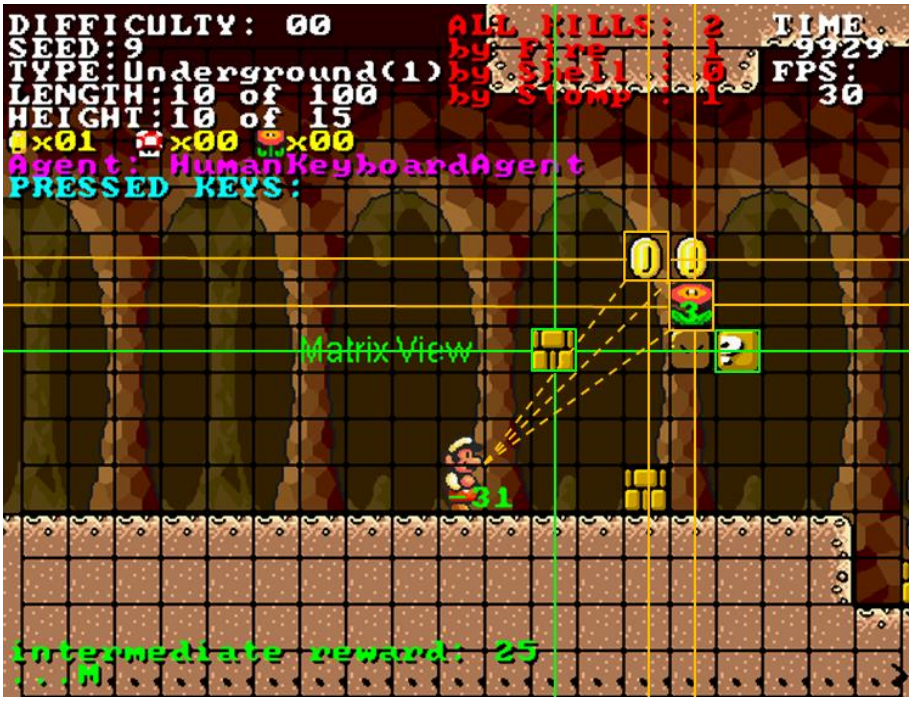


Figure 16: Collectible Input Sensing: Coin and power-up feature extraction is represented as the orange box (Coin and Flower). The dashed line shows the Euclidean distance to closest and second closest collectible. The green box (Brick Block) returns only the closest brick block or hidden block to Mario.

The three categories described will provide the visual feedback of the state space to Mario. By using methods of abstraction and feature selection, the input space has been reduced to 20 inputs from a total of 484. Four more third-person inputs were created to overcome certain limitations that were not provided by the environment.

Input Name	Details
Can Mario Jump	Returns a 1 if Mario is on the ground and the jump button is not pressed. 0 Otherwise
Mario Size	Returns the current health of Mario
Coins left in the Level	Returns the remaining percentage of coins left in the level
Input Bias	1

Table 4 Third-Person Inputs: "Can Mario Jump" is dedicated to a particular issue where Mario cannot determine without some feedback the need to release and re-press the jump button to perform the task. Mario size gives Mario updates on how many times Mario has been hit. The Coins Left input is created to add slight bias towards collecting more coins. And a dummy input bias of 1 was added for the activation threshold.

With the current input setup determined, the resulting output policy is that of a direct action policy. Mario can perform any combination actions: up, down, left, right, run/shoot, and jump. By removing the up command as it proves to be unnecessary this provides a total of 5 output commands. These commands can either be 0 or 1 and the state is determined by a threshold operator in the output layer. The logistic sigmoid activation function will frame the output value on the range of 0-1; therefore a command activation threshold is set uniformly at 0.5. Values above or equal to 0.5 result in that activation being sent to the Mario controller.

3.3 Fitness Criteria Selection

Previous trials evolving MarioAI agents were modeled on the work of Lars and Thomas [29]. A single weighted objective GA was used to train intelligent controllers to complete multiple levels. This controller was able to learn basic behaviors necessary to win but could not learn any complex behaviors, like differentiation between when to be greedy and collect coins and when to be safe. The evolved agent learned that if it ran as fast as it could and dodge the main obstacles like gaps and walls, it could win and win on at least two different levels. This agent suffered though from many pitfalls that single objective solutions face in a multi-objective optimization problem. Lack of understanding of the driving force behind specific behavior led to constant tuning and approximations of the fitness function's weighting parameters. This problem is only magnified by the fact that a human designer will try and impose a higher reward for objectives they believe to be the best. When in actuality, the Mario AI problem does not have a quantitative "best action" per state scheme. This led to agents with decent performance for one task, and detrimental performance in another. This style of fitness selection also resulted in poor convergence as the highest scoring agent using this method from Lars and Thomas took over 24 hours to calculate and over 1,050,000 different levels even without any level type variation.

This thesis proposes a multi-objective approach to circumvent the issues of fitness parameter tuning and lack of multimodal behavior. To reduce the complexity of the problem, the controllers will only be trained on two objectives. Selecting these two objectives however is not a trivial task.

As it has been said before, the main goal for Mario is to reach the end of the level. Therefore it makes sense to have one objective be to maximize the distance that Mario travels. Lack of this criterion as one of the objectives results in a controller that will never learn to explore and complete a level. So with the main form of direction as one goal, the second should be selected to both supplement a high game score as well as conflict with the first objective to develop multimodal behaviors.

One idea is to maximize Mario's health, ideally creating agents that both clear the level and do so while minimizing the amount of times Mario gets hit. While this is good form of conflict, it also results in the creation of agents that can be too scared to move (Figure 17 left). Mario will learn that the best way to maximize his health is to stay at the beginning and hide. This behavior will be continually rewarded and spread throughout the gene pool to create a population of scared Mario's.

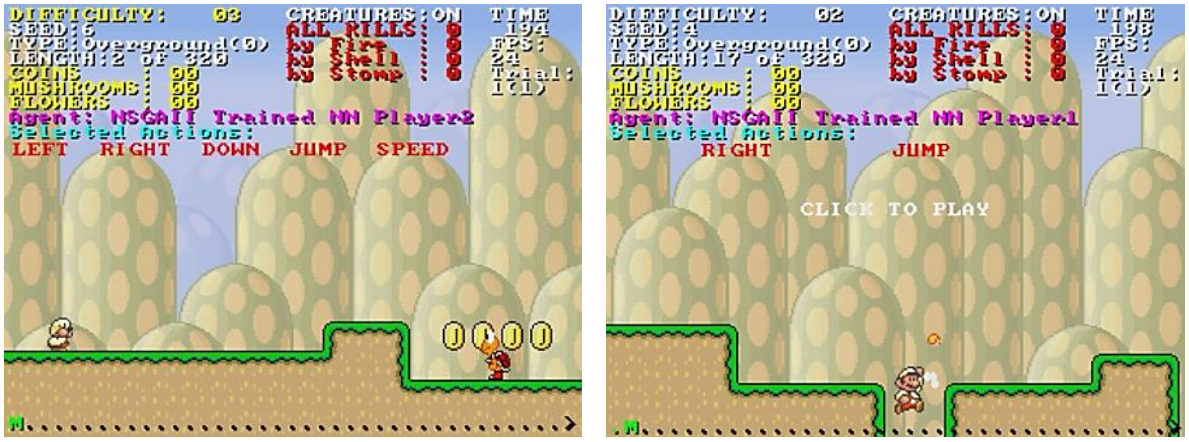


Figure 17 Fitness Criteria Issues: The agent on the left learned on an additional objective to maximize Mario's health resulting in "Scaredy Mario." The agent on the right was trained to minimize speed. A certain amount of agents determined that the fastest way to beat a level is to end the game by jumping headlong into a gap.

So maximizing health might be better suited as an addition to the niching selection scheme that an objective unto itself.

Another constructive behavior a human player might exhibit to get the highest game score is to beat the level as fast as possible. This is a good example of human developers anthropomorphizing their trained agents. A complicated task such as “beating the game fast” is an abstract idea that is easy to understand for a human, but could mean something completely different to an intelligent agent. At the risk of morbidity, Mario agents trained to beat the game as fast as possible can learn to become suicidal by finding the closest gap and jumping right in. This immediately stops the simulation and minimizing the time Mario was in the level (Figure 17 right) Obviously additional high level design could be imposed to only select agents that went a certain distance, or a certain amount of time spent on a level, but this requires time spent on determining how far or how long Mario should go. If speed was of upmost importance, the best way to minimize time on a level could be done indirectly. If the levels used training set were given half the time of the testing set, Mario would have no choice but to go as fast as possible.

For this thesis the proposed second objective will be that of coin collection. The task of collecting coins provides learned behavior that is beneficial to the overall score without creating unforeseen learning issues. It also creates a visible distinction between learned behaviors within Mario as he determines when to attempt to collect coins and when not to.

3.4 Training to Generalize

It has been found in previous work that a simple neuro-evolved controller can easily learn to beat individual levels of MarioAI. When trained on a single training level the population will eventually converge to an agent that has determined the best found way of completing that level. This means however, that the agent had become deterministic to that one level and lacks any real general knowledge of the state space. Many reinforcement learning problems have issues of generalization due to the lack of available training data. This is not the case for MarioAI, as a new version of any level and type can be generated by changing the level's "seed" value allowing for an extensive catalogue of training data.

In addition to game level's seed is the level's type. MarioAI has the capability of generating three different world types: Overground, Underground, and Castle. These types have a specific generative algorithm that allows for differing terrain and setup. The Overground level has the possibility of creating hills that can contain either treasures or enemies at the peaks. The Underground level has long ditches that can be full of coins but might be covered in blocks, so once Mario falls in he either has to run to the end of the ditch or try and break out.

Figure 18 Level Types: The three levels used in this study represent the control (top left) the Bonus level (top right) and the Marathon level. These separate levels are used to specify and visualize the distinct multi-modal qualities of the learned Mario agents.

Lastly the Castle level can be the most difficult, as it has low ceilings with many stair problems show previously in Figure 14 and in the bottom of Figure 18.

