5-2024

# Super Mario Evolution by the Augmentation of Topology

Russell A. Autin
*University of New Orleans*, raautin@uno.edu

# Super Mario Evolution by the Augmentation of Topology

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Machine Learning and AI

by

Russell AAron Autin
B.F.A Mississippi State University, 2012

May, 2024

# Contents

# List of Figures

# List of Tables

# Abstract

This paper describes the creation and development of an implementation of the NeuroEvolution of Augmenting Topologies (NEAT) architecture to train an agent to play Super Mario Brothers. Building off of a basic implementation of NEAT, this thesis project shows the process of refining the fitness calculation that ranks the networks in the population and also defines the creation and application of a dataset to train the agent. The use of a dataset to train an agent is a novel idea in the world of reinforcement learning because, generally, reinforcement learning trains an agent to complete a singular task like the pole balancing problem. Training an agent to play something as complex as a video game, however, requires that an agent is exposed to as many different situations that occur within the game as possible. The goal of this thesis project is to create an agent that has a robust general understanding of how to play the game, such that it is able to react to new situations that were not seen in training. The results of this thesis project show that this generalized understanding is possible via neuroevolution, when given enough training time, a properly designed fitness calculation, and a properly applied dataset of scenarios.

# Chapter 1

# Introduction

Ever since I learned about the existence of evolutionary algorithms in high school, I was fascinated by the concept that computer code could work on a problem and, over time, learn how to solve it. This was around the time when personal computing was making the switch to 64 bit processing, so much of the work in this area was still academic and had few concrete applications. Fast forward to my time working as a quality assurance tester for a major game producer, Electronic Arts (EA), which is where I began to see the power that these algorithm could provide. EA was beginning to develop automation processes for game testing and the department I worked in was the testing ground for these automation techniques. They were still approaching the automation process from a traditional, declarative programming style, where you explicitly tell a bot what actions to take; however, I realized that if you could figure out how to train an agent to learn how to play the game, you would have a much more powerful, adaptable system on your hands. Instead of explicitly telling a bot to check that X and Y inputs work correctly, the process of training an agent would cover not just the expected usages of an input but also many other unusual or irrational input combinations, which are all necessary for proper quality assurance testing. The seeds of this thesis project had been planted in my mind and now all I needed was the fertilizer of academia for it to grow into a proper experiment.

This thesis project is an attempt to create an agent that learns how to effectively play a game by selecting the correct action, or behaviour, for any given situation that may occur. To achieve this I have selected neuroevolution techniques as the method of training an agent because they have been shown to excel in continuous environments with sparse information on the effects of an individual action [30, 13]. This is because neuroevolution algorithms perform a search over the space of possible behaviours [30] an agent can make, as opposed to traditional reinforcement learning techniques which search for optimal valuation functions for an action [4, 31].

The basics of neuroevolution are inspired by the same evolutionary forces that have shaped life on this planet. In the beginning, a population of simple organisms appear and then, by a process of mutation and reproduction, evolve into more complex and optimized organisms that are able to survive and prosper in their given environments. Neuroevolution takes this concept and applies it to the optimization of artificial neural networks, or ANNs. This process of optimization begins with the generation of basic neural network structures, which are called *Genomes*. The concept of a genome in neuroevolution is analogous to how

the DNA sequences, or *genotype*, define the final structure, or *phenotype*, an organism takes. A genome, much like a DNA sequence, is made up of a collection of genes and these genes define what nodes an ANN has, how those nodes are connected to each other, and what weight each node connection has. In this way, the genome is very much the genotype of a neural network because it directly defines what the structure that neural network will have. Then, with their structures defined, a neural network is given a problem to solve and assessed on how well its current structure performs. This assessment is done via calculating the value of a predetermined *fitness function*. Genomes with a higher fitness value are considered more adapted to the problem and are allowed to mate with other high performing genomes, with the goal of creating child genomes that are even more adapted to the problem. Genomes with low fitness values are removed from the population and replaced with the children of high performing genomes. Additionally, both the high performing genomes and their children have a random chance of being slightly mutated by having additional connections or nodes added to their collection of genes. This process of evaluation, reproduction, and mutation is known as a generation of evolution and as neuroevolution algorithms progress from generation to generation, the genomes become more and more optimized at solving the given problem. This is a simplistic overview of the neuroevolution process and will be explained in greater detail in **Chapter 4**.

The game that was selected for this project is Super Mario Brothers (SMB) for the Nintendo Entertainment System (NES) and it is one that is near and dear to my heart. I have many fond memories of getting together with friends and family to see who could get the farthest in the game. SMB is a rather simple game by modern standards as it only allows the player to move forward, backwards, and jump over obstacles/enemies. The graphics are also pretty simplistic due to the hardware limitations of the NES. Everything that is seen on the screen is made up of repeated 16 pixel by 16 pixel tiles as seen in Figure 1.1. Despite this relative simplicity, it fully embraces the tried and true method of "simple to learn, hard to master" gameplay that makes a game truly a classic. A detailed explanation of SMB, the NES, and the software used to train and run a neural network will be done in **Chapter 5**.


Figure 1.1: Big Mario surveying his options.

This paper is organized to start by introducing the basic concepts and then builds upon them to provide an understanding of the systems and technologies used in this project. **Chapter 2** is a discussion of related works that informed and inspired this project. **Chapter 3** then moves on to describe current deep learning techniques and how they are ill suited to this class of problems. **Chapter 4** gives an introduction to the history of neuroevolution algorithms and how a novel approach to the design of these algorithms make them an effective choice for this project. **Chapter 5** explains the systems and technologies used in this project and how the improvements I made to the neuroevolution system will facilitate the creation of an effective game playing agent. **Chapter 6** is a review of the results of my experiments for this project and how those results shaped improvements for future training runs. Finally, **Chapter 7** details some future work that can be done to improve upon these results and provides some conclusions based on the results of my experiments. Although the results of this experiment did not create an agent that could *beat* a level of SMB, the results do show that it is possible to create an agent that has generalized knowledge of how to play the game and make effective progress through a level.

# Chapter 2

# Related Work

This chapter will highlight the previous work that has been done in this area of research. The first of which will be about the concept of neuroevolution from which the algorithm used in this project is built. The next related work builds off of the concept of neuroevolution and details the goals of the creator of the algorithm this project is built off of. This section serves to provide context to the design decisions I made when improving the algorithm. The third, and final, related work deals with the concept of using memory locations as a means of input into a game playing agent. The system used in this thesis project doesn't use memory locations in the exact same manner; however, it serves as an interesting and inspirational look into how they can be used to provide data to a game playing agent.

## 2.1 Neuroevolution

In the 1990's neuroevolution was, and still is, a very promising field of research into the development of optimized neural networks [18]. The concept of deep neural networks had only recently been introduced and much of the focus of machine learning research was on the

efficacy of neuroevolution vs backpropagation for training ANN classifiers. At that time, it seemed neuroevoution was the preferred method of training ANNs, with Heinz Muhlenbien stating in his 1990 paper:

> "We conjecture that the next generation of neural networks will be genetic neural networks which evolve their structure." [19]

Unfortunately for neuroevolution, the computational power increases of the coming decades would show that backpropagated deep neural networks would become the state of the art for training ANN classifiers. However, all was not lost for neuroevolution; there remains at least one class of problems which backpropagation has not yet been shown to be a good application for. This class of problems, known as Reinforcement Learning problems, deal with an agent learning complex behaviours or constructing a decision making process in order to control an autonomous construct or process [18]. The reinforcement learning problems that neuroevolution seem to be best suited for are ones that require recurrent connections and need specialized architectures [12, 16, 3, 10]. For these tasks, architectures like *NeuroEvolution of Augmenting Topologies*, or NEAT [30, 29], and HyperNEAT [11, 28] have been created to construct these complex neural networks. This project uses the NEAT architecture as a basis for its implementation and then builds upon it by creating a custom dataset, generating initial populations, and developing a customized fitness function. These implementation details will be discussed in **Chapter 5**.

## 2.2 MarI/O

This project uses a library called *MarI/O* that constructs a population of genomes and mutates them during training. This library was originally created by Seth Bling [5] with the purpose of evolving an agent that is able to beat specific levels of SMB and Super Mario World (SMW), the spiritual successor to SMB that was released in 1991 on the Super Nintendo Entertainment System (SNES). The SNES is the next generation of console released by Nintendo in 1991. Seth Bling is a video game speedrunner and computer scientist who, until June 2020, held the SMW Any% speedrun record of 41.25 seconds. An Any% speedrun is a category of speedrunning where the runner is allowed to beat a game with any percentage of the game complete. This would be in contrast to a 100% speedrun, which requires the runner to collect all collectible items, explore all maps, etc before beating the game. He has also pioneered the use of arbitrary code execution to inject external code into a physical SNES. This allowed him to recreate the game Flappy Bird in Super Mario World using assets from the game and receiving input from the SNES controller.

The MarI/O algorithm is an implementation of a NEAT architecture pioneered by Kenneth O. Stanley and Risto Miikkulainen [30]. His goal was to evolve a neural network structure that could beat a specific level, Donut Plains 1, of Super Mario World as fast as possible. The algorithm was able to construct a genome that quickly and efficiently beat the level after 34 generations, primarily by repeatedly spin jumping over obstacles and enemies. However, when the genome was applied to other levels, it had become so specialized to that level that it required an almost complete retraining to beat them. This is an example of the classic overfitting problem in machine learning. He also was able to train a genome that

could beat Super Mario Brothers World 1-1 after 95 generations; however, the same overfitting issue could be seen when the genome was applied to other levels. This project aims to improve on this application by training an agent that has a more general understanding of the behaviours such that it is able to avoid the overfitting issue and beat any level in SMB.

## 2.3 Beating Mario With Statistics

Automating the process of playing video games is a common application for reinforcement learning [36]. It provides a great framework showing the success or failure of the agent and it also provides many methods to map states and actions to a particular value of the result [26]. For a reinforcement learning system to work, some form of a fitness measure must be assigned to the results of an action as this allows the system to denote beneficial actions from detrimental actions. This way an agent can learn to select the actions that lead to the desired goal. An interesting and amusing approach to the definition of fitness function comes from Dr. Tom Murphy's paper *The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel...after that it gets a little tricky.* Although written in a humorous tone and submitted to a conference, SIGBOVIK, known for satirical and joke academic research, the author assures us "This work is 100% real" and he has the source code to prove it. This approach attempts to order the byte values of RAM locations in an attempt to sort the most beneficial values to have Mario progress through a level. The most beneficial values are sorted to the front of the list and then the agent attempts to sequentially maximize those values in order to guide Mario from the start of a level to the goal flag at the end [21].

Because the NES is a computer that uses an 8 bit, also known as a byte, processor and has access to 2048 bytes of RAM with which the entire game must be loaded into [21], this approach effectively uses the current state of the RAM as a representation of the state for the reinforcement learning algorithm. Any input action by the player changes any number of values within the RAM, like the level location ($x_l$ = byte 0x006D) and screen location ($x_s$ = byte 0x0068) of Mario. Conveniently, Mario starts a level with a $x_l = 0$ and $x_s = 0$ and the goal flag is generally located at $x_l = 11$ and $x_s = 200$, so if the level was a flat, straight line, the algorithm would simply have to maximize those two values. Unfortunately, no SMB levels are arranged like this, opting to have a more challenging assortment of obstacles and enemies to avoid, so the algorithm must learn to maximize other values to reach the goal. To do this, the algorithm looks at input action sequences from a human player and learns how to sort the RAM locations based on how those input actions affect the memory state. In this way, it is able to learn the order of byte values it needs to maximize in order to guide Mario to the goal. While the system in this project doesn't attempt this lexicographical ordering,

it does build off of this concept by using RAM locations as input values to a neural network with the goal of learning how those values inform the correct inputs an agent must select in order to complete the current level.

# Chapter 3

# Deep Learning

## 3.1 Introduction

The training of deep neural networks, also known as deep learning, is probably the most commonly seen form of machine learning algorithms. ChatGPT, Midjourney, and DALL-E are prime examples of the power of deep learning at generating realistic text and images. However, beyond these flashy, popular uses deep learning has the potential to become a powerful and important tool for many industries. An example of this in the medical field would be the ability to determine the probability that a patient has Parkinson's disease due to variations in a patients vocal features.[1] Deep neural networks have been shown to be able to solve extremely complex problems, given they have four main resources:

- A large, robust dataset consisting of tens of thousands of instances that offer a diverse and complete (as possible) representation of the potential individuals or outcomes of a problem.
- Enough time for the neural network to exhaustively analyze each training instance and tune its weights to make accurate predictions.
- Copious amounts of energy and computational power with which to perform the training.
- Sufficient memory to store and load the gigantic tables of weights in order to make predictions after a model has be properly tuned.

The first two items are a huge limiting factor for pretty much all applications of both deep learning and also neuroevolution; however, deep learning applications have become extremely effective at classification and other supervised learning tasks now that companies like Google and Amazon have been able to amass titanic server facilities to host the training and execution of these deep learning models. This chapter will cover the basics of deep neural networks and some of the limitations of those systems, with the goal of showing how, despite

their success in some areas of machine learning and AI development, there are other areas and problems that different approaches, like neuroevolution, can outperform them in and generally be better suited at solving these problems.

## 3.2 Principals of Deep Learning

### 3.2.1 Neural Network Basics

The basic function of modern neural network architectures follows that as inputs are propagated through a network, certain neurons will activate based on the value of this input. The factor that determines if a neuron will activate is the sum of all inputs ($x$) into the neuron multiplied by the weight ($W_i$) of each input has plus some bias ($B$) amount. This sum is then input into an activation function ($\phi$), whose value, based on the input, results in the neuron being activated or deactivated.

$$A = \phi(\sum_{i=1}^{n} W_i * x_i + B) \tag{3.1}$$

Because of this, the weight of an input determines how much it contributes to the firing of a neuron, with a low weight making the neuron less likely to fire and a high weight making the neuron more likely to fire on a given input. Once the inputs have propagated through the network, there will be a chain of activated neurons leading from some of the input neurons to some of the output neurons. These output neurons now have an activation value that can be used to determine actions taken or what prediction can be made.

The exact use of these activation values differs greatly between applications of the neural network. For example, a binary classification problem like determining if an image is of a cat or a dog, the neural network will have two output neurons:

- One neuron representing the chance of a positive identification (the input **is** a cat).
- Another neuron representing the chance of a negative identification (the input **is not** a cat).

In this case, determining the neural network's prediction for an input image can be done by checking which neuron has the highest activation value and selecting the class that neuron represents, either a positive or a negative identification. What's more, through a process known as backpropagation, the neural network can now use the error rate between it's two output neurons to refine the weights of it's neuron input connections and allow the network to make more accurate predictions in the future.

### 3.2.2 Training Montage

Deep learning is the process of training large neural networks, which contain many hidden layers and often millions of neuron connections (Figure 3.1), to be able to accomplish a task or make predictions. This is most commonly achieved by fine tuning the weights within the network so that the only neuron chains that will activate will be the ones that lead

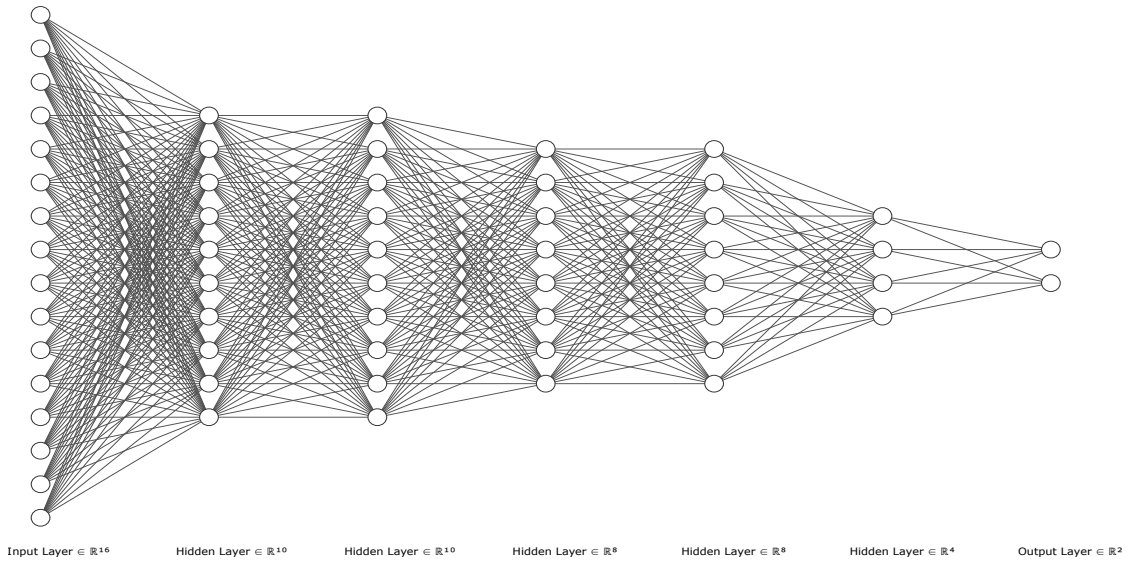



Figure 3.1: Simple deep neural network

to the correct output neuron(s). For deep learning, and also conventional neuroevolution algroithms, this all occurs over a topologically fixed neural network. This has the major advantage of allowing the training to focus solely on optimizing the weights between neurons and can give the networks the ability to make extremely precise predictions for complex problems; however, this also directly leads to an exacerbation of the requirements listed above, most notably the space, time, and power requirements. Because these networks are defined in an ad hoc fashion, they are not topologically optimized. Due to this, there can be many groups of neurons that have a detrimental effect on the accuracy of the network. In order to minimize the effects of these detrimental neurons, large chunks of training time is needed just to prevent them from being activated.

As mentioned above, the most popular method of refining the weights of a neural network is a technique known as backpropagation. First identified by Frank Rosenblatt as "back-propagating error correction" in the book *Principals of Neurodynamics* [23], back propagation is the process of working backwards through the layers of a neural network and refining the weights as it goes. This is accomplished by calculating the error gradient for a given neuron's weight and then adjusting the weight such that the error gradient leads to a reduction in the total error for the output of the network. Originally, the calculation of the error gradient was the backpropagation algorithm; however, as it has become almost ubiquitously used for training topologically fixed deep neural networks, backpropagation has become an umbrella term for numerous algorithms that also use the error gradient to tune the weights. For example, stochastic gradient descent (SGD) is a backpropagation method that takes the error gradient calculations and then applies the Newton-Raphson method for minimizing functions to determine the proper weight adjustment to minimize the output error of the network.

## 3.3 Cons of Backpropagation

Even though backpropagation has shown immense power to optimise a neural network to a particular problem, it is not without certain drawbacks. Apart from needing a long time to converge to a particular solution, the specific implementations of it can also have issues. SGD is the most popular implementation of backpropagation and it has a big problem where it can fail to find the true minimum value of the error function, also known as the global minimum, and instead become stuck in a local minimum. This happens because the nature of gradient descent is to continually minimize a value and any time it detects an increase in that value, it corrects itself by readjusting to a lower value. This can prevent it from moving over a bump in the error curve and become stuck with a value that is minimized compared to the values around it, but is not actually the minimum value that the error function can reach (Figure 3.2).

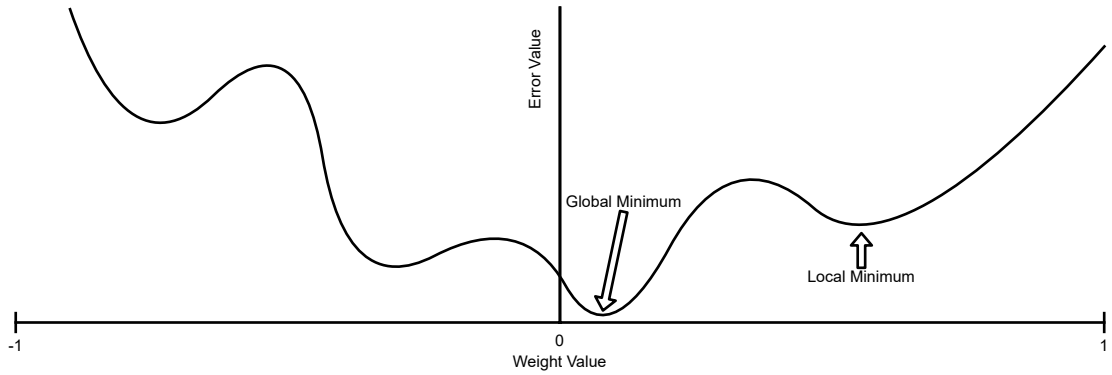



Figure 3.2: Error function with one global minimum and multiple local minimums

Over the years there have been various solutions proposed that address this issue and have been shown to alleviate the effects of this; however, this problem is still one that must be accounted for when designing a neural network. In addition to this issue, there are two other major problems that have to be addressed when designing a backpropagation algorithm: vanishing and exploding gradients.

### 3.3.1 Vanishing Gradients

In a nutshell, a vanishing gradient is, as the error gradient is passed backwards through the neural network, it eventually becomes so small that the weights in the shallow layers of the network are no longer updated. As a reminder, backpropagation relies on calculating the error gradient of the network output, which is then used to tune the weights of a neural network. If these gradient values become so small that the backpropagation algorithm is unable to actually update the weights, the neural network stops learning and essentially becomes frozen.

### 3.3.2 Exploding Gradients

The opposite of the previous section can also occur when training a neural network. Instead of the gradients vanishing into nothing, as the error gradients move back through the network they can become exponentially bigger with each step. This makes the gradient descent algorithm adjust the weights at the maximum step size at every iteration. The results of this lead to the error value wildly fluctuating back and forth over the minimum value and can even cause the network to diverge into infinity and, ultimately, fail to learn to solve a problem.

## 3.4 No Free Lunch

The drawbacks illustrated for the backpropagation algorithm only further prove that continued research into different methods of optimizing neural networks is necessary. Backpropagation is clearly an effective method of optimizing a neural network for a specific set of problems, specifically classification based on labeled datasets; however, the No Free Lunch theorem shows that there will always be another set of problems that it does not perform well on. Some of these applications are neural networks that require specialized, non-differentiable activation functions or classification tasks on unlabeled datasets which require unsupervised learning techniques [18]. This project will focus on the creation of a neural network agent that can solve a behavioural problem that requires a reinforcement learning algorithm. This agent must be able to learn the complex decision process that selects the optimal behaviour for a scenario. The details of this application will be discussed in **Chapter 5**.

# Chapter 4

# NeuroEvolution of Augmenting Topologies

## 4.1 Life Will Find A Way

*NeuroEvolution of Augmenting Topologies*, or NEAT, is a neuroevolution system proposed by Kennity O. Stanley and Risto Miikkulainen that modifies not only the connection weights within a neural network, but also the topology, or structure, of the neural networks themselves [30]. While NEAT itself is a novel proposition for the architecture of a neuroevo-

lution system, the broader concept of neuroevolution to optimize a neural network is not. There has been much research into the application of genetic algorithms to optimize neural networks dating back to the 90s. Much of this research has fallen into two main categories:

1. Neuroevolution of the connection weights withing a fixed topology network.

2. The simultaneous neuroevolution of both the weights and topology of a network.

Much of what is considered conventional neuroevolution falls under the optimization of fixed topology networks. These networks generally consist of a layer of input nodes, a hidden layer of nodes, and then a layer of output nodes [30]. Each layer of nodes is fully connected to the next layer's nodes, meaning each input node is connected to each hidden node and then each hidden node is connected to each output node. The number of hidden nodes is generally determined by trail and error by the network designers. These networks are then optimized, or evolved, by mating the high performing networks with each other and mutating the weights of a network. [30] Before we discuss the NEAT architecture, it is important to give some historical context to the design choices NEAT uses by first describing the neuroevolution architecture it was born from, *Topology and Weight Evolving Artificial Neural Networks*, or TWEANNs.

### 4.1.1 TWEANNs

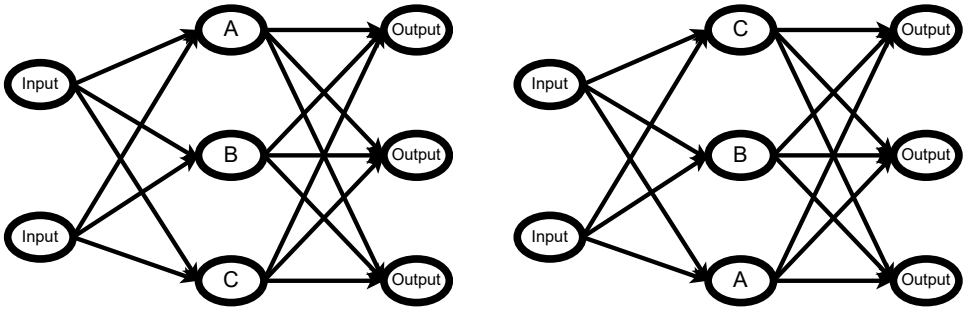
Much of the early research into neuroevolution has been focused on the development of TWEANNs. For these systems, the issue of how to encode genetic information within the network has been a subject of much debate. The two methods of encoding genetic information in TWEANNs are either a direct encoding or an indirect encoding. Direct encoding methods are generally simpler to implement and are the most commonly used in TWEANNs [30]. This method entails specifying exactly how many nodes and connections exist for a given genome. Alternatively, an indirect encoding scheme requires the creation of rules about how a connection or node will exist in a genome. This allows a genome's structure to be inferred from those rules and can result in a much more truncated representation of a given genome; however, the creation of these rules is a complex task that requires careful crafting in order to create viable network structures from a genome.

Direct encoding schemes often use an explicit graph structure to represent the network that is generated from a genome [30]. This has distinct advantages over other encoding schemes because it allows network graphs to be broken into smaller sub-graphs known as *functional units*. These functional units can then be swapped between pairs of genomes during mating; unfortunately, the efficacy of these matings cannot be guaranteed outside of trial and error evaluation during evolution of the genomes [30].

This has lead to some TWEANN designers omitting the process of mating during the evolution of a population of genomes. [34] and [3] have demonstrated that, although they have removed the concept of mating from the design of their algorithms, they have still been able to create successful TWEANNs with only topological and connection weight mutations. Additionally, TWEANNs have three major issues that the NEAT architecture succinctly solves. These problems are presented below and then the solutions devised by the architects of the NEAT algorithm will be discussed in detail later in the chapter.

### 4.1.2 A Genome by Any Other Name

When attempting to evolve a neural network structure that solves a particular problem, there can be many representations of the network that achieve the intended goal. [15] and [25] have coined this the *Competing Conventions Problem*, or the *Permutations Problem*. It is not a problem in and of itself that multiple network/genomic structures can solve a particular challenge; however, if one wishes to use crossover to mate genomes during evolution, these structural, or topological, differences can present unique challenges to creating viable offspring. At worst, these topological differences can lead to unusable, broken genomes; however, most often these topological differences will lead to the offspring losing vital information from one or both parents genome. This can result in genomes performing worse over time and, ultimately, failing to solve the problem they are tasked to. For example, lets take two genomes, $A$ and $B$ (Figure 4.1), and create a child genome from via crossover mating.




(a) Genome A (b) Genome B

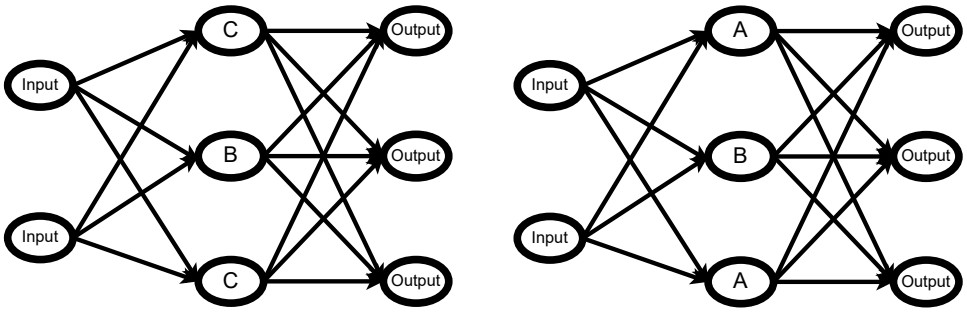
Figure 4.1: Genomes to be mated via crossover

Both of these genomes ultimately calculate the same value because, from the genome's perspective, they both contain the same connections and nodes; however, from a topological perspective, they are both only two of the $3! = 6$ different possible permutations that this genome can take. It is important to note that both node A and C plus their connections are different, distinct structures that happen to exist in the same topological location. Therefore, when these two genomes are crossed over during mating, the resulting child genomes will lose a third of the information of the parent genomes due to either node A being placed where node C is or vice versa. (Figure 4.2).

Over the years, TWEANN designers have proposed many solutions to counter this issue; however, are unable to prove these solutions completely eliminate the *Competing Conventions* problem. The NEAT architecture introduces a novel concept to avoid this issue entirely by implementing a historical tracking system that remembers when during the evolution of a genome a particular connection was created. The mechanism of this will be explained in the **NEAT Architecture** section.

### 4.1.3 Initial Populations

The initial population of a TWEANN architecture generally starts with an assortment of randomly generated topologies. While this ensures a diverse set of genomes within the population, it creates some problems for the optimization of them. One problem that direct




(a) Node A's information lost (b) Node C's information lost

Figure 4.2: Information loss due to the *Competing Permutations* issue during crossover

encoding TWEANN's encounter is the potential for the creation of genomes that don't have a path of connections from the input nodes to the output nodes. These networks must then be weeded out during the training of the genomes which results in wasted time that could be better spent optimizing valid genomes from the start.