These three level types (Figure 18) have very distinct setups and are used to promote generalization of certain aspects of the game. To further promote these aspects, the levels in this thesis have been altered farther. The Overground level is the control level with no modifications to the level score or setup. It contains at most 100 coins and requires passing 320 blocks worth of distance to the right to find the exit. The Underground level, which will be known as the “Bonus” level, has been modified to contain a maximum of 150 coins as well as the coins counting for double the overall score. It is the same length as the Overground level but has a higher probability

of generating coins. This level type is meant to both promote the learning behavior of coin collection as well as visualization of that task. The Castle level, which will be referred to as the "Marathon" level, has been extended from a maximum width of 320 blocks to 640 blocks. It also has the maximum coin reduced to 50 thereby doubling the length and halving the coin value of that of the Overground level. The new Marathon level is used to establish nimble and consistent movement behavior within trained Mario agents as well as visualization of said behaviors.

The last approach of generalization through training is actually the order of levels that are fed to the controller to play on. If the controllers are allowed to play on one level at a time, generalization is lost after each level increment. A controller that trains up to level 3 will have difficulty not only completing that level, but even the simpler previously learned levels. Therefore to better invoke generalization within the learning agents, each individual must be allowed to train on multiple level difficulties per population evaluation. This has some tradeoffs as the designer must determine how many levels that should be to get a complete representation of the state space without becoming too complex and computationally inefficient. For this study it was found that training from level 0 to level 3 to be the best range.

3.5 Reducing Variance in Noisy Environment

With the addition of these different level types and the generation variance induced by different level seeds, the training data has substantial noise. Handling of this noise is necessary as without noise reduction in the training set or the evaluation, poor scoring individuals would have some probability of getting a better score just based on luck. To overcome this issue, simple but effect measures were used to reduce noise in the objective evaluation stage.

Static resampling is a common and valid method of noise reduction. Jin and Branke found that by re-evaluating each individual a fixed number of times, a new averaged objective value can be calculated [40]. The reduction of noise in the averaged objectives is then proportional to the standard deviation of the objective by a factor of $\sqrt{n}$, where n is the fixed sampling rate. This comes at a price though, as computational efficiency is increased by a factor of the sample size.

Reduction of noise can also be handled through the increase of population size. The assumption is that for large populations there is a higher possibility of similar solutions. Therefor any noise within the search space is compensated for as the frequency of random outliers due to noise is effectively negated from a larger population moving into favorable search space regions with higher frequency.

Lastly a normalization of objectives both contains noise variation to a bound as well as provides a standard to which agents can be measured equally.

For this study, the best trade-off for minimizing the effect of evaluation noise while also maximizing computation efficiency was to be around *n* = [15-20], *population size* = 50, and an averaged objective evaluation normalized to the percentage of completion of the relative objectives. That is, the new objective calculation is:

$$f(coins) = \frac{\sum_{i=1}^{n}(\frac{coinsGained}{TotalCoins})}{n} * 100$$

$$f(distance) = \frac{\sum_{i=1}^{n}(\frac{distancePassed}{TotalDistance})}{n} * 100$$

Where n is the fixed resampling rate and the fitness is multiplied to scale from [0-100] %.

3.6 Evolving the Network

This section is to explain how the culmination of ideas explained so far translates to a working multimodal neuro-evolved agent. The flowchart of this process can be seen on the following page in Figure 19.

The algorithm begins by initializing the total number of controllers, the stopping criteria, the fixed re-sampling rate, and the neural architecture (*PopulationSize*, *TotalGenerations*, *SamplingRate*, and *LayerSize[]* respectively. The Neural Network architecture is a fully connected feedforward network where the number of layers and nodes in each layers is specified as an array {*25, 10, 5*}. (Three total layers: 25 inputs neurons, 10 hidden neurons, and 5 outputs neurons). The activation function used for each layer is the log-sigmoidal function. Next the populations of neural nets are initialized with a uniform distribution of random weights which is standard of GAs, as opposed to the standard Gaussian distribution of zero mean and standard deviation $1/\sqrt{n_{init}}$ weight initialization for gradient descent methods.

After the ANNs have been created, they are mapped to an instance of NSGA-II where the phenotypes are the individual ANNs and the genotypes are the weights parameters of each individual. The crossover and mutation probability as well as the distribution index for the reproduction operators are set within this class.

After initialization, the controllers are allowed to train until the stopping criterion of *TotalGenerations* is reached. Then for every individual in the population, allow them to evaluate on a randomly seeded level of random level type and set difficulty. This individual will train until a total *SamplingRate* is achieved and an averaged set of objective scores are calculated and saved. Once every individual has been evaluated, the ranking, sorting, and repopulation are performed. This process continues until the stopping criterion is reached and a set of multi-modal Pareto-optimal Mario controllers are created.

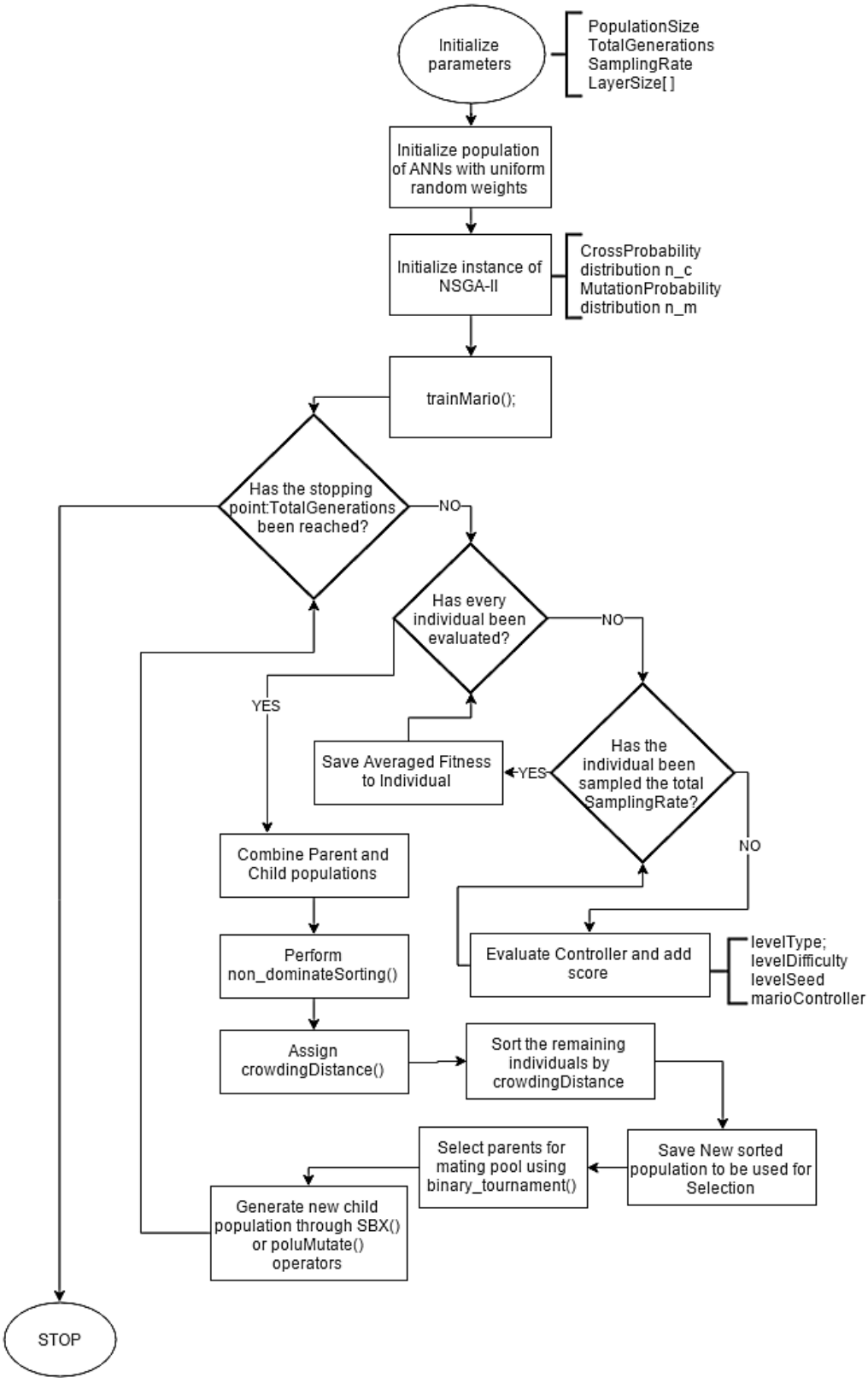


Figure 19 Evolving Mario-Flowchart

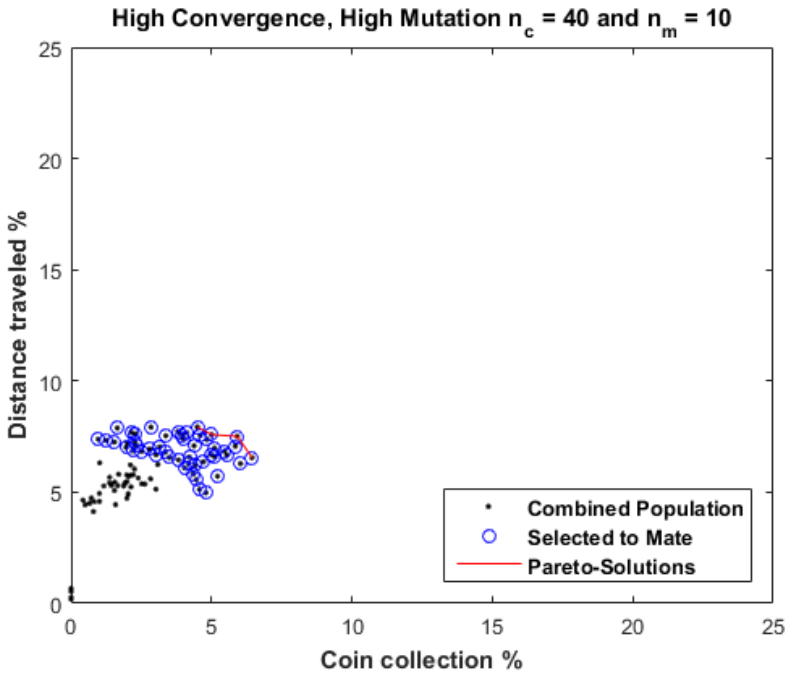


Figure 20 Non-optimal parameter selection: This graph displays the lack of a population set with too high a distribution index for crossover and mutation. Each individual ANN is represented within the objective space as a dot. It can be seen that training on this setup results in lack of traversal of the objective space. For aid in visualization, the x and y axis has been confined to 25 out of 100%.