Another problem with starting with random topologies is that these network structures have been created without regard to the fitness of them. This means there is a high likelihood that these genomes have suboptimal or erroneous nodes and connections within them. Unfortunately, the process of mutations and crossover mating has very little inherent drive to remove these extraneous structures. This leads to the TWEANN creating large, bloated networks and, as long as the networks have a high enough fitness, they will persist throughout training. Some TWEANNS, [35], have proposed adding a penalty for network size to the fitness calculation; however, these fitness penalties must be carefully crafted to not influence the evolution of genomes in unintended directions [30].

### 4.1.4 Protecting Innovation

When a genome is mutated to add a new node or connection, this mutation quite often lowers the fitness of the resulting genome. This is due to the new structure starting with a randomized weight that has not had a chance to be optimized. Without some method to protect this new structure, it is not going to survive in the population long enough to evolve into something beneficial (unless it happens to be immediately effective, which is extremely unlikely). This necessitates a system that protects these new structures.

[3] proposes a system of protecting new mutations by adding them to the network without being connected to it. This can give the new structure extra time to optimize its connections and weights without dragging the fitness of the overall network down. Eventually, a mutation will connect this structure to the overall network, which may lead to increased performance of it. However, this connection is not guaranteed to be created; furthermore, when connected, this new structure may still decrease the fitness of the network. Ultimately, this results in, at best, wasted training time that could have been better spent developing a more performant network and, at worst, eliminating an otherwise well performing network from the population.

## 4.2 NEAT Architecture

Similar to some of the best performing TWEANNs, NEAT genomes use direct encoding with a linear representation of connection genes that get translated into a graph structure network, or phenotype, for training and evaluation. The linear representation of genomes facilitates the easy incorporation of mating into its evolution because two genomes can be lined up and checked for compatibility. Stanley and Miikkulainen also show that NEAT, while still performing very well without mating, adding mating significantly speeds up the convergence of the algorithm to the solution. In the same fashion as TWEANNs, NEAT employs mutations that change the weights of connections and the topological structure of genomes. The mechanics of these mutations will be discussed in detail in **Chapter 5** [30]. The rest of this chapter is dedicated to how NEAT solves the above issues that occur in TWEANN architectures with novel architectural improvements.

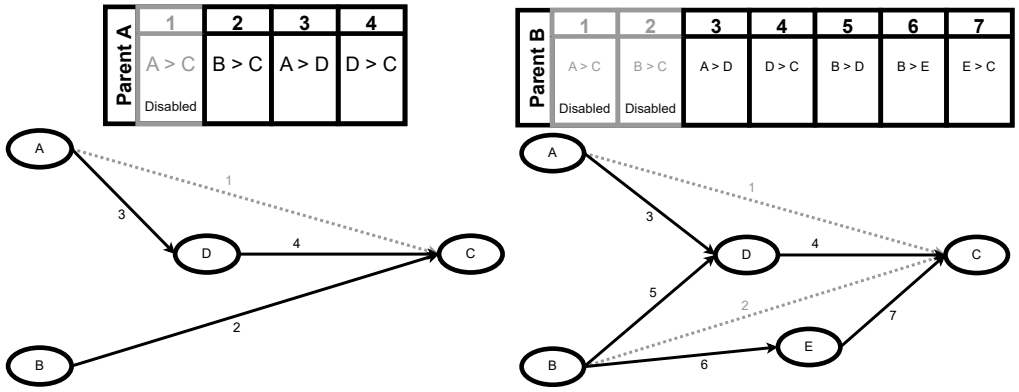
### 4.2.1 Historical Markings

The first issue with a TWEAN architecture deals with the *Competing Conventions* problem. As a reminder, this issue is where multiple similar genome structures can all solve the same problem but ultimately be unable to produce offspring that retains all the information learned by their parents due to subtle topological differences. This problem exists for not only neural network evolution but also for the evolution of life on this planet. An organism's physical makeup is determined, in large part, by their genetic code, and this physical structure is very important to the success, or failure, of said organism. As an example, a predator fish without fins specialized for speed and agility would be very unlikely to be able to find food and, thus, be very unlikely to find a mate. Eventually, it would fall out of the population in favor of fish who's fins are optimized for hunting. The process in which an organism's genome becomes more complex over time is known as *Gene Amplification* [8, 32]. However, this process of amplification cannot allow genes to be inserted randomly without regard to order, otherwise the *Competing Conventions* problem would result in offspring missing vital genes for survival [30]. This means the location of inserted genes must be structured in such a way that facilitates optimal sexual reproduction.

In nature, some organisms have solved this problem with specialized proteins that line up corresponding genetic traits during reproduction in a process called *synapsis* [22, 27]. For neural network reproduction, this structural analysis is not easily done. Therefore, NEAT proposes an elegant solution by tracking the historical origin of a connection gene by assigning it an *Innovation Number* when it first evolved. This way, if another genome in the population evolves this same connection gene in the same or a future generation, it is assigned the same *Innovation Number* as the original incarnation of the connection gene. This ensures that each *Innovation Number* represents the exact same structure within a population, thereby facilitating sexual reproduction without fear of *Competing Conventions* causing a loss of genetic information in offspring.

During crossover mating, *Innovation Numbers* allow us to easily line up the connection genes that share an *Innovation Number* into groups of matching genes. Any genes that are not in both genomes are then classified as disjoint or excess genes, depending on whether their *Innovation Number* lies within or without the range of *Innovation Numbers* that the

genome contains. The resulting child genome inherits all the matching genes of its parents, with the weights being randomly selected from either parent. Then the child genome inherits all the disjoint and excess genes from the most fit parent that the gene exists in. Additionally, if a gene is disabled in one parent but enabled in another parent, that gene will randomly be either disabled or enabled in the child genome.




(a) Parent A Genome (b) Parent B Genome

Figure 4.3: Parent Genomes Used for Crossover Mating

Ultimately, this process of assigning *Innovation Numbers* to track the historical genesis of connection genes efficiently prevents the *Competing Conventions* problem without having to perform complex topological analysis of a genome. Because these connection genes can now be lined up during crossover mating (Figure 4.4), we can ensure the child will have all the structural information of both parents. This creates a robust pool of genomes that can continually increase in complexity during the course of evolution toward solving a problem.

### 4.2.2 Speciation is the Key

The next big issue faced when training a TWEANN is how to protect newly mutated structures from being eliminated from the population before they have a chance to optimize. Because adding a new node or connection gene to a genome will often lower its overall performance, the genome can die off before the newly evolved structure has a chance to actually prove if its valuable. To solve this issue, NEAT introduces a concept called *Speciation* to the evolutionary process. This concept closely mimics the biological concept of niches and how a species of organisms will evolve to perform well within that niche. This allows organisms to only have to compete for resources with a smaller subset of a population, instead of the entire population. For example, take a hypothetical lake that has some fish and seaweed in it. Because the ultimate goal of these fish is to eat enough food to procreate, this creates two main niches for finding food, a herbivorous species of fish that eat seaweed and carnivorous species of fish that eat other fish. Because the carnivorous fish aren't trying to eat the seaweed, the herbivorous fish are only competing with themselves to eat enough seaweed to procreate. In this way, any structural changes to the mouths of the herbivorous species of fish will only affect its survivability, or fitness, in relation to other members of its species.

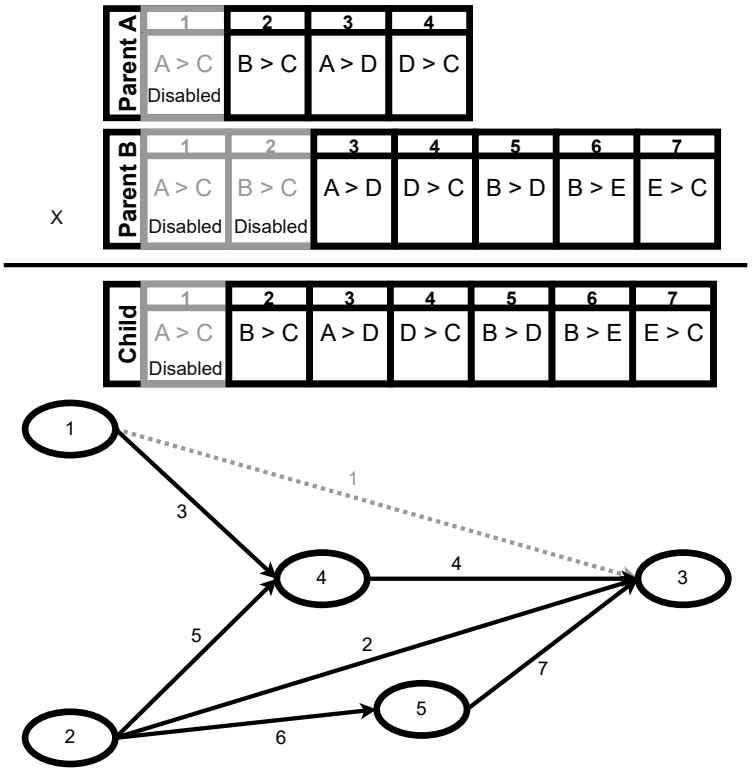



Figure 4.4: Child Genome Resulting from Crossover Mating of Parent Genomes (Figure 4.3)

*Speciation* in NEAT follows a similar path to the above hypothetical, wherein the survivability of a particular genomic structure is only assessed in relation to the other members of the species it is assigned to. This way, when a new connection gene or node is added to the structure through mutation, it is not immediately killed off by better performing genomes with unrelated genomic structures. The tracking of *Innovation Numbers* of connection genes makes speciating a genome an extremely simple process, as a comparison of the excess and disjoint genes provides a good measure of how related two genomes are. Genomes that have many disjoint and excess genes have different evolutionary paths, and are therefore not very compatible for fitness evaluation [30]. A simple linear combination of the excess ($E$) and disjoint ($D$) genes plus the average weight of matching genes ($\overline{W}$) gives us a compatibility difference ($\delta$) value, which we can use to determine if a given genome belongs to a particular species.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{4.1}$$

The hyperparameters $c_1$, $c_2$, and $c_3$ control the importance of each factor and $N$ normalizes the excess and disjoint by the number of genes in the larger genome. For this application of NEAT, the hyperparameters were set to $c_1 = 2$, $c_2 = 2$, and $c_3 = 0.4$ and ultimately a genome must have a $\delta < 1$ to be considered a member of a species. When a new genome is created via mating, it's $\delta$ value is calculated sequentially for all the current species within

the population. A genome is placed into the first species it qualifies for and if it doesn't qualify for any species, a new species is created and it is placed into that species. This way, no genome can ever exist in multiple species.

### 4.2.3 Minimally Effective Structures

As stated in the above section, TWEANN's start with a distribution of randomly generated topologies to ensure a diverse population to evolve from. This, as has been shown, can lead to an inefficient evolution process that wastes time evolving sub-optimal genomes. One of the main goals of NEAT is to create an efficient evolution process that limits the overall genomic structure search space [30]. This means searching for the optimal minimal structure during all steps of the evolution process, not just for the final genomes. The NEAT architecure proposed by Stanely and Miikkulainen starts with a *Minimal Structure* that consists of a network with zero hidden nodes and every input node connected to every output node. This forces the genomes to incrementally grow from topological mutations and only allows such mutations that improve the fitness of the genome to continue procreating. This efficiently minimizes the time searching through sub-optimal structures and focuses the evolution process into creating genomes with a minimally effective structure. Stanely and Miikkulainen show that the principal of evolving minimally effective topologies has distinct performance advantages [30]. They do this by comparing performance of the NEAT architecture against other neuroevolution algorithms using the Pole Balancing benchmark tests.

## 4.3 Pole Balancing Benchmarks

The Pole Balancing benchmarks tests are a commonly used scenario to benchmark reinforcement learning algorithms [13, 33]. For this task, an agent must balance two poles of different lengths on top of a cart as it moves from a starting location to a goal location. These benchmarks generally take two primary forms, pole balancing with velocity information and pole balancing without velocity information. These two forms show how well a reinforcement learning algorithm performs with a Markovian and Non-Markovian task, respectively. Because the task that provides velocity information can allow an algorithm to easily map changes in the velocity of a pole to the current state of the pole, it is considered a Markovian task due to the ability to create chain of actions that leads to a desired outcome. When velocity information is not available, however, it is impossible to map state changes in the pole to a specific action. Therefore, it is a Non-Markovian task because a clear chain of actions that leads to the desired outcome cannot be easily created. Instead, the agent must learn sets of behaviors that allow it to react appropriately, given the current state of the poles. In general, the ability to directly map state changes to actions makes Markovian tasks much simpler to solve than Non-Markivian tasks. Stanely and Miikkulainen [30] are

able to show that NEAT is able to perform over 2000% better than a conventional neuroevolution algorithm [24] and between 200% - 5% better than three other TWEANN algorithms [17, 13, 33].

# Chapter 5

# A NEAT Approach to Agent Evolution

## 5.1 Hardware and Software

Before explaining the Super Mario Evolution by the Augmentation of Topology (SMEAT), architecture, it is important to describe the hardware and software involved with running Super Mario Brothers and the agent learning to play the game. This section will provide a brief explanation of how Super Mario Brothers is played, how the Nintendo Entertainment System hardware runs the game, and how modern emulation technology allowed me to create an agent that learns to play the game. The goal of this is to give context to the architectural choices used to represent the game state and create a decision process that leads to a successful completion of a level in the game.

### 5.1.1 Super Mario Brothers

For this thesis project, Super Mario Brothers (SMB) for the Nintendo Entertainment System (NES) was selected as the environment for the agent to learn. SMB was released on the NES in 1985 and is often ranked in the top 20 greatest games of all time [20, 14]. The core gameplay of SMB tasks the player with navigating Mario through and over a series of obstacles to get to the goal flag at the end of the level. SMB has a total of eight worlds that each contain four levels. They are denoted in the format of "world-level", meaning World 3-2 is the second level in World 3. The first three levels in a world are depicted as outside levels that contain a varied mix of obstacles and enemies. The last level in each world is colloquially known as the "End Castle" and takes place, surprisingly, inside of a castle environment. The castle levels generally only have fireballs and Bowser as enemies and most of the challenges are difficult jumps over pitfalls while dodging the fireballs.

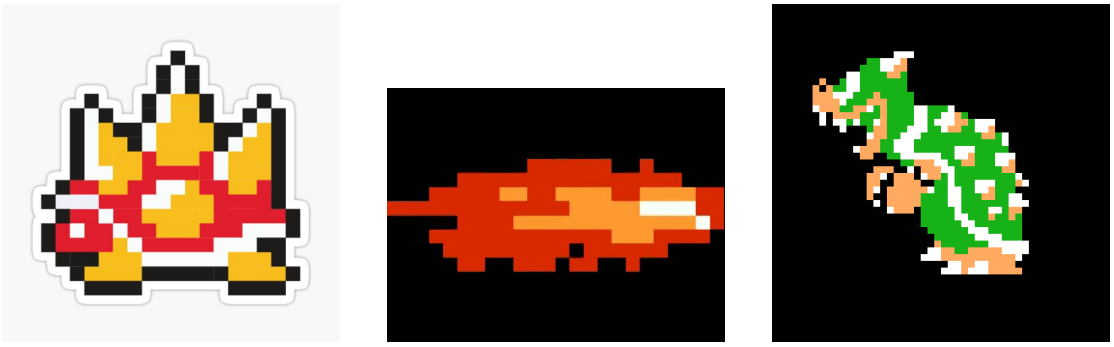
Each level begins with the player on the left most end of the level and the goal flag is at the right most end of the level. The player's primary means of navigation are moving left and right, corresponding to the left and right button on the Directional Pad (D-pad), while also having the ability to jump by pressing the A button and run by holding down

the B button. In addition, the player is able to duck down by pressing the down button on the D-pad and this also serves as the method of entering pipes, which generally lead to secret areas in the game. Occasionally, the player is able to climb vines to secret areas by pressing up on the D-pad; however, these secret area's often only contain coins and powerups and are not required to complete a level. The two notable exceptions to this are the secret warp zones, which contain warp pipes that allow Mario to skip to later levels of the game. Due to the hardware limitations of the time, the player is only able to move the screen to the right and once something has moved past the leftmost side of the screen, the player is unable to backtrack to it. This creates a distinctly linear gameplay that greatly simplifies the behaviors that an agent has to learn to successfully play the game.

Also featured in levels, scattered amongst the blocks and pitfalls that Mario must avoid, are a wide variety of enemies. A large number of these enemies can be stomped by landing on their head, like Goombas (Figure 5.1a), Koopas (Figure 5.1b), and Cheep-Cheeps (Figure 5.1c); however, there are also quite a few enemies that Mario must avoid touching at all, like Spinys (Figure 5.2a), fireballs (Figure 5.2b), and Bowser (Figure 5.2c). The gameplay is often praised for having tight controls and captures the essence of an easy to learn but hard to master type of game.

(a) Goomba (b) Koopa Troopa (c) Cheep-Cheep

Figure 5.1: Stompable Enemies



(a) Spiny (b) Fireball (c) Bowser

Figure 5.2: Unstompable Enemies

### 5.1.2 Nintendo Entertainment System

The Nintendo Entertainment System, or NES, was released in the USA in 1985 and quickly became on of the best selling consoles of its time. The NES is often credited with the revival of the USA gaming industry after the 1983 crash [6]. It used an 8 bit processor developed by Ricoh that ran with a clock speed between 1.66 MHz and 1.76 Mhz and had access to 2 KB of on board RAM to run any given game [2, 7]. The games were released on individual cartridges that were loaded into the console via an access port on the front. This design choice was meant to resemble how a video cassette was loaded into a VHS player, with the goal of making the console feel more like an entertainment device instead of a computer, which may have turned off less tech savvy consumers [7].

Because the NES had limited RAM available to store game visuals, developers created sets of tiles to use as visual assets which could then be referenced and redrawn many times while still taking up minimal space. This allowed them to create complex level structures by arranging the tiles in a grid pattern (Figure 5.3).

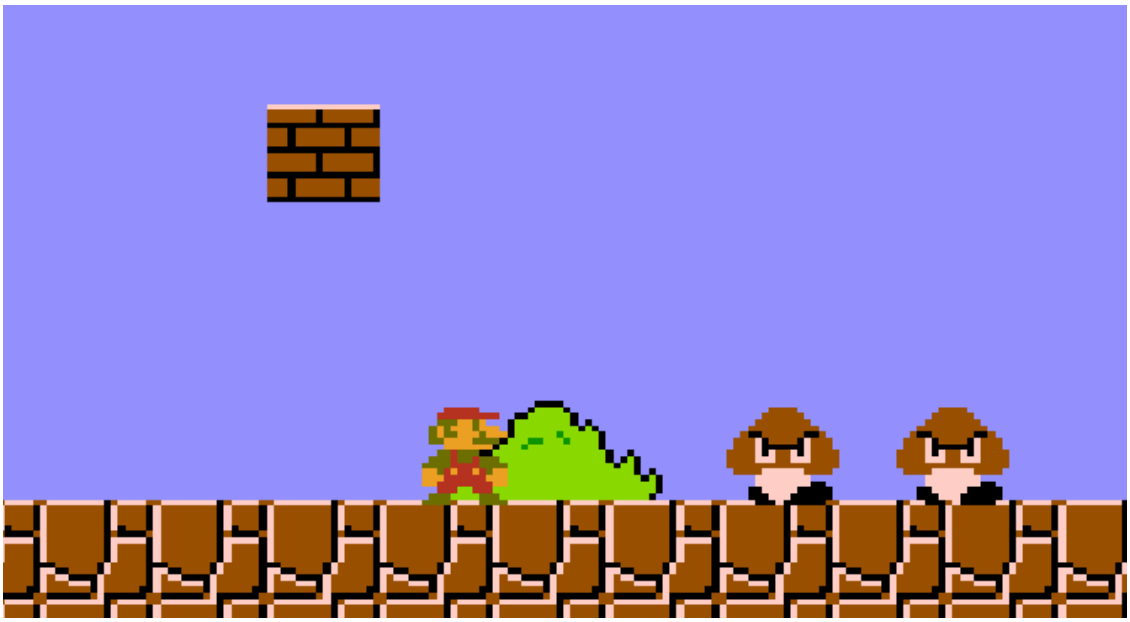


Figure 5.3: Sample of how tilesets were used to create game visuals

### 5.1.3 BizHawk

Retro games hold a special place in the hearts of many gamers today. It hearkens back to a simpler time before widespread internet connectivity created the dreaded day one patches, expensive downloadable content, and lootbox bonanzas that plague many AAA game titles today. Retro games show us that all you really need in a game is an engaging game mechanic and tight, responsive controls to create a truly enjoyable gaming experience. Unfortunately, hardware does not last forever and many consoles that actually run these games have been

lost to the ravages of time. To this end, people have created a plethora of emulators that are able to leverage the power of modern computing to mimic the functionality of these older consoles.

Because these games have been around for a long time, people have become quite adept at playing them. So much so in fact that it has spawned a new way to play these games known as *Speedrunning*. Speedrunning tasks the players with not just beating a game, but trying to beat the game as fast as humanly possible. For a game like SMB, it has been refined to such a level that timing a speedrun is counted in frames, not minutes or seconds. Human players train themselves to input button combinations at specific frames of gameplay (known as a frame perfect trick) in order to leverage glitches and bugs that allow them to shave off a frame or two from their total time. At this point the current world record for SMB is basically unbeatable by a human due to this extreme optimization of gameplay. However, the human spirit is not defeated by such simple obstacles and in recent years a new form of speedrunning had emerged known as *Tool Assisted Speedruns*, or TAS. A TAS takes the current extreme optimization of frame perfect tricks and leverages the power of computers to allow these tricks to be fed to an emulator every single frame of the game. This allows speedrunners to record the input sequences of frame perfect tricks and then string them together to achieve mind blowing feats that no human player could hope to achieve. For example, there are TAS runs of Super Mario Brothers 3 in which Mario only touches the ground at the start of a level and then explots a series of frame perfect glitches to remain in the air for the rest of the level.

The most popular emulator for these TAS runs of games is BizHawk. Most emulators simply mimic the hardware for a particular console and allow a user to play a game on it. BizHawk takes this to the next level by also incorporating an LUA interpreter into the core of the system. This allows a developer to inject custom code that runs in unison with the emulated game, allowing for an endless variety of customization of the games and gameplay within them. This project leverages this functionally to run a system that constructs neural networks and trains them to play the game without any human input at all.

## 5.2 Super Mario Evolution by the Augmentation of Topology

Super Mario Evolution by the Augmentation of Topology (SMEAT) is an improvement on the original MarI/O NEAT implementation in three distinct ways. The goal of these improvements is to create genomes that have a more general understanding of how to play a SMB level effectively and more easily adapt to new challenges it hasn't seen in training. The first of the improvements is the creation of a dataset of *Save States* throughout various levels of SMB, which I have named *Mario Moments*. A save state is simply a snapshot of the NES RAM state at a specific time that can be reloaded in the future. These will be discussed in greater detail in the **Mario Moments** section of this chapter. The second improvement deals with how these Mario Moments are applied during the training of a population of genomes. The application of the Mario Moments was developed as part of the research aspect of this project; therefore, the specifics of these application changes will be discussed

in the **Results** chapter of this paper. The third and final improvement that SMEAT employs is a refinement of the fitness calculations used in the original MarI/O implementation. The specific details of the nature and form of the refinements were part of the research aspect of this project and will be discussed in greater detail in the **Results** chapter of this paper. However, a basic discussion of how the fitness function works is included later in this chapter.

### 5.2.1 Mario Moments

As stated above, the goal of this project is to create a game playing agent with a broad, general understanding of how to effectively play a SMB level. MarI/O was able to provide a proof of concept that a NEAT algorithm can learn to play **a** level of SMB; however, as stated in the **Related Works** chapter, that implementation suffered from an overfitting issue that made the knowledge gained from World 1-1 less transferrable to other SMB levels. To that end, SMEAT uses two main tactics to sidestep the overfitting problem seen in previous applications and create a general understanding of gameplay. The first of which was to create a collection of random save states, called Mario Moments, in every level from World 1-1 to World 4-3. The total number of Mario Moments used in this experiment was 222. These Mario Moments represent the myriad of different challenges Mario has to overcome during gameplay. These challenges can range from simply jumping over a stack of blocks to progress the level or as complex as dodging a Koopa shell that is flying at Mario after it has bounced off an obstacle behind him. It is important to note World 2-2, which is an underwater level whose gameplay is rather distinct from the rest of the game, has not been included in the Mario Moments dataset. In this world the player makes forward progress by repeatedly pressing the A button to swim slightly forward and up. This means that the player is not required to stand on the walkable areas to make progress and can essentially jump without ever touching the ground. Because this departure from the standard gameplay controls would require the agent to learn a completely new set of behaviors, this level was omitted from the set of Mario Moments.

### 5.2.2 Moment Shuffling

The second tactic that was developed to prevent overfitting was to randomize a list of Mario Moments at the start of training and, eventually, for every new generation. The list of Mario Moments is shuffled via a Fisher-Yeats shuffling algorithm [9]. This algorithm ensures that each Mario Moment has an equal chance of being in any spot in the list of Mario Moments for any given training run or generation. The practice of reshuffling the dataset for every iteration is a staple of modern machine learning algorithms as it prevents the model from overfitting to the order in which the dataset is presented and instead promotes a more generalized understanding of the patterns in the data. It is important to note that the concept of *Moment Shuffling* was not present in every training run for this project. The exact nature and appearance of the shuffling algorithm will be discussed in the **Results** chapter of this paper.

### 5.2.3 Activation Function

The activation function used to determine if a given neuron is activated is a modified sigmoid function. In *Evolving Neural Networks through Augmenting Topologies* [30] Stanely and Miikkulainen propose a steepened sigmoid function of the form seen in Figure 5.4. Because SMEAT expects a value ranging from -1 to 1 to determine neuron activates, the sigmoid function has been modified to have a range of (-1, 1). A neuron is considered to be active if the activation value is greater than zero and not active if the value is zero or lower. It has the new form seen in Figure 5.5.

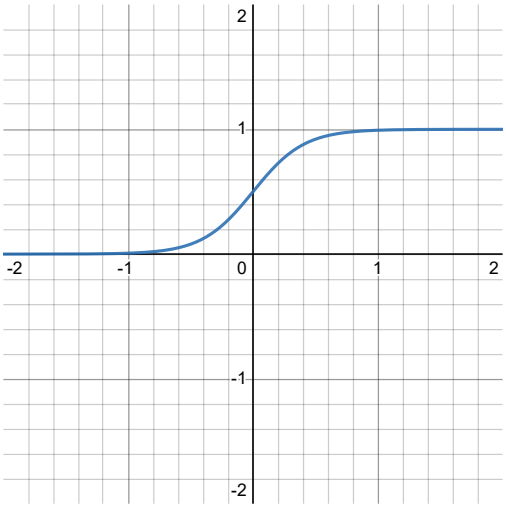



Figure 5.4: $\varphi(x) = \frac{1}{1+e^{-4.9x}}$

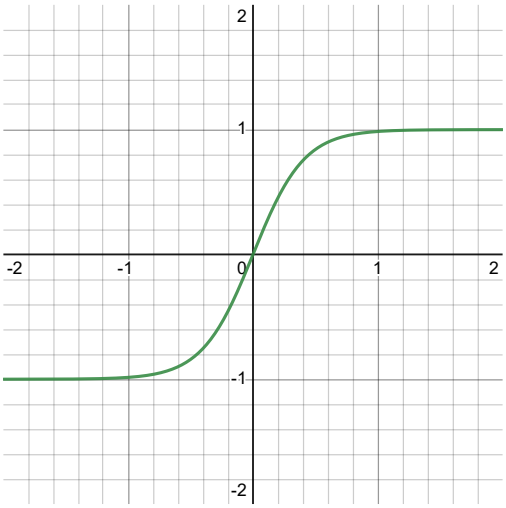



Figure 5.5: $\sigma(x) = \frac{2}{1+e^{-4.9x}} - 1$

## 5.3 Fitness Calculation

As with any reinforcement learning task, the definition of the fitness function is an extremely important step. Apart from structuring and applying the dataset of Mario Moments, refining the fitness function was a major part of this thesis research project. Defining a fitness function is an oddly philosophic undertaking as it requires you to ask the question "What does playing a Super Mario level effectively look like?" Should you reward stomping on enemies to promote the agent learning to beat enemies and continue making progress? Is successfully landing on solid ground after jumping something to be rewarded in order to better train the agent to jump over pitfalls? How valuable is maintaining a high speed when compared to the other challenges Mario faces in successfully navigating a level? Or are these issues immaterial to the ultimate goal of successfully reaching the goal flag and, therefore, should the agent only be rewarded when it actually reaches it? The exact from and the effects of the fitness functions used in each test run will be discussed in detail in the **Results** chapter of this paper; however, the basic themes that are shared between them are important to explicitly state now.