3.7 Parameter tuning and dynamic mutation rate

The process of evolving multi-modal Mario controllers is efficient, but to create the "best" Pareto-optimal solutions requires fine tuning to optimize the exploration and exploitation of the search space.

The tunable parameters specific to exploitation and exploration is that of the crossover and mutation operations. To traverse the search space and discover beneficial regions requires a healthy mix of convergence and diversity. Too much crossover could result in being stuck in a local optima, and too much mutation would result in overall poor convergence to any solution and can be seen in Figure 20.

To reduce the amount of total parameters needing to be tuned, an ad hoc method of dynamic mutation rate was used to regulate the probability of mutation and the mutation distribution index like as described in [22]. The idea behind this method was to promote exploration early and exploitation later. Therefore the new values for the mutation distribution index n_m and the probability of mutation p_m are as follows:

$$n_m(t) = n_m + t$$

$$p_m(t) = \frac{1}{w} + \frac{t}{t_{max}}(1 - \frac{1}{w})$$

Where *w*, is the total number of weight variables in the system, and *t* and t_{max} are the current generation and total generations respectively.

It can then be seen that at the first generation the average probability of mutation is that of only one weight variable ($p_m(1) = \frac{1}{w}$) and the last generation would be ($p_m(t_{max}) = 1$). The distribution index n_m increases linearly as generations increase. It can then be observed that early generations would result in mutation of only a few variables but each variable is mutated with high perturbance. As the generations approach the maximum generation value, more weight variables will be selected to mutate but each mutation will be of less perturbance. This then results in the promotion of high exploration early on and then high exploitation as generations approach maximum generations.

Additional effects of parameter tuning will be discussed further within the “Findings” chapter.

3.7 Decision Maker NN Ensemble

When working with multi-objective optimization problems, a solutions set that forms on the Pareto-front is found. If the size of the solution is greater than one, some decision should be made as to which individual is to be selected. This deciding process is usually through some third-party that understands extensively the system that the objectives were evolved to. This method is not applicable within this problem as it is costly and would counteract the value of creating multiple Mario agents with differing behaviors. Therefore a *Decision Maker* (DM) neural network ensemble is proposed.

The neural networks within this study so far have been trained as a function approximator of the MarioAI state space problem. As well as function approximators, neural networks are also efficient pattern classifiers. Therefore it is interesting to determine if strong pattern recognition could be learned through the same neuro-evolutionary method as that of the Mario Agents.

The DM will take three inputs and result in an output selection of one Mario game playing agent out of the total Pareto solutions within the Pareto front. These three inputs were chosen as they are the only pertinent visible variables of the environment that Mario can understand without delving into the level code to reveal the environment a priori. The three inputs can be seen in the table below:

Input Name	Details
Level Difficulty	Returns the current evaluated level's difficulty
Level Coins	Returns the total coins in the level based on the level type.
Level End	Returns the total distance of the level based on the level type.

Figure 21 Decision Maker Neural Network Inputs

This neural network will then train using the same method of static sampling and evaluation using the games actual scoring criteria as a single objective fitness function. The neural network architecture of the decision maker is a fully connected feed forward network where the hidden layer uses a hyperbolic tangent activation function (TanH) for smoother learning, and the output layer uses a logistic sigmoid (LogSig) activation function to squash the outputs to a range of 0-1. The outputs of the DM are a preference neuron of each trained intelligent agent within the Pareto-optimal set. The selection of which trained agent is to play a level is through a winner takes all policy as the output with the highest activation value represents the chosen Pareto-optimal agent. The architecture representation can be seen in Figure 22

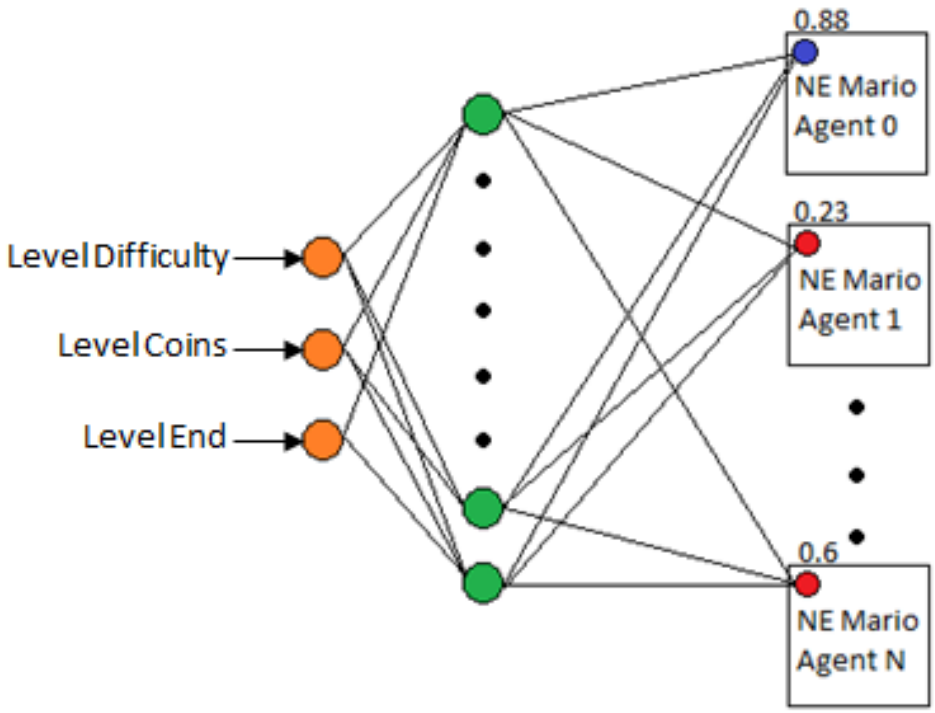


Figure 22 DM Architecture: This figure shows the process for deciding the trained Pareto-Optimal Solution to play a certain level of MarioAI. The output selection is based on the winner takes all strategy of highest preference value wins. This can be seen by the blue neuron with value of 0.88 being chosen as it has the highest value. The numerical values above the neuron represent the preference value for that specific trained network. The NE Mario Agent 0 is then decided as to be the "best" agent to play the level to maximize the overall game score. The DM weights are updated through Neuroevolution where the objective is to select the agent to maximize the overall game score for the specified level.

CHAPTER IV

FINDINGS

In this chapter the creation and selection of multi-modal agents will be observed. The main topics for which are: creating quality solutions through parameter tuning, visual representation of multiple modes of behavior, and DM selection efficiency for overall game scoring.

4.1 The Effects of Parameter Tuning on Learning Agents:

In this section, system parameters and hyper parameters will be modified to observe the effects on the convergence to a solution as well as multi-modal solution quality. The measuring metric for convergence estimation in this section will be the Hyper Volume indicator. This metric is used to quantify how close the solutions are to that of the true Pareto-front by calculating the volume of the hypercube between the solutions to that of an estimated reference point. The reference point used for the maximization of coins and distance will be the zero vector, thus the larger the distance from the reference point, the higher the hyper volume. For noisy and fractured problems like MarioAI, a maximum hyper volume of 1 might never be found, but the higher the better.

It was explained in the previous chapter that too large a distribution of mutated individuals combined with too confined a distribution of children results in lack of traversal in the search space (Figure 20.)

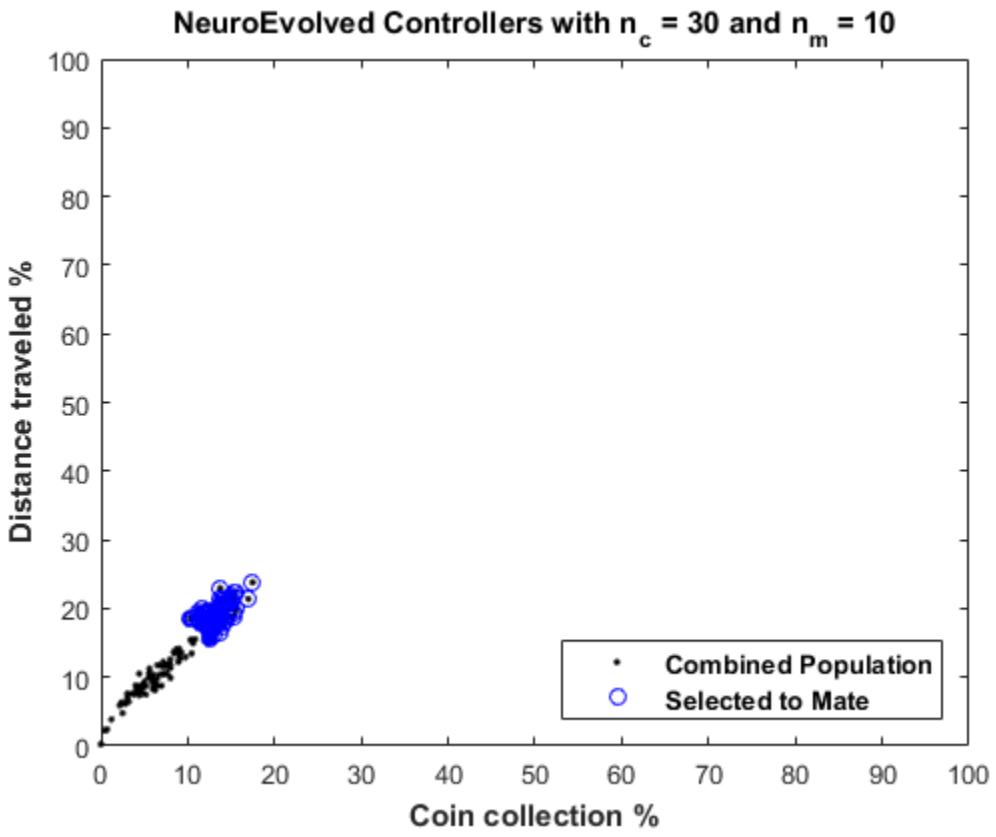


Figure 23 High Convergence, without Exploration. This figure depicts the issue of local optima entrapment.