The primary means of a genome being more fit than another is how much forward progress it makes within a Mario Moment. That is to say, how many units right from the starting point ($M_{Start}$) in a Mario Moment did the genome get before dying or reaching the goal flag ($M_{end}$). On average a SMB level can have 3000 units between the start and finish of a level. Since a Mario Moment can start at any point in a level, the total fitness score for a Mario Moment can be between 1000 and 3000 units. To account for this difference, the basic fitness function takes the form of:

$$F_{base} = M_{end} - M_{start} \tag{5.1}$$

It is important to also note that in SMB everything that happens in a level is based on how much time has passed while playing that level. This means that enemies and moving obstacles begin their movements as soon as a player starts a level. This isn't formally stated anywhere, but is instead something learned by playing the game for a long enough period of time. As one gains proficiency in overcoming the platforming challenges within a level, they begin to see that getting though the level as fast as possible makes the level easier to beat. Because game time is measured in frames, minimizing the number of frames it takes to get to the end of a level is an important factor for the agent to learn. Since SMB runs at sixty frames per second and a level can take minutes to complete, penalizing the total number of frames can easily lead to negative fitness values. This means that only a fraction of the total frames can be used as a penalty, with $a$ determining the fractional amount of the total frames ($t$). This leads to the modified form of the above fitness function:

$$F_{mod} = F_{base} - (t * \frac{1}{a}) \tag{5.2}$$

### 5.3.1 Genomic Structure

The structure of the genomic networks in SMEAT use the geometry of the level and the locations of the enemies within the level as the input for the network. This input consists of a 13 block by 13 block square that has Mario located in the center, giving a 169 blocks of input plus one bias block, for a total of 170 input neurons into the network. Each block equates to one 16 pixel by 16 pixel tile that is drawn on the screen and is assigned a value based on the type of tile that block represents. The values are:

- 1 – Walkable Area
- 0 – Empty Space
- -1 – Enemy

Because NEAT genomes start with a minimal structure, the initial population of genomes contain only the 170 input neurons and the six output neurons; however, once generated, each genome in the initial population has a chance of a single node or connection mutation being applied to it. In a traditional NEAT architecture, each input neuron is connected to each output neuron when initally generated; however, for this application of the NEAT architecture this step has been omitted. If a fully connected input and output was generated, there would be many detrimental connections in the genome which would have to be removed with mutations. This would create the exact issue TWEANNs have with their initial populations and ultimately defeat the purpose of a NEAT architecture, as it was created to avoid this issue. I believe this alteration holds inline with the *Minimum Optimal Structure* doctrine that NEAT subscribes to.

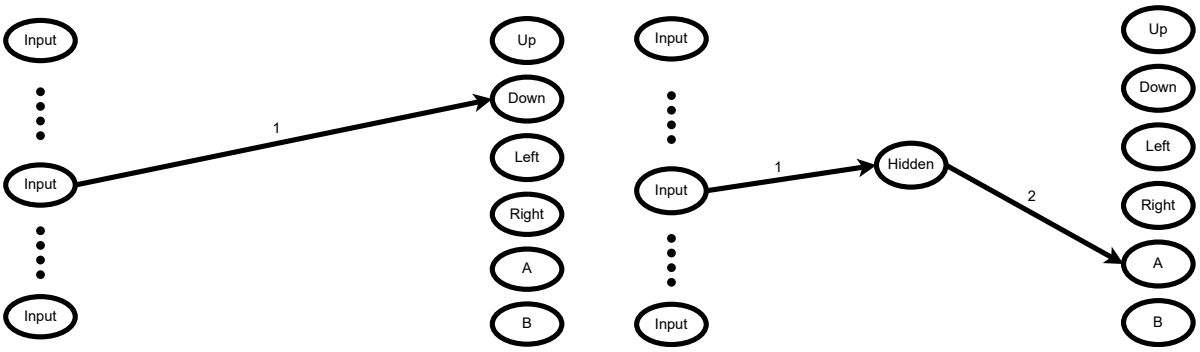
Much like a traditional neural network, the input node values are then forward propagated through any connections and hidden neurons contained within the network and eventually reach the six output nodes. These output nodes each represent the different buttons on the controller and, when activated, represent the agent pressing that button.

### 5.3.2 Genetic Population

Initially, SMEAT begins with a population size of 300 randomly generated minimal genomes. As stated above, these genomes all contain the 170 input nodes and the six output nodes.

At the start of every generation, each genome has a chance of acquiring a random mutation to either its connection weights or its topological structure. When a genome undergoes a connection weight mutation, each connection in the genome will have it's weight modified in one of two ways, Perturbation mutation or Cold Weight Reset mutation. A connection has a 90% chance it will undergo a Perturbation mutation, where the weight value will be increased or decreased by a random amount within a predefined step value ($s$). Otherwise the weight value is set to a new value randomly selected between -2 and 2. These mutations are expressed by the following formulae:

$$W_n = W_i + 2s(rand[0,1]) - s \tag{5.3}$$




(a) Single Connection Genome (b) Single Hidden Node Genome

Figure 5.6: Example Basic Genomes

$$W_n = 4(rand[0,1]) - 2 \tag{5.4}$$

When a genome undergoes a topological mutation, there are four different mutation types that can occur. The first is when a new neuron is inserted into a connection between two neurons, thereby increasing the non-linearity of the genome. When this mutation occurs, the original connection is disabled and two new connections are created. The new neuron now receives input from the input node from the original connection and its output is now sent to the output node of the original connection.

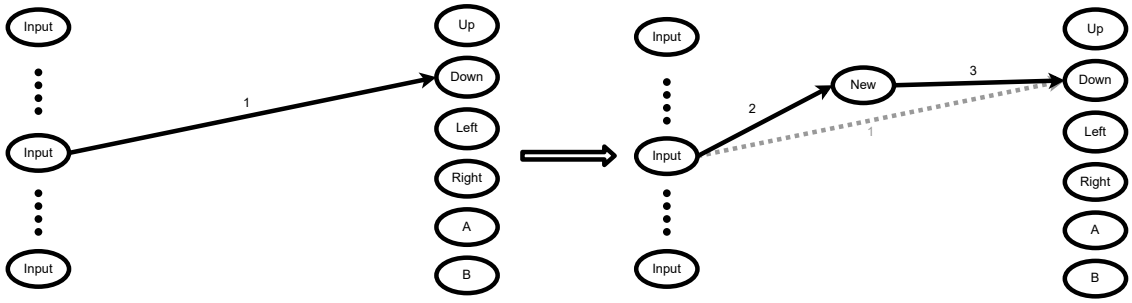



Figure 5.7: New Node Mutation

The second topological mutation type is a when a new connection is created between two previously unconnected neurons. When this mutation occurs, the first neuron's output is now also sent as input to the second neuron and this connection is then assigned a random weight between -2 and 2.

The third topological mutation type is when a connection between two neurons is disabled. This mutation type is one of the more difficult mutation types to account for when it is applied to a genome. It very possible that a mutation that disables the wrong connection can make a whole section of the genome's topology inaccessible and result in a broken genome. This is best illustrated by an example. Given the genome in Figure 5.9, there are multiple connections that can result in a broken genome.

Of all the connections available for disabling, if connection 5, 7, or 12 is disabled, the genome will have isolated neurons and become broken. Figure 5.10 is an example of connection 7 being disabled, resulting in over half the neurons and connections (highlighted in

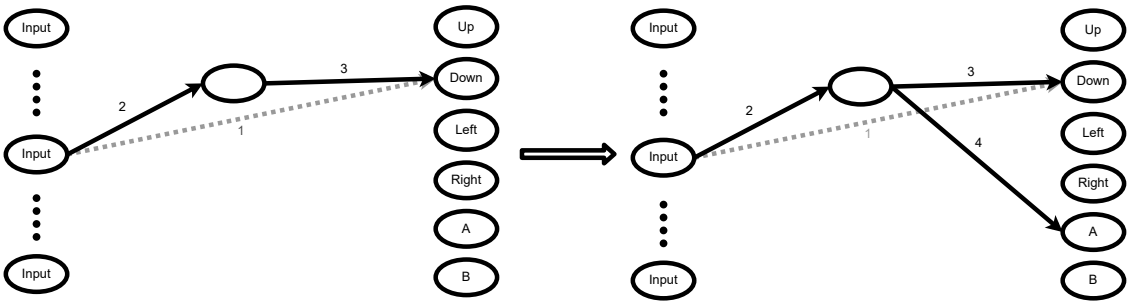



Figure 5.8: New Connection Mutation

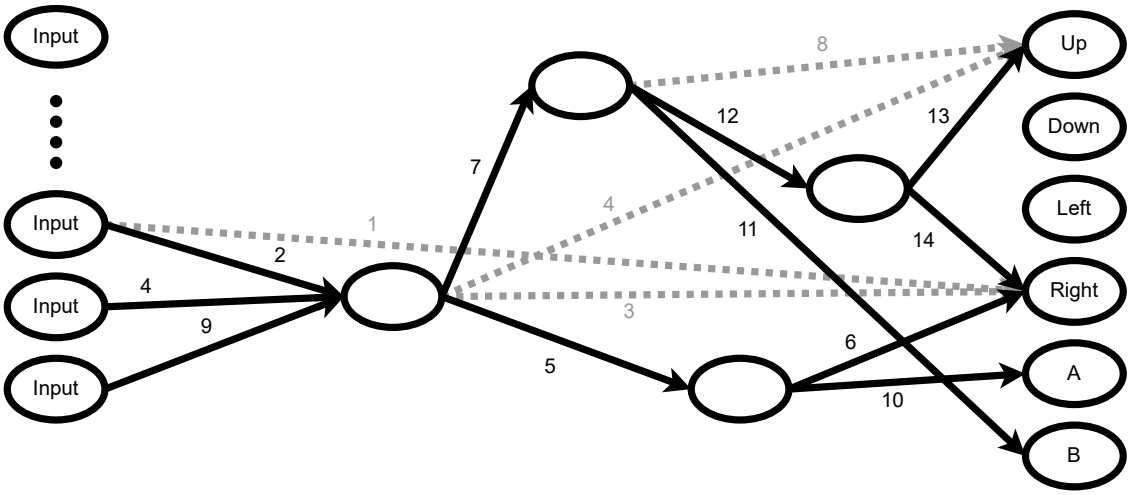



Figure 5.9: Complex genome to be mutated

red) of the genome becoming inaccessible. This violates the minimal structure doctrine that a NEAT algorithm maintains because multiple applications of a poorly selected connection deactivation mutation can result in a bloated genome that has irrelevant neurons.

In order to prevent this issue from happening, the algorithm must only select connections that fulfil one of two criteria. The first criteria is the connection's output neuron is a genome output neuron, and the second criteria is the connection's output neuron has multiple inputs into it. If these criteria are followed, then a valid connection for disabling could be connection 13, which would result in the mutated genome in Figure 5.11

The last topological mutation that can occur is when a previously deactivated connection between two neurons is reactivated. When a connection is reactivated by this mutation, the connection's output neuron uses the same connection weight from when it was last active (Figure 5.12).

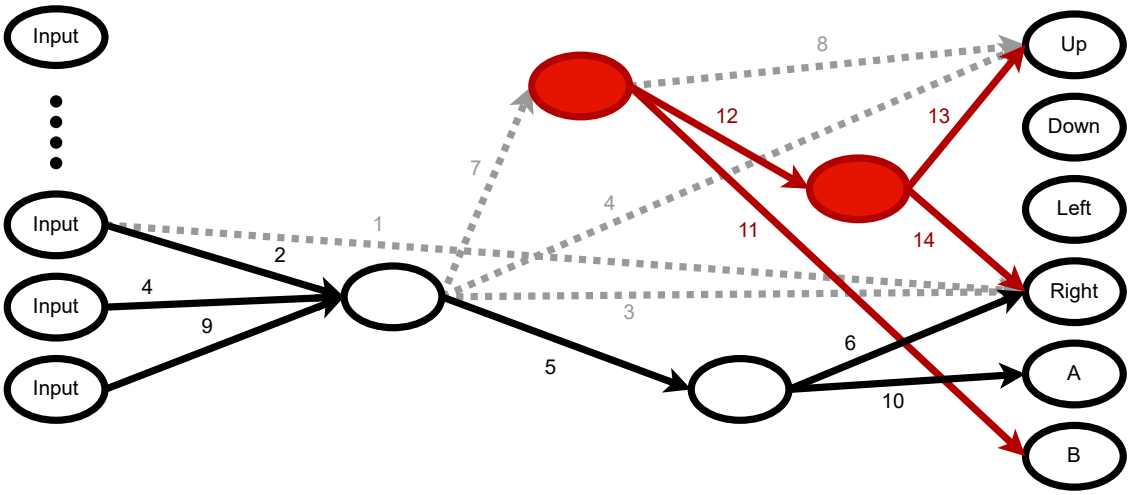



Figure 5.10: Broken genome due to mutation

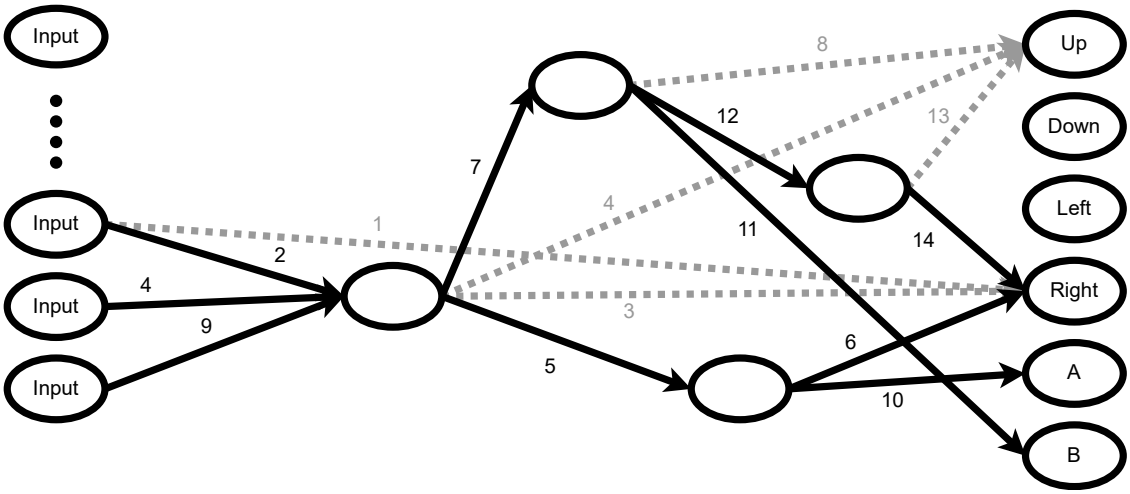



Figure 5.11: Valid disable connection mutation

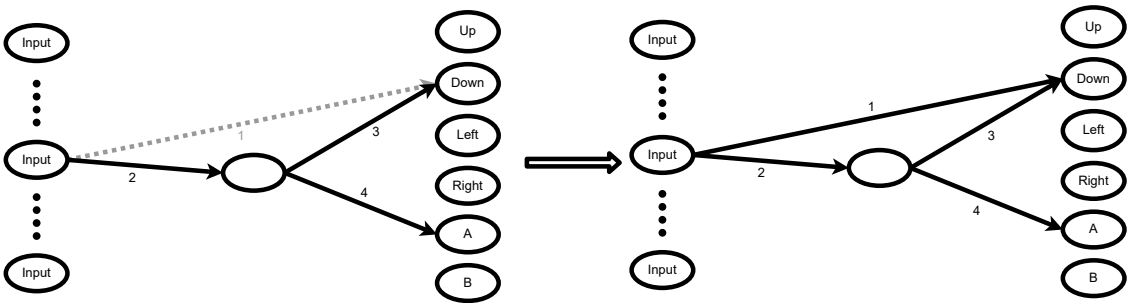



Figure 5.12: Enable connection mutation

# Chapter 6

# Results

This chapter will detail how the SMEAT algorithm development progressed and how the results of each training run affected the improvements/changes that were made in subsequent training runs. The primary development of SMEAT focused on building the Mario Moment dataset used in training, the application of the Mario Moment dataset, and the refinement of the fitness function. It will then go on to detail the outcomes of the training run and describe how the aforementioned changes affected the outcome.

## 6.1 Training Parameters

A training run refers to the generation of a population of genomes and the subsequent neuroevolution of those genomes, with the goal of teaching the genomes how to successfully play SMB. In each generation of the training run, every genome in the population is given one or more Mario Moments as the starting point and, once the genome's run concludes, it is scored via the particular fitness function used for that training run. A genome's run is generally concluded when one of three conditions is met:

- Mario dies by falling down a pitfall or hitting an enemy.
- Mario fails to make any forward progress for twenty frames plus a small bonus of $\frac{1}{4}$ of the frames generated in that run.
- Mario successfully touches the goal flag in a level.

This last item doesn't always end a run, however, because starting in training run #3, touching the goal flag will load the next Mario Moment in a list and the genome's run continues. In these cases, a genome's run is only ended when either of the first two conditions occurs. The only other caveat for this is training run #8, where the application of Mario Moments to the training run changes dramatically from all other training runs. This will be explained in greater detail in **Training Run #8**. Additionally, all training runs use the same mutation rates to determine when a mutation is added to a genome between generations:

- Connection Weight Mutation: 80%
  - Perturb Connection Weight Mutation: 90%
  - Cold Reset Connection Weight Mutation: 10%
- Enable Connection Mutation: 60%
- Disable Connection Mutation: 80%
- Add Connection Mutation: 90%
- Add Node Mutation: 50%

After a training run was concluded, I took the top performing genome, or champion, from each species of the population and had them attempt each level as a verification of how effective the training process was. It is important to note that for this verification test, I omitted World 2-2 and World 7-2, as those are both water levels and the genomes were not trained on any water levels during neuroevolution. To measure how effective the training sessions were, I tracked the total distance traveled in each level and if the genome was able to reach the goal flag for that level.

## 6.2 Training Run #1

### 6.2.1 Setup

This training run was used as a proof of concept test that the implementation of Mario Moments as the training dataset was possible. This run loaded a random Mario Moment at the start of each generation of training and then used that Moment as the input for each genome in the population. If a genome was able to reach the goal flag for that moment, it was awarded a 1,000 point bonus, $B$, and the next genome was started. The fitness function used was the base fitness function, with a frame penalty of $\frac{1}{2}$ of the total frames for the run:

$$F = M_{end} - M_{start} - (t * \frac{1}{2}) + B \tag{6.1}$$

### 6.2.2 Results

Because this run was meant for verification purposes, I only let it train for a week and the results were, by no means, successful at teaching the genomes how to play the game. However, it did verify that the algorithm was functioning at a base level and was ready to be refined and improved with further training runs. Table 6.1 displays the results of the champions of each species in the population.

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | |
|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 255 | No | 74 | No | 233 | No | 107 | No |
| World 1-2 | 126 | No | 51 | No | 127 | No | 18 | No |
| World 1-3 | 246 | No | 71 | No | 241 | No | 128 | No |
| World 1-4 | 509 | No | 156 | No | 191 | No | 81 | No |
| World 2-1 | 266 | No | 71 | No | 266 | No | 132 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 108 | No | 103 | No | 106 | No | 0 | No |
| World 2-4 | 206 | No | 206 | No | 208 | No | 167 | No |
| World 3-1 | 458 | No | 71 | No | 362 | No | 156 | No |
| World 3-2 | 149 | No | 71 | No | 143 | No | 11 | No |
| World 3-3 | 295 | No | 71 | No | 249 | No | 128 | No |
| World 3-4 | 239 | No | 246 | No | 238 | No | 235 | No |
| World 4-1 | 282 | No | 71 | No | 283 | No | 192 | No |
| World 4-2 | 159 | No | 18 | No | 168 | No | 97 | No |
| World 4-3 | 393 | No | 71 | No | 344 | No | 112 | No |
| World 4-4 | 94 | No | 60 | No | 94 | No | 0 | No |
| World 5-1 | 135 | No | 71 | No | 127 | No | 15 | No |
| World 5-2 | 170 | No | 78 | No | 186 | No | 138 | No |
| World 5-3 | 279 | No | 71 | No | 294 | No | 172 | No |
| World 5-4 | 267 | No | 247 | No | 258 | No | 163 | No |
| World 6-1 | 311 | No | 71 | No | 313 | No | 192 | No |
| World 6-2 | 250 | No | 72 | No | 251 | No | 250 | No |
| World 6-3 | 237 | No | 71 | No | 250 | No | 128 | No |
| World 6-4 | 505 | No | 156 | No | 191 | No | 81 | No |
| World 7-1 | 251 | No | 71 | No | 305 | No | 251 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 108 | No | 103 | No | 106 | No | 0 | No |
| World 7-4 | 265 | No | 257 | No | 328 | No | 163 | No |
| World 8-1 | 156 | No | 71 | No | 150 | No | 18 | No |
| World 8-2 | 234 | No | 71 | No | 192 | No | 13 | No |
| World 8-3 | 235 | No | 71 | No | 491 | No | 234 | No |
| World 8-4 | 107 | No | 106 | No | 118 | No | 138 | No |

Table 6.1: Training Run #1 Results

## 6.3 Training Run #2

### 6.3.1 Improvements

This training run was used as an analysis of the effectiveness of the base application of Mario Moments as input and the first refinement of the fitness function. Like the previous training run, at the start of a generation, a random Mario Moment was selected to train each genome with; however, instead of rewarding a genome for successfully reaching the goal, a punishment, $P$, was applied if a genome failed to reach the goal. If a genome reached the goal, $P = 0$ and if it failed, $P = L_{goal} - M_{end}$, where $L_{goal}$ was the location of the goal in the Mario Moment and $M_{end}$ was the furthest progress the genome achieved. This resulted in a fitness function with the form:

$$F = M_{end} - M_{start} - (t * \frac{1}{2}) - P \tag{6.2}$$

### 6.3.2 Results

In order to get a good sense of the efficacy of the fitness function alteration, this training run ran for 3 weeks and generated a total of 1042 generations in that time. This training run resulted in two distinct species dominating the population (Table 6.2) and was the start of a pattern that began to emerge in successful training runs. As the genomes were neuroevolving during a training run, two methods of play seemed to be the first divergence of the species in a population:

- A species that would run forward quickly and make small jumps over obstacles and pitfalls.
- A species that would jump as much as possible while still making small forward progress.

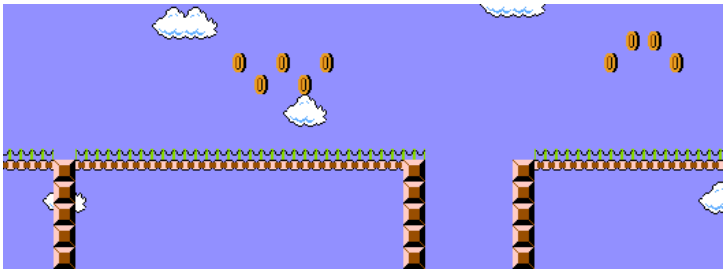


Figure 6.1: Section of World 2-3

Species #2 took the first approach where it would build up speed while making small jumps. This would sometimes result in it successfully avoiding enemies and leaping over/on top of small obstacles. Unfortunately, this was not an overall successful approach because it also had a tendency to stop jumping when it touched an obstacle, so apart from World 5-1 and World 8-1, it often made little progress through levels.

Species #1 took the second approach where it would try to jump a lot and often try to get as much height as possible out of those jumps. This approach often led to more success than Species #2, with the best example being World 2-3 (Figure 6.1). This level consists of long, flat bridges and the only enemies fly up at Mario from the bottom of the screen. Species #1 achieved a distance of 1039 units, which was it's best performance, specifically because it focused on making as much forward progress as possible. This behaviour could have even lead to this species beating the entire level, if not for an unfortunately timed jump before the first pitfall.

The reason for this divergence comes from the application of Mario Moments as the dataset. Because a Mario Moment can start from almost any part of a level, this leads to essentially two possible scenarios for any given generation of training. Either the Mario Moment begins in a flat, open section of the level, which benefits a genome that tends to

run forward or the Mario Moment begins near an obstacle or pitfall, which would benefit a genome that prioritizes jumping high. Since this training run has a generation only select a single Mario Moment to train the genomes on, some generations would favor the running genomes and some generations would favor the jumping genomes. This had the effect of narrowing the search performed by the algorithm along these two lines, which was actually a beneficial effect; however, it also had the detrimental effect of not allowing those behaviours to develop outside of their specific species. This would have to be addressed in future training runs.

| | Species #1 | | Species #2 | |
|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 554 | No | 276 | No |
| World 1-2 | 251 | No | 250 | No |
| World 1-3 | 385 | No | 276 | No |
| World 1-4 | 190 | No | 76 | No |
| World 2-1 | 282 | No | 266 | No |
| World 2-2 | 0 | No | 0 | No |
| World 2-3 | 1039 | No | 138 | No |
| World 2-4 | 303 | No | 76 | No |
| World 3-1 | 331 | No | 383 | No |
| World 3-2 | 906 | No | 356 | No |
| World 3-3 | 251 | No | 379 | No |
| World 3-4 | 211 | No | 129 | No |
| World 4-1 | 282 | No | 282 | No |
| World 4-2 | 426 | No | 306 | No |
| World 4-3 | 367 | No | 353 | No |
| World 4-4 | 95 | No | 95 | No |
| World 5-1 | 409 | No | 652 | No |
| World 5-2 | 139 | No | 138 | No |
| World 5-3 | 375 | No | 274 | No |
| World 5-4 | 304 | No | 76 | No |
| World 6-1 | 411 | No | 304 | No |
| World 6-2 | 250 | No | 250 | No |
| World 6-3 | 341 | No | 239 | No |
| World 6-4 | 190 | No | 76 | No |
| World 7-1 | 394 | No | 251 | No |
| World 7-2 | 0 | No | 0 | No |
| World 7-3 | 752 | No | 138 | No |
| World 7-4 | 303 | No | 129 | No |
| World 8-1 | 302 | No | 506 | No |
| World 8-2 | 218 | No | 219 | No |
| World 8-3 | 393 | No | 235 | No |
| World 8-4 | 130 | No | 251 | No |

Table 6.2: Training Run #2 Results

## 6.4 Training Run #3

### 6.4.1 Improvements

As an attempt to solve the problems that arose in the previous training run, this training run saw an alteration in the application of the Mario Moment dataset and a slight change to the fitness function. Instead of selecting a single Mario Moment for every generation, a list of all Mario Moments was created and then shuffled at the start of training. Each genome in every generation was then trained on the same list of Mario Moments. Additionally, when a genome reached the goal for the level of that Mario Moment, the next Mario Moment was loaded and the training of that genome continued. Training did not progress to the next

genome until it either beat every Mario Moment, failed to make forward progress within the timeout time, or Mario died during a run. In this way, a genomes fitness was measured by not just how far it progressed through a single Mario Moment, but also by how many Mario Moments it was able to progress through. Furthermore, the fitness function was also slightly modified by removing the penalty for not completing a Mario Moment. This produced a fitness function of the form:

$$F = M_{end} - M_{start} - (t * \frac{1}{2}) \tag{6.3}$$

### 6.4.2 Results

The goal of this modification to the fitness function was to focus the training on incentivising genomes to make progress through multiple Mario Moments. While this approach was able to successfully develop genomes that could make progress through a given level, it was ultimately flawed in it's primary assumption. The training resulted in the creation of only a single species throughout 309 total generations of training (Table 6.3). The relatively small number of generations in this training run is due to only running this session for one week.

After a week of training, each genome was able to complete 90% of the first Mario Moment in the list and then became stuck in the exact same place. This may have been alleviated with more training time; however, it was indicative of a pattern where the genomes would become stuck in a local maximum and then further training time would be wasted trying to get out of it. This is evidenced by the creation of only one dominant species within the population. Because a genetic algorithm like NEAT uses crossover mating to allow genomes with different structures to combine their knowledge to create better, more complex behaviours, it is supposed to generate a variety of different structures to combine. The focusing of the genomes into a single species that happened in this training run, effectively negates the algorithms ability to do this. I hypothesized that the reason this training run became so focused was due to the list of Mario Moments only being shuffled once at the start of training. This lead to the genomes only ever seeing the same level over and over, causing the very same overfitting issue that was seen in Seth Bling's MarI/O experiment. Because of this I decided to end this training run early in order to test my solution.

## 6.5 Training Run #4

### 6.5.1 Improvements

For this training run, I reviewed the basics of training image classification networks that I learned in my machine learning classes. A very important step in the training of deep neural network classifiers is to shuffle the dataset at every epoch. This is primarily done to prevent the classifier from overfitting to the training dataset due to it learning the order in which the images are input into the network. So, to alleviate the overfitting issue from the

| | Species #1 | |
|---|---|---|
| Level | Distance | Reached Goal? |
| World 1-1 | 396 | No |
| World 1-2 | 218 | No |
| World 1-3 | 243 | No |
| World 1-4 | 188 | No |
| World 2-1 | 284 | No |
| World 2-2 | 0 | No |
| World 2-3 | 140 | No |
| World 2-4 | 296 | No |
| World 3-1 | 459 | No |
| World 3-2 | 356 | No |
| World 3-3 | 256 | No |
| World 3-4 | 221 | No |
| World 4-1 | 282 | No |
| World 4-2 | 619 | No |
| World 4-3 | 351 | No |
| World 4-4 | 92 | No |
| World 5-1 | 143 | No |
| World 5-2 | 170 | No |
| World 5-3 | 359 | No |
| World 5-4 | 296 | No |
| World 6-1 | 410 | No |
| World 6-2 | 250 | No |
| World 6-3 | 346 | No |
| World 6-4 | 188 | No |
| World 7-1 | 377 | No |
| World 7-2 | 0 | No |
| World 7-3 | 750 | No |
| World 7-4 | 296 | No |
| World 8-1 | 506 | No |
| World 8-2 | 303 | No |
| World 8-3 | 235 | No |
| World 8-4 | 105 | No |

Table 6.3: Training Run #3 Results

last training run, I created a Fischer-Yeats shuffling algorithm [9] that I applied at the start of each generation to ensure every Mario Moment had an equal chance of appearing in any spot in the list.