Therefore it must be observed the effects of these distribution index's on the trained population. Each experiment, unless specified otherwise, was trained with a sampling rate of 15, a population size of 50, with a static crossover rate of 0.75 and a deterministically dynamic mutation rate as described in the previous chapter.

In the experiment above (Figure 23), n_c was set to 30, and n_m set to 10. By doing so, the population was able to find the correct form of direction as the population begins to align with the diagonal of the objective space. It can be seen though that the convergence pressure is too strong. The population gets stuck in local optima. Ideally, a high mutation parameter could be used to search locally to find a better solution.

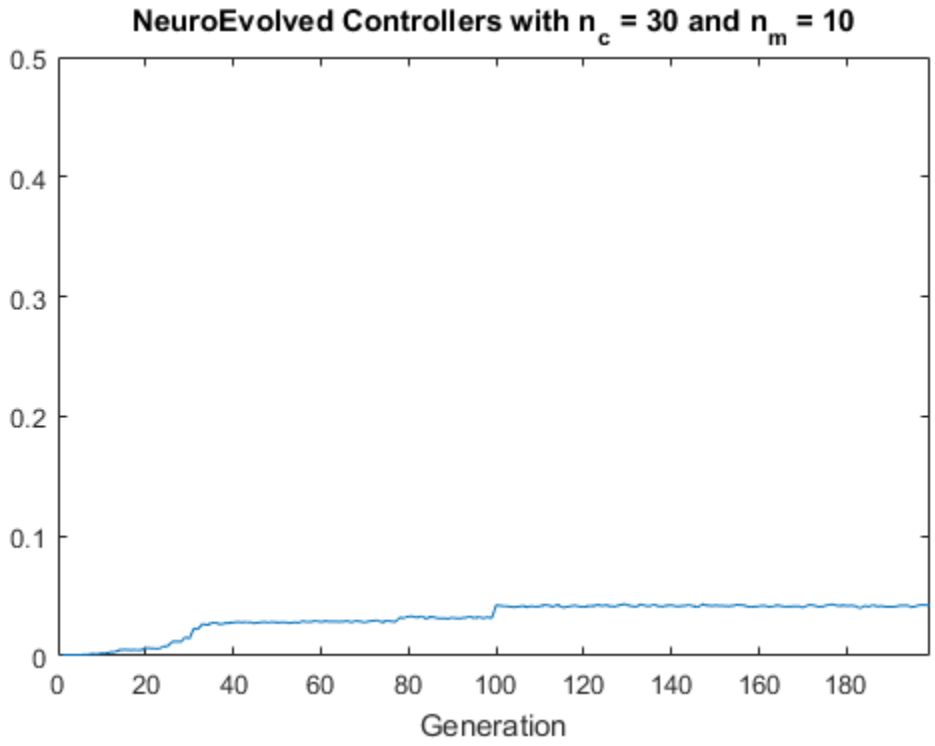
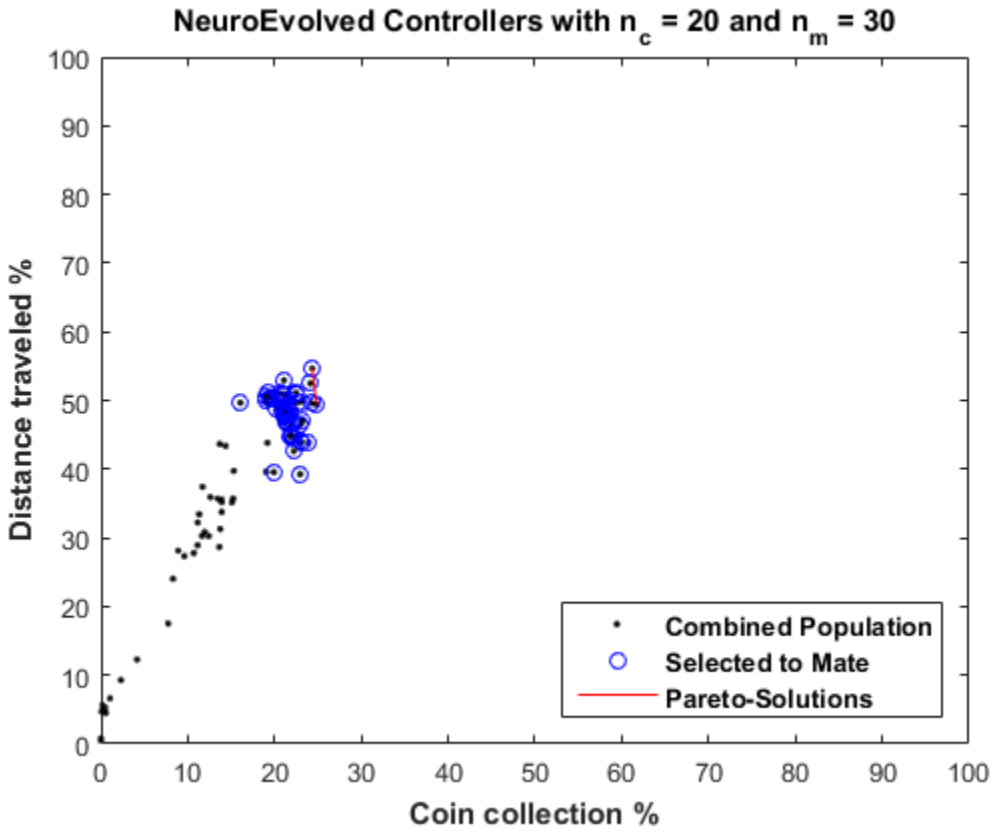


Figure 24 Hyper Volume of population stuck in local optima: This figure shows entrapment of local optima where the population gets stuck. At around generation 27 the newer generations fail to find better solutions until generation 100. At this point population has saturated around the local optima.

In this case however, perturbation of weights through mutation, due to the large distribution and no bounded weight values, creates offspring that perform worse than the parents and no exploration occurs. This can be further observed by the learned population's hyper volume Figure 24.

By reducing n_c and increasing n_m it can be seen that the population discovers better fitness values through a less restricted search. The population still suffers from diversity issues as individuals still cluster around local maxima, but the population is able to move farther along in the maximum distance and coin collection objective directions.

The point of entrapment around generation 50 with a maximum Hyper Volume of 0.15 and can be seen in Figure 25.



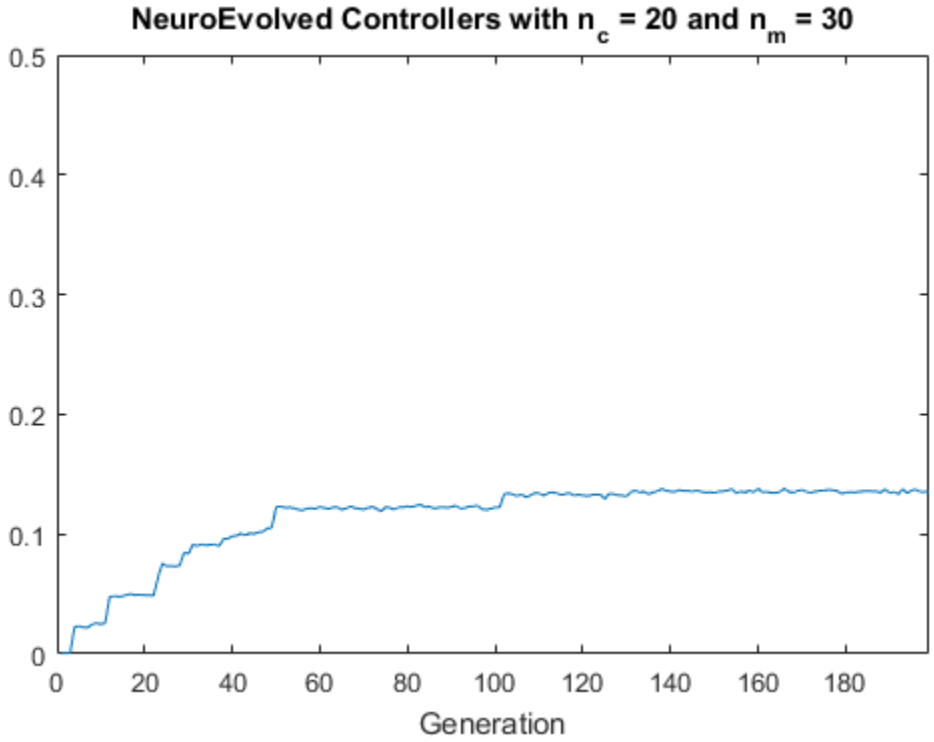
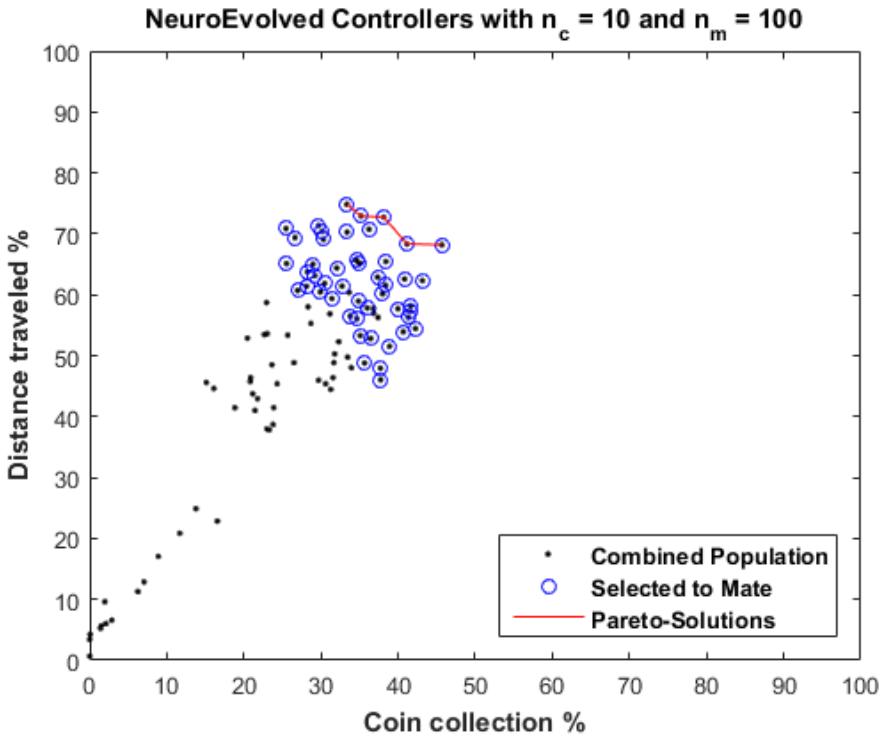


Figure 25 Neuro Evolved population and Hyper Volume with n_c = 20 and n_m =30

Based on the pattern seen so far, effects of the distribution parameters need to be lessened still. Since the previous experiments have shown issues with local maxima entrapment, n_c can then be set to $n_c = 10$ to lessen convergence pressure. Separately n_m can then be set to $n_m = 100$. This method will reduce perturbation within the mutated children so as to provide a mutated child distribution closer to the parents. This will account for the erratic weight distribution creating individuals that always get culled and thus result in higher selection pressure to local optima.