In addition, I also slightly tweaked the fitness function to be more lenient on genomes that made slow, steady progress through a level. By reducing the frame penalty portion of the fitness function to $\frac{1}{4}$ of the number of frames generated on a run, I was able to prioritize level progress higher than the speed the progress was made. This would allow, for example, a genome that made 950 units of progress in 5 seconds a fitness of $F = 875$ vs a genome that made 900 units of progress in 2 seconds a fitness of $F = 870$. This resulted in a fitness function of the form:

$$F = M_{end} - M_{start} - (t * \frac{1}{4}) \tag{6.4}$$

### 6.5.2 Results

The addition of Mario Moment reshuffling with every new generation completely negated the overfitting issue and created a robust population that contained the largest number of species, six, of any training run. Also, by reviewing the results of the champion verification test (Table 6.4), each different species was able to develop different approaches to solving a level. This lead to some species being more skilled at high jumps, some species more suited to long jumps, and some species more suited to flat areas that only required small jumps. An important example of these behavioural differences is species #2 in World 2-2. This world has an extremely tricky section in the beginning that very few genomes are able to solve. As seen in Figure 6.2 Mario must solve this area in one of three ways. The first is to simply

wait for the Koopa to walk through the single tile gap in the bricks to clear the path, the second is for Mario to have eaten a mushroom to turn into big Mario and break the blocks in front of the gap to jump over it, or the third is for Mario to move fast enough to beat the Koopa through the gap. This species actually learned to execute a frame perfect trick that speedrunners use to clear this challenge. It requires exploiting a bug in the game, where if Mario is in the falling state and he makes contact with a killable enemy in any way, the game counts that as a **stomp** action and the enemy is killed. In this instance, Mario must be in the gap and jump exactly two frames before touching the Koopa, which will result in Mario bouncing off the above bricks, entering the falling state one frame before touching the Koopa, and, finally, **stomping** the Koopa in the frame they make contact (Figure 6.3).

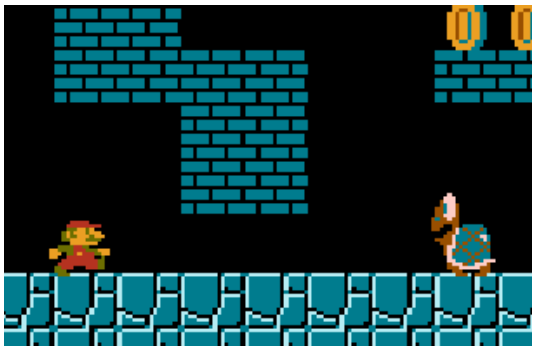
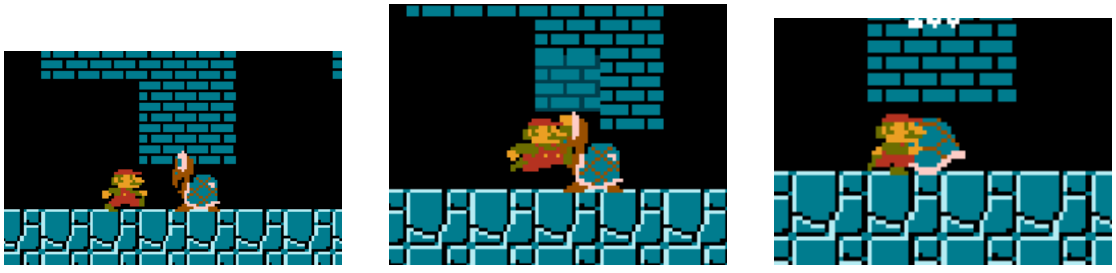


Figure 6.2: Challenging location in World 2-2

As stated and exemplified above, this diversity in genome behaviours is very important to the neuroevolution process, because when a genome that specializes in high jumps is mated with a genome that specializes in long jumps, there is a chance that the resulting child genome will now be specialized in both high and long jumps. Over time, this combination of specialties will result in genomes that are able to intelligently select the proper behaviour to overcome a wide set of challenges, ultimately leading to genomes that have broad, general understanding of how to beat an SMB level. I was so pleased with the progress of this training run, I considered letting it train for the rest of thesis research time; however, I wanted to experiment with some tweaks to the fitness function that may incentivize genomes to keep Mario alive, so I ended this training run after four weeks of training and 2167 total generations.



(a) Mario entering the gap (b) Mario jumping (c) Mario killing the Koopa

Figure 6.3: Mario executing a frame perfect trick to kill the Koopa

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | | Species #5 | | Species #6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 12 | No | 395 | No | 635 | No | 555 | No | 555 | No | 683 | No |
| World 1-2 | 282 | No | 849 | No | 115 | No | 251 | No | 314 | No | 122 | No |
| World 1-3 | 0 | No | 256 | No | 252 | No | 393 | No | 264 | No | 398 | No |
| World 1-4 | 191 | No | 193 | No | 411 | No | 411 | No | 190 | No | 509 | No |
| World 2-1 | 288 | No | 275 | No | 431 | No | 491 | No | 290 | No | 315 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 12 | No | 1046 | No | 1036 | No | 356 | No | 771 | No | 1297 | No |
| World 2-4 | 0 | No | 320 | No | 303 | No | 378 | No | 307 | No | 304 | No |
| World 3-1 | 13 | No | 327 | No | 460 | No | 364 | No | 459 | No | 567 | No |
| World 3-2 | 10 | No | 143 | No | 620 | No | 615 | No | 457 | No | 321 | No |
| World 3-3 | 29 | No | 278 | No | 288 | No | 268 | No | 292 | No | 278 | No |
| World 3-4 | 3 | No | 476 | No | 236 | No | 366 | No | 214 | No | 212 | No |
| World 4-1 | 284 | No | 282 | No | 1804 | No | 283 | No | 283 | No | 493 | No |
| World 4-2 | 218 | No | 174 | No | 219 | No | 168 | No | 284 | No | 170 | No |
| World 4-3 | 5 | No | 353 | No | 353 | No | 204 | No | 351 | No | 208 | No |
| World 4-4 | 587 | No | 92 | No | 9 | No | 586 | No | 92 | No | 92 | No |
| World 5-1 | 30 | No | 418 | No | 229 | No | 231 | No | 142 | No | 130 | No |
| World 5-2 | 11 | No | 218 | No | 414 | No | 682 | No | 698 | No | 186 | No |
| World 5-3 | 265 | No | 345 | No | 261 | No | 264 | No | 400 | No | 390 | No |
| World 5-4 | 4 | No | 341 | No | 378 | No | 248 | No | 375 | No | 272 | No |
| World 6-1 | 12 | No | 300 | No | 605 | No | 449 | No | 456 | No | 301 | No |
| World 6-2 | 250 | No | 252 | No | 250 | No | 250 | No | 250 | No | 250 | No |
| World 6-3 | 345 | No | 241 | No | 787 | No | 239 | No | 237 | No | 319 | No |
| World 6-4 | 192 | No | 191 | No | 410 | No | 288 | No | 188 | No | 288 | No |
| World 7-1 | 17 | No | 215 | No | 808 | No | 395 | No | 399 | No | 523 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 8 | No | 1041 | No | 1034 | No | 750 | No | 1038 | No | 1038 | No |
| World 7-4 | 2 | No | 304 | No | 303 | No | 294 | No | 1241 | No | 260 | No |
| World 8-1 | 293 | No | 298 | No | 507 | No | 507 | No | 507 | No | 506 | No |
| World 8-2 | 11 | No | 161 | No | 300 | No | 168 | No | 299 | No | 266 | No |
| World 8-3 | 30 | No | 219 | No | 928 | No | 492 | No | 492 | No | 207 | No |
| World 8-4 | 124 | No | 101 | No | 8 | No | 107 | No | 126 | No | 131 | No |

Table 6.4: Training Run #4 Results

## 6.6 Training Run #5

### 6.6.1 Improvements

For this training run, because the dataset reshuffling was previously quite successful, the only changes that I made to the algorithm was to the fitness function. My primary goal for these changes was to prioritize the genomes that were able to keep Mario alive when a run was ended. Because a large number of Mario Moments lead to a quick death if the correct behaviour was not selected, I knew that if I just gave a fitness bonus for Mario being alive, genomes would be trained to take no action. Therefore, in order to get the survival bonus ($S$), a genome must have made at least 100 units of progress. My thought was that if a genome was forced to take some amount of action before qualifying for the bonus, that would result the genomes that learned how to make progress in a level and keep Mario alive. As you will see in the next section, this was the result of this training, just not in the way that I intended it to be. In addition to this, I also wanted to test how giving a small fitness bonus for killing enemies would affect the training of genomes. The fitness function for this training run took the form of:

$$F = M_{end} - M_{start} - (t * \frac{1}{4}) + S + (20 * N_{stomp}) \tag{6.5}$$

Where $N_{stomp}$ is the total number of enemies killed by a genome and $S = 1000$ if Mario was alive at the end of a run or $S = 0$ if Mario was dead. Because the flaw in my approach to this training run became apparent once the genomes figured out how to survive the initial challenge of a Mario Moment, it only ran for a total of two weeks and generated 610 total generations.

### 6.6.2 Results

Reviewing the results of the champion verification run (Table 6.5), we can see that species #1, #2, and #3 each had a couple of levels they were able to make decent progress on; however, for the most part none of the species were able to make much meaningful progress on any levels. This is because once the genomes learned that keeping Mario alive was beneficial, they learned to make exactly enough progress to qualify for that bonus and then stopped moving. Despite this undermining the intended goal of teaching genomes to keep Mario alive **while** making progress through a level, it does highlight an important lesson about designing a fitness function for neuroevolution and, really, reinforcement learning in general.

It shows us that we have to pay careful attention when designing a fitness function, to ensure that we are guiding the agent to solve the **entire** problem they are tasked with. We, as the designers of these systems, can't get too granular with our rewards/punishments or we risk influencing our agents along unintended, detrimental paths. In this case, it is obvious that the agent should keep Mario alive, as that is the only way to continually make progress in a level; however, explicitly rewarding a genome solely for keeping Mario alive will often lead to the genomes gaming the system to maximize its own fitness. Instead, we have to capture some sort of implicit reward for keeping Mario alive, such as rewarding the genome for making progress through a level or maintaining a high speed during a run. In this way, we leave it up to the genome to learn the best way to make continual progress and thereby implicitly teaching the genome that keeping Mario alive is a good thing to do.

The reward for killing enemies actually had zero effects on the training of genomes. Because the survival bonus was so high, most genomes stopped moving to end a run before any of them reached an enemy. This is even more evidence of how granular rewards for specific behaviours have to be carefully crafted and, most often, should be ignored in favor of rewards/punishments that imply the proper behaviour. There could be an argument for having a much smaller reward for Mario surviving a run; however, there would need to be a lot of ad hoc research to justify what exactly that value should be. Ultimately, this would mostly be wasted effort because the goal of reinforcement learning is not to dictate *exactly* what behaviours an agent should take but instead it is to show an agent *how* a particular behaviour affects the overall state of the world it exits in. This way an agent learns not just what behaviours are possible but also the context in which the behaviour exists in. In most cases, the context of why one behaviour should be chosen over another is more important than the behaviour itself because it allows an agent to be flexible and adapt to new situations. It is all fine and well to tell Mario to jump when he sees an enemy, as running into it will lead to death; however, without the entire context of the current situation this jump could still lead to Mario's death. For instance, if that enemy is on the edge of a pitfall, simply jumping over the enemy will just end up with Mario falling into the pit. The agent controlling Mario

has to be aware of the larger world in order to survive this challenge, otherwise it may miss a floating platform that would allow Mario to avoid both the enemy and the pitfall and enable him to continue to make progress through the level.

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | | Species #5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 395 | No | 396 | No | 234 | No | 256 | No | 20 | No |
| World 1-2 | 79 | No | 250 | No | 14 | No | 125 | No | 0 | No |
| World 1-3 | 244 | No | 241 | No | 248 | No | 206 | No | 20 | No |
| World 1-4 | 187 | No | 189 | No | 0 | No | 428 | No | 0 | No |
| World 2-1 | 266 | No | 282 | No | 127 | No | 222 | No | 20 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 65 | No | 106 | No | 122 | No | 79 | No | 49 | No |
| World 2-4 | 268 | No | 297 | No | 0 | No | 268 | No | 0 | No |
| World 3-1 | 354 | No | 354 | No | 352 | No | 445 | No | 20 | No |
| World 3-2 | 906 | No | 738 | No | 780 | No | 144 | No | 20 | No |
| World 3-3 | 209 | No | 244 | No | 246 | No | 176 | No | 20 | No |
| World 3-4 | 235 | No | 235 | No | 0 | No | 237 | No | 0 | No |
| World 4-1 | 282 | No | 283 | No | 282 | No | 283 | No | 20 | No |
| World 4-2 | 82 | No | 156 | No | 10 | No | 125 | No | 0 | No |
| World 4-3 | 357 | No | 370 | No | 357 | No | 208 | No | 20 | No |
| World 4-4 | 90 | No | 94 | No | 14 | No | 93 | No | 0 | No |
| World 5-1 | 129 | No | 128 | No | 650 | No | 132 | No | 20 | No |
| World 5-2 | 154 | No | 154 | No | 138 | No | 140 | No | 20 | No |
| World 5-3 | 334 | No | 243 | No | 248 | No | 270 | No | 20 | No |
| World 5-4 | 268 | No | 297 | No | 0 | No | 268 | No | 0 | No |
| World 6-1 | 308 | No | 302 | No | 443 | No | 301 | No | 20 | No |
| World 6-2 | 250 | No | 250 | No | 53 | No | 141 | No | 20 | No |
| World 6-3 | 240 | No | 248 | No | 247 | No | 244 | No | 20 | No |
| World 6-4 | 187 | No | 189 | No | 0 | No | 428 | No | 0 | No |
| World 7-1 | 251 | No | 251 | No | 251 | No | 250 | No | 20 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 66 | No | 106 | No | 122 | No | 79 | No | 49 | No |
| World 7-4 | 265 | No | 297 | No | 0 | No | 268 | No | 0 | No |
| World 8-1 | 506 | No | 506 | No | 148 | No | 155 | No | 20 | No |
| World 8-2 | 188 | No | 165 | No | 156 | No | 218 | No | 20 | No |
| World 8-3 | 234 | No | 223 | No | 187 | No | 235 | No | 20 | No |
| World 8-4 | 107 | No | 128 | No | 16 | No | 107 | No | 1 | No |

Table 6.5: Training Run #5 Results

## 6.7 Training Run #6

### 6.7.1 Improvements

This training run was a test of two changes to the fitness function. The first of these changes was to alleviate the issue seen in the last training run where genomes would make just enough progress to qualify for the survival bonus and then stop making progress to end the run with Mario alive. Instead of basing the survival bonus on a fixed amount of progress, I changed it to be based off of the speed of Mario when a genome's run was ended; however, since the only way a genome's run could end was Mario dying or staying still for too long, this bonus was completely unachievable. The second change was designed to promote more intelligent jumping from a genome. Whenever Mario initiated a jump, if Mario was alive at the end of the jump, the genome was awarded a 50 point fitness bonus. This created a fitness function of the form:

$$F = M_{end} - M_{start} - (t * \frac{1}{4}) + S + (20 * N_{stomp}) + (50 * N_{jump}) \tag{6.6}$$

Where $N_{jump}$ is equal to the number of successful jumps during a genome's run and $S = 500$ if Mario's velocity was greater than six or $S = 0$ if not.