As it can be seen on the following page in Figure 26, this setup of parameters results in the best solution thus far. The population has multiple Pareto solutions that could contain varying behavioral modes. For example, the individual that bounds the Pareto-front from the left could provide better level traversal techniques at the tradeoff of maximizing coin collection. This setup also results in efficient convergence. It can be seen within the hyper volume plot that a hyper volume of around 0.15 is reached after only 10 generations.



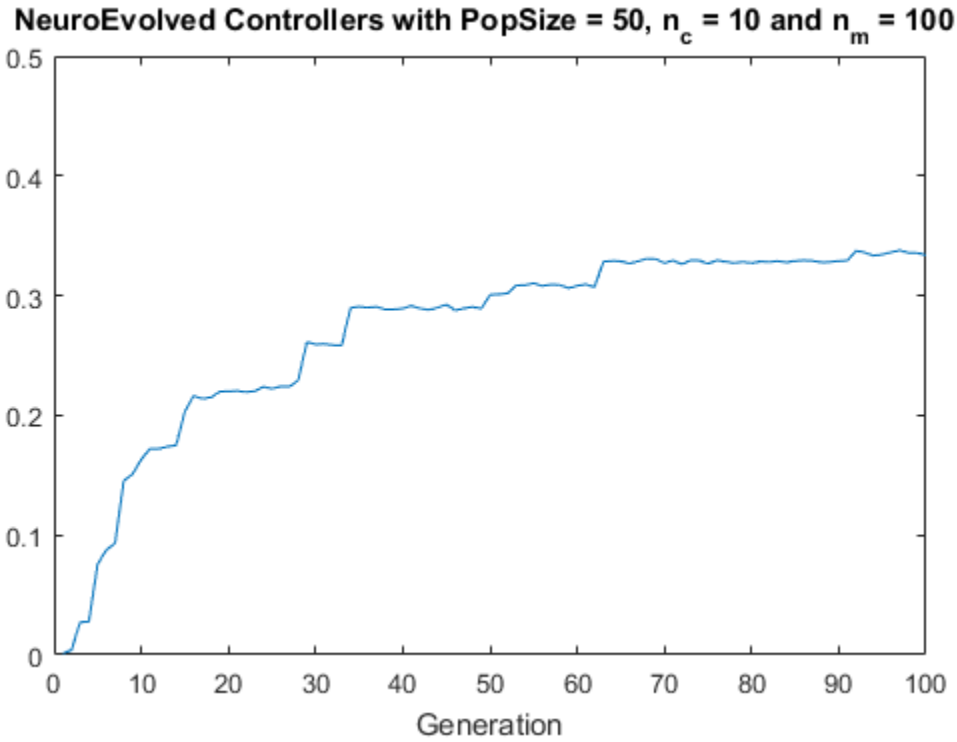
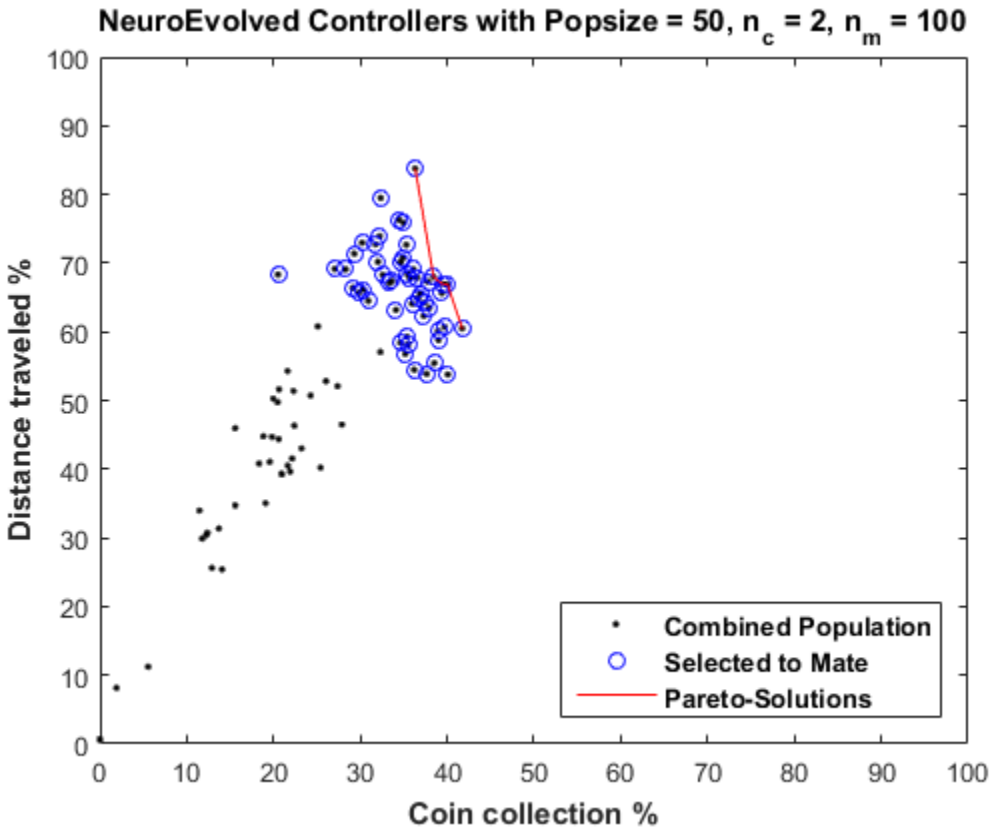


Figure 26 Best Solution Setup: These graphs show the optimal found setup for the MarioAI Neuro Evolved Networks.

This parameter setup has been found to provide the most optimal so far but additional testing is needed to validate this claim. By lowering the crossover distribution index further to that of $n_c = 2$ (Similar distribution function of binary crossover), exploration will be valued over exploitation as more children will be created with weights outside of the bounds of the parent's weighted values.

Looking at Figure 27 on the following page, it can be seen that the consequence of decreasing n_c to 2 also decreases the convergence rate within the Hyper volume. What took the previous population with $n_c = 10$ only 10 generations to reach a hyper volume of around 0.15, it takes this population almost 60 generations. Additionally it can be seen that agents producing higher values of distance completion but pressure of exploitation in the direction of the coin collection objective is low. Therefore, it can be said that the previous setup of $n_c = 10$, $n_m = 100$ to be the optimal parameters for the creation of quality multimodal solutions.



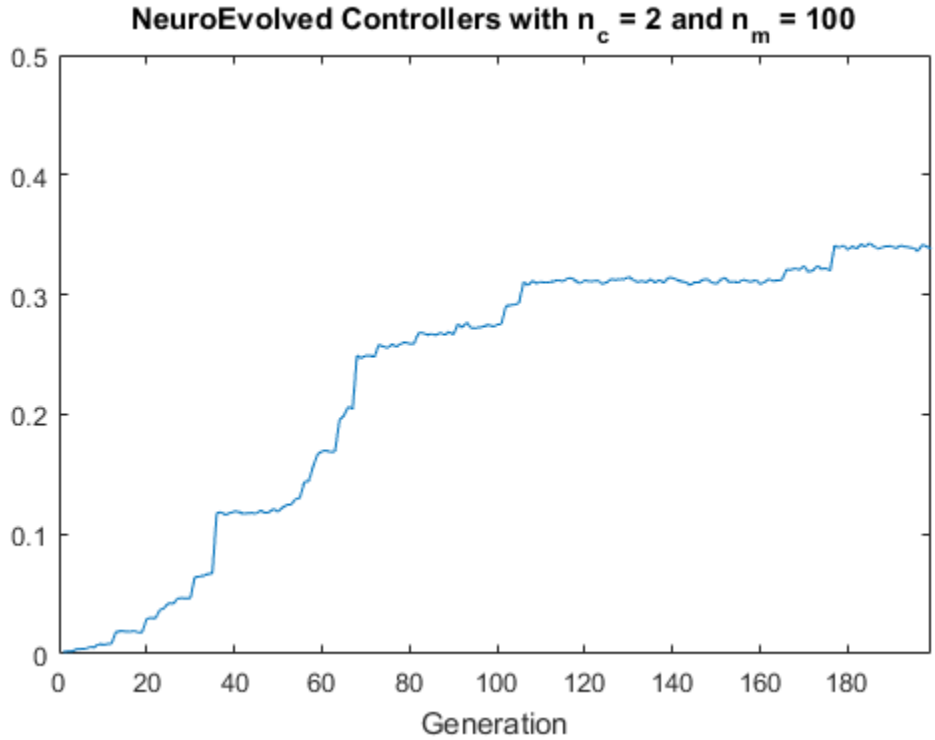
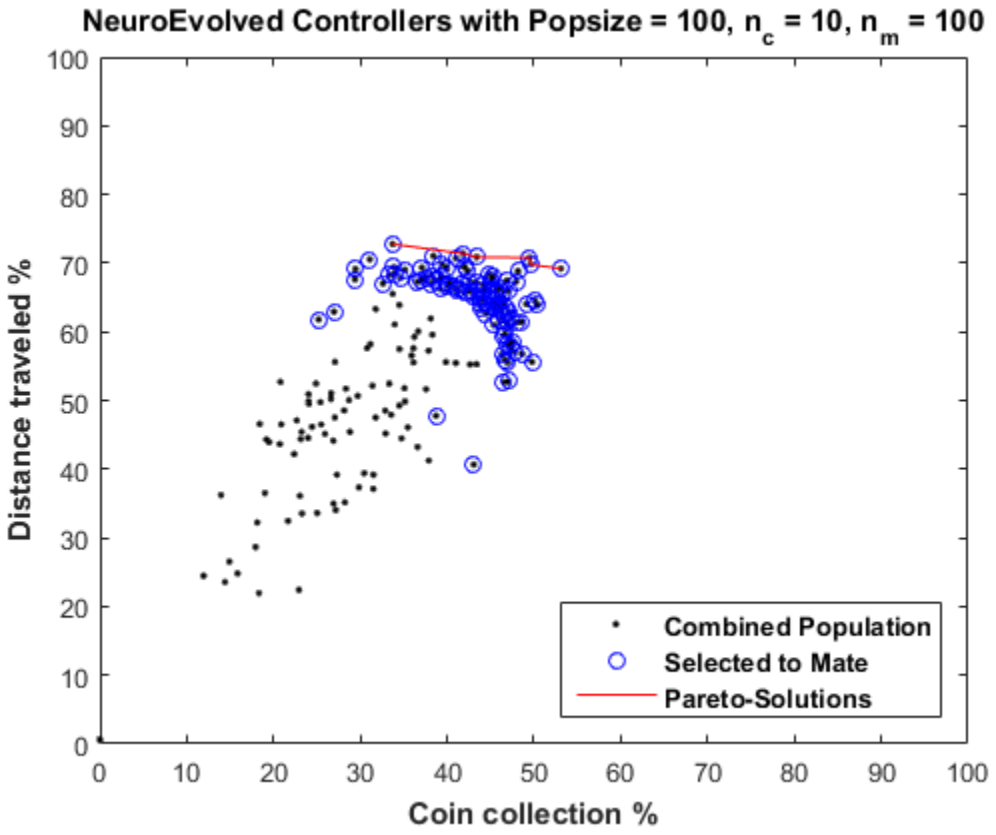


Figure 27 Exploration over Exploitation: These graphs display the effect of reducing the convergence pressure of the exploitation mechanism n_c.

It is interesting to determine if population size has any additional benefits to convergence speed and multimodal Pareto solution creation.

The findings of this experiment show that increasing the population size would result in a slightly better solution set. Additionally given that the hyper volume in Figure 28 is still marginally increasing after 100 generations, the total generations was set to 200 to determine if better performance could be found. This tradeoff is not computationally inefficient however as it is increased by a factor P of the population size.



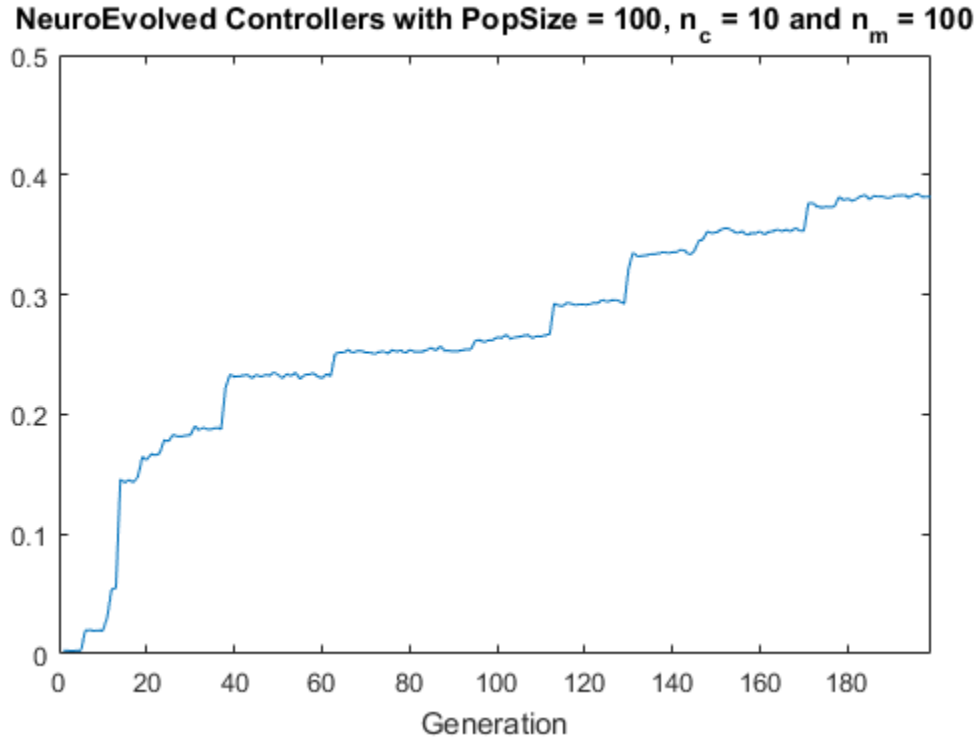


Figure 28 Effects of Increasing Population Size on Optimal Parameter Setup

Lastly, the optimal parameters are tested for statistical significance. In Figure 29 are the results of evolving different neural networks over five different runs. The parameters trained on are in the table below (*Layer Size {Input Neurons, Hidden Neurons, Output Neurons}):*

Population Size = 100	***Generations = 200***
Sample Size = 15	***Layer Size = {25, 20, 5}***
Crossover Probability = 0.70	$n_c = 10$
Mutation Probability = Dynamic	$n_m = 0.1$

Observing the average hyper volume in Figure 29, it can be seen that even though the standard deviation of convergence at the beginning is large and radical, the convergence rate is consistent as the standard deviation begins to reduce after 60 generations. The large variance between each training session is due to the high variance in the objective values. This can be further reduced through a higher sampling rate, but due to the computational complexity, the rate was kept at 15.

The resulting multi-modal Pareto solutions created can then be considered as both quality and consistent solutions.

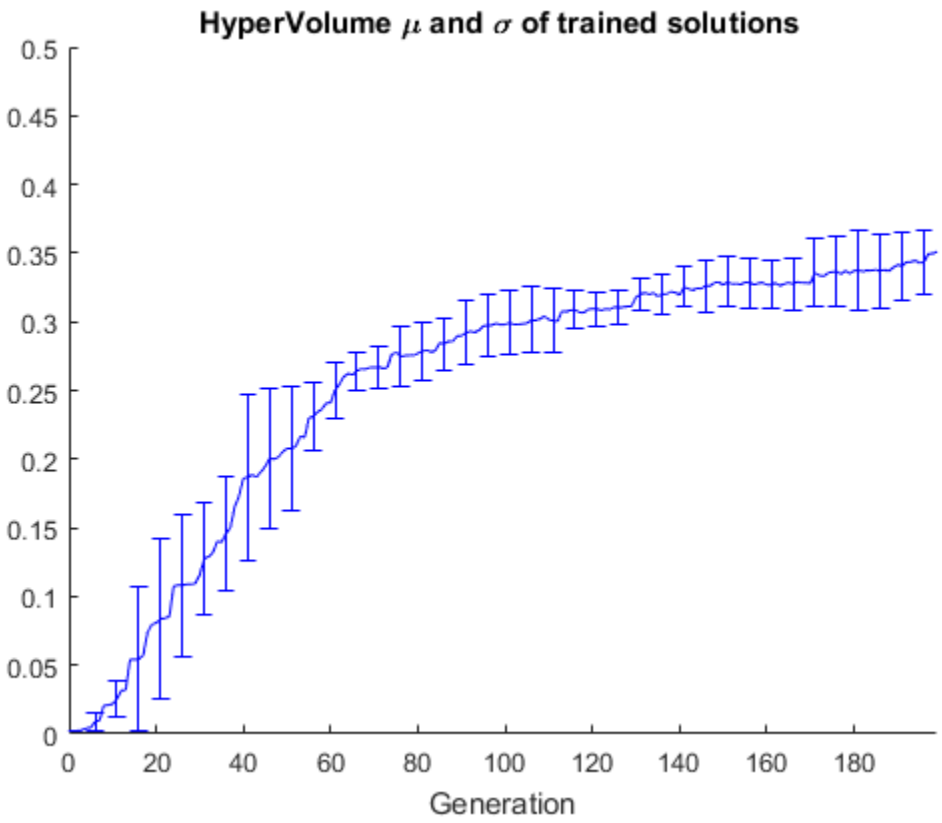


Figure 29 Average Hyper Volume of Optimal Parameters: This graph shows the average of hyper volume over five different runs where the line plot is the average hyper volume and the error bars represent the standard deviation. The rate of convergence varies greatly from generation 20 to generation 40. However as populations approach 100 generations almost all converges to the same hyper volume.

4.2 **Visual Representation of learned Multimodal Behavior**

Trying to quantify learned behavior without any predefined policy is a complex task. Solution quality can be graphically observed and estimated by the hyper volume indicator, but there is no better method in MarioAI that physical visualization. In this section the MarioAI environment will be used to find interesting forms of learned multi-modal behavior. The Neuro evolved solution used for this experiment is the one shown in Figure 28 since the Pareto-optimal solutions share similar distance passing fitness values, but widely varying coin collecting fitness values. Meaning each individual could have some interesting behaviors when attempting to complete a level.

The multi-modal Mario agents are given the order of which they appear on the Pareto-optimal front from left to right. This means that a Mario agent having the ending value of 0 depicts the individual that has maximized distance travel but has the tradeoff of collecting the least amount of coins during training. The solution set in question has 6 Pareto-optimal solutions, resulting in an ending value range 0 to 5.

Take first the complex behavior involved in different aspects of coin collecting. Sometime Mario is given simple states where the coins are laid directly in front of him, and sometimes Mario is faced with a highly fractured state like the one captured in the top left image of Figure 30. In this captured frame, agent number 4 jumps up a hill and is confronted by an enemy blocking his coin. He responds by leaping up to kill the enemy and simultaneously grabbing the coin that the enemy was blocking.

Next, a frame of agent 1 is caught showing complex learned behavior in the top right image of Figure 30 that was never explicitly handled through any input/output methodology.

Figure 30 Complex Multi-Modal Behavior: Going from top left, clock-wise: Agent 4, Agent 1, and Agent 0 showing off complex maneuvers for overcoming complex aspects of the game.

Agent 1 learned that if after he lands on top a turtle he can run and grab the shell and use it as a form of protection. He uses this turtle shell shield to bypass an enemy that can't be killed through normal means.

Lastly agent 0 is found completing the difficult stair problem with ease as he leaps from the top of the ascending stairs on the left and lands perfectly on the top of the descending stairs (Figure 30 bottom image). Additionally, agent 0 is showing similar behavior as that of agent 1. This gives visualization of shared learned behavior through crossover.

4.3 Testing the Neuro Evolved DM Ensemble

Given the creation of strong multi-modal agents through multi-objective optimization and parameter tuning, it has been shown that a Pareto-optimal solution set is created with differing complex behaviors. Selecting the best individual with the optimal set of skills is imperative to maximizing overall game score. In this section the experiment for testing the neuro evolved DM will be observed.

The DM was allowed to train on a random data set from seed 21 and upward. Seeds 0 – 20 were saved for later as the testing set. The test run was composed to allow candidates to perform and score on each level type for every difficulty 0 to 3. Each contestant's specific level score was averaged over a number of trials where each trial was of a generated level of a different seed.

The GA parameters for training the DM are listed below in the table (Output Layer for Layer Size is total number of agents to select from the Pareto front):

Population Size = 100	***Generations = 100***
Sampling Rate = 8	***Layer Size = {3, 10, 6}***
Elite Size = 20	***Crossover Rate = 0.90***
Mutation Rate = 0.10	

Four DM (0 – 3) were created for the sake of statistical significance. These DMs are each their own neural network ensembles for selecting which individual agent gets to play the level for maximum overall game score. Additionally, a comparable GA from previous studies was trained using a modified version of the overall game score as a single objective highly tuned weighted fitness function. This was to provide a baseline score and comparison of single objective to multi-objective solutions. The GA was trained with a population of 100 and over 400 generations on the same training data as the NSGAII trained NN. Each individual agent was allowed to play to

verify the agent with the highest overall game score as well as to determine if the DM scored better than individuals alone. Lastly, agents were chosen at random to play one level in a run. Random selection was evaluated over 10 runs to get a statistically significant average score. This was to compare and see if a DM could score better than a random selection of individuals.

The equation for overall game score is based on multiple weighted criteria:

$$\text{Game } Score = \{(d * lengthOfLevelPassed) + winStatus * w + MarioHealth * h + (c * totalCoins) + killsTotal * k + timeLeft * t\}$$

Where all weights (*d, w, h, c, k, t*) are fixed except the value of *d* and *c*. If the level type is the bonus stage then the distance weight, *d*, is halved and the coin weight *c* is doubled.

The scores for all the runs can be seen in the table below:

	Total Coins	***Total Distance***	***Competition Score***
GA NN	1344	12136	65104.0
NSGAII NN (0)	2151	13077	70865.2
NSGAII NN (1)	1940	12626	67815.2
NSGAII NN (2)	1943	13562	70992.2
NSGAII NN (3)	1205	13139	66850.0
NSGAII NN (4)	**2162**	11899	65620.2
NSGAII NN (5)	1633	13031	68373.4
NSGAII NN(Random)	1840	12946	68326.0
NSGAII DM (0)	2035	**13568**	**72128.2**
NSGAII DM (1)	1871	13127	70572.0
NSGAII DM (2)	2111	13399	71795.0
NSGAII DM (3)	1949	12667	68251.0

Table 5 Competition Scores of 20 seed run through four incrementing levels of each level type: Best Scores are bolded. Best individual scores are underlined. NSGAII NN (4) had the best individual run as well as the best overall total Coins collected.

It can be seen that all NSGAII agents scored better than the highly tuned GA agent. The best overall agent is NSGAII NN (2) as it was able to maximize the distance traveled which held the most weight to the overall game score. The second best individual was NSGAII NN (0) as he collected the second most amounts of total coins and a good balance of total distance.

NSGAII NN (4) was able to collect the most amount of coins throughout the run but resulted in the second worst competitive score. This shows that agents that are the best at one objective will not always result in the best overall score. Lastly all of the individual NSGAII trained agents outperformed the single objective GA agent.

For comparison of decision makers, all scored higher than the GA, and all but NSGAII DM (3) scored better than the average randomly selected agents. Agents 0 and 1 scored better individual scores than NSGAII DM (3) and NSGAII DM (1). The best possible overall run for the individual agents in the Pareto-solution set and the best performing NSGAII DM (0) can be seen below:

	Overground Level	***Bonus Level***	***Marathon Level***
Difficulty (0)	1	1	1
Difficulty (1)	4	3	3
Difficulty (2)	3	5	4
Difficulty (3)	4	3	4

Table 6 Map of best scoring agents per each level and each type:

	Overground Level	***Bonus Level***	***Marathon Level***
Difficulty (0)	1	4	1
Difficulty (1)	4	4	1
Difficulty (2)	3	4	3
Difficulty (3)	3	4	3

Table 7 Best Performing NSGAII DM (0) Selection Map

The maximum score can be calculated as to be 77104 and the difference then from the best DM and the maximum score is 4,975.8. This is a substantial difference but due to the limited amount of visible information as inputs and training in a noisy environment it is understandable outcome. What is interesting however is the selection error of the DM. The DM was only able to correctly select 4 out of 12 of the best agents for the specified level (Table 7). What is most surprising however is that many incorrect selections are $\mp$ 1 agents away from absolute best solution. This provides information that the DM has learned to generalize selection of agents for the testing levels.

CHAPTER V

CONCLUSION

In this thesis, the multi-objective Neuro-evolutionary method for evolving multimodal behavior was presented. This chapter will assess the findings and conclude the results.

5.1 Evaluation of Findings

This section is dedicated to the results of the experiments tested as well as the understandings of the outcomes.

5.1.1 Parameter Selection

It became very apparent how profitable tuning the parameters dedicated to searching the objective space was. If convergence pressure is too high due to a high value of the crossover distribution index n_c, then the controllers suffer from high probability of entrapment within a local optima. This entrapment forces creation of a population of locally saturated Mario agents that would share common forms of behavior, negating the benefits of multi-objective evolution. Mutually, if perturbance of weight parameters is too high, then the resulting offspring will have a high probability of being worse than the parent solutions. Thus the mutation operator becomes a null factor and leaves the resulting Mario agents to search using crossover alone. Therefore a well-balanced parameter setting is necessary for optimal learning of multi-modal behavior.

5.1.2 Multimodal Behavior Assessment

Training using multi-objective NE created a Pareto front of solutions, each depicting some form of differing behavior. These complex behaviors were used to overcome difficult tasks within the environment. This ranged from scaling to the tops of hills to collect coins, to using shells as a shield from enemies. Yet these agents were not perfect. Many agents still suffered from falling into gaps, running into enemies, and getting stuck on walls. Some also displayed very behaviors. Many agents that would maximize distance would revert into just running as fast as possible while continuously jumping. This makes sense however, as the most of the state changes and difficult tasks happened on the lower half of the environment (Jump over gaps, Dodge enemies). Another interesting behavior was the every Mario agent that was trained, never learned to go left. All the agents would run to the right while attempting to collect coins and dodge obstacles along the path. This could be explained through the driving nature of the main objective to reach the end of the level and any distance backtracked would result in a negative of the fitness value.

5.1.3 Comparison of Single Objective to Multi-Objective NE

It was observed within this study, the quality as well as efficiency in Mario agents developed through multi-objective criteria as compared to the extensive hand tuning necessary to creation of single weighted objective trained agents. Not only was a higher amount of generations necessary to provide comparable results, but also the time dedicated to assessing which objective within the combined single objective fitness function provided the most positive feedback to the system. Therefore, it can be stated that for systems that require the need to exhibit multiple modes of behavior in a fractured and noisy environment, then the use of multi-objectives is not only more robust, but also less complex to the designer.

5.1.4 Evaluation of the Neuro Evolved Decision Maker Ensemble

After quality Pareto-optimal solutions were created, a method for deciding which agent to use within the Pareto front was proposed. This method not only showed the effectiveness of learning on minimal information of a noisy environment, but that a basic decision maker can be created using methods of Neuroevolution. The best DM provided not only better results than the single objective trained NN, but also showed to be better than any one individual and a random selection of individuals for each level. Therefore using a DM ensemble proves to be an appropriate solution for overall goal optimization using multi-objective trained agents.

Given that the Mario agents are each trained on sub objectives of the overall game objective it can then be postulated that any ensemble of multimodal agents trained on sub objectives of a system could then be selected for optimal performance of the overall system. This idea can be assessed further in future work in other environments and deeper layers of NN ensembles could create different levels of abstraction and behavioral complexity.

5.2 Conclusion

This thesis explored the effectiveness of multiple objective genetic algorithms to create intelligent multimodal controllers for a modified version of the platform game Super Mario Bros called MarioAI. MarioAI was used as the testbed to provide a system that required the use of multiple behaviors to overcome the fractured environment states through differing gameplay tasks (i.e. collect coins, don't get hit by an enemy). Furthermore, MarioAI provided additional problem complexity in the form of environmental noise through level seeds. The differing levels seeds provided the need to create a controller that could generalize to the state space and not become deterministic to individual levels.

This method of combining NSGA-II and neuro evolved ANNs provided a means to create controllers that could support the aspects necessary to overcome the difficulty of the MarioAI test bench. Results showed that it's possible to train agents on sub objectives (maximizing coin collection and maximizing distance traveled), of the overall game objective to create a set of Pareto-optimal solutions with multi-modal behavior. As well as exhibiting multiple forms of complex behavior, robust agents were created that could learn to generalize over many varying levels and types, and could outperform a competing single objective neuro-evolved MarioAI controller. The creation of quality agents was not trivial however, as the tuning of objective space searching parameters was needed to provide the most optimal and diverse Pareto-solution set of MarioAI agents.

Lastly, a DM ensemble was proposed as a means of selecting which agent from the Pareto-front was allowed to play a specific level. It was found that a DM could learn through NE on minimal inputs to provide better average overall scores than any individual agent or collection of randomly selected agents. This then means that a multi-objective NE system could train on precise sub objectives of a system and then allow the neuro evolved DM to abstractly select these multimodal NNs to provide the best possible solution. The results also show that while this is an interesting claim it is not fail proof as not all DM learn a better scoring solution. Therefore more research could be spent on creating better DM structure and design.

REFERENCES

[1] Kaku, Michio. The Future of the Mind: The Scientific Quest to Understand, Enhance, and Empower the Mind. N.p.: n.p., n.d. Print.

[2] P. Henaff, Chocron, O., "Adaptive learning control in evolutionary design of mobile robots," in Systems, Man and Cybernetics, 2002 IEEE International Conference on , vol.3, no., pp.5 pp. vol.3-, 6-9 Oct. 2002 doi: 10.1109/ICSMC.2002.1176065

[3] J. Togelius and S. M. Lucas. “Evolving robust and specialized car racing skills.” In Evolutionary Computation, 2006. CEC 2006. IEEE Congress on, pages 1187 1194. IEEE, 2006.

[4] Schrum, J.; Miikkulainen, R., "Evolving Multimodal Networks for Multitask Games," in Computational Intelligence and AI in Games, IEEE Transactions on , vol.4, no.2, pp.94-111, June 2012 IEEE.

[5] M. Mitchell, “An introduction to genetic algorithms.” Cambridge, Mass.: MIT Press, 1996.

[6] De Jong, K. A. “An Analysis of the Behavior of a Class of Genetic Adaptive Systems.” PhD thesis, The University of Michigan, Ann Arbor, MI. University Microfilms No. 76-09381. 1975

[7] D. Goldberg, “Genetic Algorithms in Search, Optimization and Machine Learning.” Boston: Addison-Wesley Longman Publishing Co., 1989.

[8] Blickle, T. and Thiele, *L.*: “A Comparison of Selection Schemes used in Genetic Algorithms .“ (2. Edition. TIK Report No. 11, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zürich, Switzerland, 1995.

[9] M. Hagan, H. Demuth and M. Beale, Neural network design. Boston: PWS Pub., 1996.

[10] Stinchcombe, M.; White, H., "Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions," in Neural Networks, 1989. IJCNN., International Joint Conference on , vol., no., pp.613-617 vol.1, 0-0 1989

[11] Belew RK, McInerney J, Schraudolph NN “Evolving networks: using the genetic algorithm with connectionist learning.” In: Langton CG, Taylor C, Farmer JD, Rasmussen S (eds) Proceedings of the 2nd Conference on Artificial Life. Addison-Wesley, Reading, pp 511–548, 1992)

[12] Xin Yao, "Evolving, training and designing neural network ensembles," in Intelligent Engineering Systems (INES), 2010 14th International Conference on , vol., no., pp.11-11, 5-7 May 2010)

[13] D. Whitley, T. Starkweather, and C. Bogart, “Genetic algorithms and neural networks: Optimizing connections and connectivity,” Parallel Comput., vol. 14, no. 3, pp. 347–361 1990.)

[14] P. Melendez, 'Controlling non-player characters using support vector machines', in Future Play, Vancouver, 2009, pp. 33-34.

[15] F. Gomez, J. Schmidhuber, and R. Miikkulainen. “Accelerated neural evolution through cooperatively coevolved synapses.” The Journal of Machine Learning Research, 9:937–965, 2008.

[16] P. Verbancsics and K. O. Stanley. “Evolving static representations for task transfer.” The Journal of Machine Learning Research, 11:1737–1769, 2010.

[17] E. J. Hastings, R. K. Guha, and K. O. Stanley. “Automatic content generation in the galactic arms race video game.” Computational Intelligence and AI in Games, IEEE Transactions on, 1(4):245–263, 2009.)

[18] Stanley, K.O.; Bryant, B.D.; Miikkulainen, R., "Real-time Neuroevolution in the NERO video game," in *Evolutionary Computation, IEEE Transactions on* , vol.9, no.6, pp.653-668, Dec. 2005

[19] Jin-Hyuk Hong; Sung-Bae Cho, "Evolution of emergent behaviors for shooting game characters in Robocode," in *Evolutionary Computation, 2004. CEC2004. Congress on* , vol.1, no., pp.634-638 Vol.1, 19-23 June 2004

[20] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, 'A fast and elitist multiobjective genetic algorithm: NSGA-II', IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, 2002.

[21] N. Srinivas and K. Deb, “Multiobjective function optimization using nondominated sorting genetic algorithms,” *Evol. Comput.*, vol. 2, no. 3, pp. 221–248, Fall 1995.

[22] K. Deb, “An efficient constraint handling method for genetic algorithms”, Computer Methods in Applied Mechanics and Engineering, vol. 186, no. 2-4, pp. 311-338, 2000.

[23] K. Deb, A. Kumar. “Real-coded Genetic Algorithms with Simulated Binary Crossover: Studies on Multimodal and Multiobjective Problems”, Complex Systems, vol. 9, pp. 431-454 1995

[24] Schaffer JD, Whitley D, Eshelman LJ “Combinations of genetic algorithms and neural networks: a survey of the state of the art.” COGANN-92. IEEE Press, New York

[25] K. Deb, A. Kumar. “Simulated Binary Crossover for Continuous Search Space”, Complex Systems, vol. 9, pp. 115-148 1995

[26] Schaffer JD, Whitley D, Eshelman LJ.”Combinations of genetic algorithms and neural networks: a survey of the state of the art.” Proceedings of an international workshop on the combinations of genetic algorithms and neural networks (COGANN-92). IEEE Press, New York, 1992.

[27] Booker, L. B. , "Recombination Distribution for Genetic Algorithms," in *Foundations of Genetic Algorithms* , edited by D. Whitley (Morgan Kaufmann, San Mateo, 1993

[28] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis. The Mario AI Championship 2009–2012. AI Magazine, 34(3):89–92, 2013.

[29] Pedersen, C.; Togelius, J.; Yannakakis, G.N., "Modeling player experience in Super Mario Bros," in Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on , vol., no., pp.132-139, 7-10 Sept. 2009

[30] J. Ortega, N. Shaker, J. Togelius, and G.N Yannakakis. Imitating human playing styles in super Mario bros. Entertainment Computing, 4(2):93-104, 2013.

[31] Lars Dah Jørgensen and Thomas Willer Sandberg. “Playing Mario using advanced AI techniques.” Advanced AI in Games, 2009

[32] Togelius, J.; Karakovskiy, S.; Koutnik, J.; Schmidhuber, J., "Super Mario Evolution," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on* , vol., no., pp.156-161, 7-10 Sept. 2009

[33] N. Kohl, “Learning in fractured problems with constructive neural network algorithms,” Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas, Austin, 2009.

[34] J. Schrum and R. Miikkulainen, 'Discovering Multimodal Behavior in Ms. Pac-Man through Evolution of Modular Neural Networks', *IEEE Trans. Comput. Intell. AI Games*, pp. 1-1, 2015.

[35] Herrera, F., M. Lozano, and J. L. Verdegay. "Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis." Artificial Intelligence Review 12.4 (1998): 265-319. Web

[36] Braun, H., and Weisbrod, J. “Evolving feedforward neural networks.” In *Proceedings of ANNGA93, International Conference on Artificial Neural Networks and Genetic Algorithms*. Innsbruck: Springer-Verlag, 1993.

[37] Deb, Kalyanmoy ; Goyal, Mayank. “A combined genetic adaptive search (GeneAS) for engineering design Computer Science and Informatics,” 26 (4). pp. 30-45. ISSN 0254-781, 1996.

[38] S. Zanetti and A. E. Rhalibi. “Machine learning techniques for fps in q3.” In Proceedings of the 2004 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, ACE ’04, pages 239–244, New York, NY, USA, 2004. ACM. ISBN 1-58113-882-2

[39] J. Togelius and S. M. Lucas. Evolving controllers for simulated car racing. In Evolutionary Computation, 2005. The 2005 IEEE Congress on, volume 2, pages 1906–1913. IEEE, 2005

[40] Jin, Y., Branke, J., “Evolutionary optimization in uncertain environments – a survey.” IEEE Transactions on Evolutionary Computation 9 (3), 303–317.)

VITA

Ian Patrick Finley

Candidate for the Degree of

Master of Science

Thesis: EVOLVING INTELLIGENT MULTIMODAL GAMEPLAY AGENTS AND DECISION MAKERS WITH NEUROEVOLUTION

Major Field: Electrical Engineering

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2015.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2012.

Experience:

June 2011 to August 2011
Undergraduate Researcher, Oklahoma State University, Stillwater, OK. Duties: worked under Dr. Chandler to create a wirelessly controlled RC Car driven using an android mobile phone. This was created as project to pique electrical engineering interests in younger students high school and below.

August 2012 to December 2012
IEEE R5 Robot, Capstone II Design project to produce a fully autonomous mobile robot utilizing SLAM techniques to navigate a course simulating a forest destroyed by wildfire.

March 2012 to May 2012
Engineering Consultant, Oklahoma State University, Stillwater, OK. Duties: designed and developed a bee sensing and shocking circuit to be used at the Understanding of Bees Convention 2012 in Istanbul, Turkey.

January 2013 to November 2013
Software Engineering Consultant, Chesapeake Energy, Oklahoma City, OK. Duties: designed and implemented an automated project management and test and measurement tool for oil well development.

May 2014 to August 2014
Electrical Engineering Intern, VADovation, Oklahoma City, OK. Duties: aided in the development of the controller system for a new Right Ventricle Assist Device (RVAD) heart pump to be placed in patients with left and right ventricular failure.