### 6.7.2 Results

Unfortunately, because the goal of this project was to create an agent that could effectively play SMB, this run was a complete failure. It essentially suffered from the same problem as the previous run, in that it focused too much on a particular behaviour an agent can do and failed to fully account for the complete context that behaviour exists in. The genomes produced from this training were extremely adept at jumping the maximum number of times possible; however, as the champion verification test data shows (Table 6.6), this generally led to very poor performance, except in the few worlds where jumping really fast is required to beat the level. This run ultimately had to be ended after a week of training due to a bug in fitness calculation. During generation 411, a single genome repeatedly got a zero fitness score. To the system, this meant that genome hadn't yet been ran for that generation, leading to an infinite loop of running that genome. This training run was very amusing to watch; however, so I guess it wasn't a total loss.

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | |
|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 395 | No | 0 | No | 254 | No | 368 | No |
| World 1-2 | 442 | No | 0 | No | 250 | No | 192 | No |
| World 1-3 | 178 | No | 0 | No | 249 | No | 223 | No |
| World 1-4 | 38 | No | 49 | No | 187 | No | 188 | No |
| World 2-1 | 291 | No | 241 | No | 285 | No | 266 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 75 | No | 0 | No | 31 | No | 78 | No |
| World 2-4 | 38 | No | 49 | No | 0 | No | 269 | No |
| World 3-1 | 195 | No | 162 | No | 459 | No | 167 | No |
| World 3-2 | 906 | No | 27 | No | 458 | No | 28 | No |
| World 3-3 | 176 | No | 145 | No | 245 | No | 236 | No |
| World 3-4 | 38 | No | 49 | No | 0 | No | 238 | No |
| World 4-1 | 283 | No | 282 | No | 282 | No | 256 | No |
| World 4-2 | 97 | No | 0 | No | 155 | No | 139 | No |
| World 4-3 | 161 | No | 0 | No | 353 | No | 207 | No |
| World 4-4 | 17 | No | 16 | No | 0 | No | 95 | No |
| World 5-1 | 399 | No | 229 | No | 223 | No | 572 | No |
| World 5-2 | 218 | No | 138 | No | 219 | No | 112 | No |
| World 5-3 | 189 | No | 36 | No | 245 | No | 43 | No |
| World 5-4 | 38 | No | 49 | No | 261 | No | 269 | No |
| World 6-1 | 303 | No | 208 | No | 315 | No | 282 | No |
| World 6-2 | 145 | No | 144 | No | 250 | No | 224 | No |
| World 6-3 | 177 | No | 144 | No | 245 | No | 224 | No |
| World 6-4 | 38 | No | 49 | No | 187 | No | 188 | No |
| World 7-1 | 251 | No | 97 | No | 346 | No | 100 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 75 | No | 0 | No | 356 | No | 78 | No |
| World 7-4 | 38 | No | 49 | No | 261 | No | 269 | No |
| World 8-1 | 506 | No | 29 | No | 147 | No | 29 | No |
| World 8-2 | 149 | No | 25 | No | 151 | No | 30 | No |
| World 8-3 | 94 | No | 0 | No | 214 | No | 83 | No |
| World 8-4 | 17 | No | 17 | No | 116 | No | 102 | No |

Table 6.6: Training Run #6 Results

## 6.8 Training Run #7

### 6.8.1 Improvements

Because of the failures of the previous two runs, this training run was more of a "back to basics" change to the fitness calculations. I removed the bonuses for killing enemies, jumping, and staying alive at the end of the run. In addition, I reintroduced the bonus for reaching the goal flag and I also introduced a system to reward Mario for maintaining a high speed during the run. For every unit of progress a genome made, if Mario's speed value was higher than 24, which is the max walking speed, that unit of progress counted as two units of progress. This created a fitness function of the form:

$$F = M_{end} - M_{start} - (t * \frac{1}{4}) + S + B_{goal} \tag{6.7}$$

Where $S$ equals the number of progress units where Mario's speed was greater than 24 and $B_{goal} = 1000$ if Mario reached the goal flag for that level or $B_{goal} = 0$ if not.

### 6.8.2 Results

This training run was a confirmation that rewarding/punishing behaviours that imply the intended goal are much more effective than rewarding/punishing specific behaviours directly. It produced a diverse population similar to training run #4 and champion verification test results (Table 6.7) that were also similar to that training run. Because of this similarity in training outcomes, I knew that I was back on the right track with my fitness function.

## 6.9 Training Run #8

### 6.9.1 Improvements

For my final training run, I knew that my fitness function was in good shape and so I decided to reassess my application of the Mario Moment dataset. For the last four training runs, each generation would shuffle the list of Mario Moments and then every genome in the population would attempt to get as far through that list as possible. Because of the results of training runs #4 and #7, I knew this was an effective baseline strategy; however, I knew there was still room for improvement. Because a genome's training run was cut off when Mario died or stopped making progress, there were many generations where genomes had short training runs. This would occur where the first Mario Moment in the list was a particularly difficult point in a level; therefore, many times every genome in a generation was only able to make small amounts of progress before their run ended. This created a situation where genomes that were ill suited to that particular Mario Moment but well suited to other Mario Moments could get a low fitness score and be killed off. This would result in the overall population losing important knowledge and potentially hampering the development of generalized knowledge about how to play SMB. Because the goal of this project was to create an agent that had that generalized knowledge, I realized this was an issue that needed to be fixed.

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | | Species #5 | | Species #6 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 554 | No | 395 | No | 276 | No | 395 | No | 255 | No | 275 | No |
| World 1-2 | 348 | No | 283 | No | 219 | No | 133 | No | 17 | No | 250 | No |
| World 1-3 | 276 | No | 404 | No | 370 | No | 279 | No | 277 | No | 280 | No |
| World 1-4 | 193 | No | 416 | No | 410 | No | 132 | No | 190 | No | 10 | No |
| World 2-1 | 304 | No | 285 | No | 280 | No | 284 | No | 267 | No | 301 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 1037 | No | 79 | No | 78 | No | 1042 | No | 108 | No | 123 | No |
| World 2-4 | 255 | No | 294 | No | 335 | No | 75 | No | 333 | No | 10 | No |
| World 3-1 | 385 | No | 458 | No | 322 | No | 460 | No | 363 | No | 329 | No |
| World 3-2 | 617 | No | 146 | No | 159 | No | 457 | No | 115 | No | 461 | No |
| World 3-3 | 277 | No | 279 | No | 275 | No | 252 | No | 271 | No | 280 | No |
| World 3-4 | 216 | No | 214 | No | 310 | No | 75 | No | 206 | No | 11 | No |
| World 4-1 | 283 | No | 283 | No | 282 | No | 398 | No | 284 | No | 282 | No |
| World 4-2 | 285 | No | 222 | No | 159 | No | 184 | No | 158 | No | 159 | No |
| World 4-3 | 356 | No | 378 | No | 373 | No | 377 | No | 355 | No | 356 | No |
| World 4-4 | 382 | No | 93 | No | 18 | No | 159 | No | 94 | No | 95 | No |
| World 5-1 | 650 | No | 227 | No | 141 | No | 143 | No | 121 | No | 233 | No |
| World 5-2 | 219 | No | 218 | No | 154 | No | 156 | No | 158 | No | 219 | No |
| World 5-3 | 276 | No | 264 | No | 370 | No | 266 | No | 277 | No | 374 | No |
| World 5-4 | 255 | No | 294 | No | 387 | No | 75 | No | 395 | No | 10 | No |
| World 6-1 | 302 | No | 467 | No | 302 | No | 300 | No | 301 | No | 304 | No |
| World 6-2 | 251 | No | 250 | No | 250 | No | 251 | No | 251 | No | 252 | No |
| World 6-3 | 241 | No | 237 | No | 356 | No | 239 | No | 272 | No | 338 | No |
| World 6-4 | 193 | No | 418 | No | 410 | No | 132 | No | 190 | No | 11 | No |
| World 7-1 | 339 | No | 396 | No | 318 | No | 253 | No | 252 | No | 394 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 78 | No | 79 | No | 78 | No | 1041 | No | 108 | No | 125 | No |
| World 7-4 | 288 | No | 294 | No | 321 | No | 76 | No | 382 | No | 10 | No |
| World 8-1 | 508 | No | 507 | No | 169 | No | 507 | No | 125 | No | 447 | No |
| World 8-2 | 332 | No | 302 | No | 544 | No | 171 | No | 219 | No | 301 | No |
| World 8-3 | 457 | No | 490 | No | 235 | No | 226 | No | 235 | No | 234 | No |
| World 8-4 | 126 | No | 938 | No | 125 | No | 98 | No | 107 | No | 104 | No |

Table 6.7: Training Run #7 Results

My solution to this was to run every genome in the population through all of the Mario Moments before moving to the next generation. This way, genomes that performed well in certain levels but poorly in others wouldn't get eliminated due to bad luck, but instead would be able to optimize their performance in the levels they did do well and potentially learn to do better in levels they didn't. It would only be when a genome failed to improve in both aspects that it would get eliminated from the population, in favor of genomes that were improving in both aspects.

This change in dataset application did necessitate a change in the fitness function. The principal idea behind the fitness function didn't change; however, just the calculation of a genomes progress through each of the Mario Moments was altered. Instead of adding up all the progress in each Mario Moment into a big number, the progress through each Mario Moment plus any bonuses was then averaged by the total number of Mario Moments. This way, genomes that continued to improve their average progress were allowed to survive and genomes who's average progress went down were killed off. This produced a fitness function of the form:

$$F = \frac{1}{N}\sum_{i=1}^{N} M_i^{end} - M_i^{start} - (t_i * \frac{1}{4}) + S_i + B_i^{goal} \tag{6.8}$$

Where $N$ equals the total number of Mario Moments and $M_i^{end}$, $M_i^{start}$, $t_i$, $S_i$, & $B_i^{goal}$ is the total progress, starting progress, total frames, speed bonus, and goal bonus, respectively, of the $i^{th}$ Mario Moment.

### 6.9.2 Results

This training run was unique in that it was the longest of any of the training runs with a training time of two months and it also produced the fewest generations with a total of 169 generations. This is primarily the result of each individual genome training run taking much longer to complete and, therefore, each generation also took much longer to complete. In general, each generation in previous training runs took between five to ten minutes to complete, compared to over an hour and half for each generation for this training run. This actually highlights one of the biggest issues facing neuroevolution techniques in machine learning, having to run hundreds of members of a population over a dataset that can contain tens of thousands of individual instances. A solution to this problem has been proposed in a paper by Morse and Stanley [18], and I will discuss how it can be applied to this project in the **Future Works** section of **Chapter 7**.

The improvement in dataset application really complemented the fitness function improvements from previous training runs. While this training run may have created a less diverse population of species than training runs #4 and #7, each species in this training run performed much better over a larger number of levels. Looking at the results of the champion verification test in Table 6.8, it can be seen that there were multiple levels that each species was able to achieve over 1000 units of progress, including multiple instances that were on levels outside of the training dataset. This shows that species were able to gain some level of generalized knowledge about how to play an SMB level, which was the entire goal of the project.

It was disappointing to see that none of the species were able to beat even a single level; however, I am quite confident that is achievable with more training time. Considering that a SMB level is generally about 3000 units long, species #3 was very close to beating World 3-2 as it was able to make 2650 units of progress on that level. Whats more, because other species were unable to make close to that progress, it also shows that each species was developing distinct styles of play. This is also evidenced by multiple other instances of this occurring, such as species #2 with World 2-4 and species # 3 with World 5-1. As noted before, this is a crucial component to successful neuroevolution and it was only a matter of time before the knowledge of those different styles was disseminated to the rest of the population.

| | Species #1 | | Species #2 | | Species #3 | | Species #4 | |
|---|---|---|---|---|---|---|---|---|
| Level | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? | Distance | Reached Goal? |
| World 1-1 | 556 | No | 394 | No | 395 | No | 394 | No |
| World 1-2 | 218 | No | 220 | No | 640 | No | 251 | No |
| World 1-3 | 384 | No | 386 | No | 724 | No | 386 | No |
| World 1-4 | 1208 | No | 1163 | No | 188 | No | 412 | No |
| World 2-1 | 266 | No | 975 | No | 432 | No | 867 | No |
| World 2-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 2-3 | 1985 | No | 1038 | No | 1037 | No | 1038 | No |
| World 2-4 | 303 | No | 1227 | No | 306 | No | 309 | No |
| World 3-1 | 458 | No | 571 | No | 890 | No | 870 | No |
| World 3-2 | 731 | No | 340 | No | 2650 | No | 340 | No |
| World 3-3 | 417 | No | 425 | No | 520 | No | 413 | No |
| World 3-4 | 315 | No | 812 | No | 226 | No | 234 | No |
| World 4-1 | 1803 | No | 1594 | No | 1282 | No | 1815 | No |
| World 4-2 | 304 | No | 160 | No | 225 | No | 308 | No |
| World 4-3 | 369 | No | 353 | No | 355 | No | 353 | No |
| World 4-4 | 95 | No | 94 | No | 586 | No | 95 | No |
| World 5-1 | 664 | No | 662 | No | 1371 | No | 662 | No |
| World 5-2 | 188 | No | 346 | No | 346 | No | 347 | No |
| World 5-3 | 279 | No | 391 | No | 391 | No | 393 | No |
| World 5-4 | 303 | No | 271 | No | 306 | No | 309 | No |
| World 6-1 | 410 | No | 507 | No | 507 | No | 478 | No |
| World 6-2 | 277 | No | 336 | No | 253 | No | 394 | No |
| World 6-3 | 648 | No | 640 | No | 336 | No | 651 | No |
| World 6-4 | 299 | No | 315 | No | 188 | No | 413 | No |
| World 7-1 | 682 | No | 345 | No | 396 | No | 346 | No |
| World 7-2 | 0 | No | 0 | No | 0 | No | 0 | No |
| World 7-3 | 595 | No | 751 | No | 546 | No | 515 | No |
| World 7-4 | 384 | No | 271 | No | 306 | No | 304 | No |
| World 8-1 | 532 | No | 783 | No | 532 | No | 781 | No |
| World 8-2 | 331 | No | 302 | No | 299 | No | 291 | No |
| World 8-3 | 491 | No | 1860 | No | 1855 | No | 1182 | No |
| World 8-4 | 132 | No | 1157 | No | 125 | No | 139 | No |

Table 6.8: Training Run #8 Results

# Chapter 7

# Conclusion

## 7.1 Future Work

The results from these training runs show that more research into the application of neuroevolution as a way to create neural networks that learn behaviours is definitely needed. For my future work into this project, there are two main areas that I would like to improve upon in order to get even better results. These areas are expanding the Mario Moments dataset and altering the application of the Mario Moments dataset.

### 7.1.1 Expanding Mario Moments

As stated in the **Chapter 5**, the current dataset of Mario Moments consists of 222 randomly selected points between World 1-1 and World 4-3. This random sampling of different points across these fifteen levels is a good place to start; however, I believe there is still room for improvement. Because these points were essentially created by playing the game and creating a *Save State* while randomly playing, some levels could end up having more Mario Moments in them than other levels. This could result in a bias forming that favors genomes learning some levels better than others. For my next attempt at this project, I would like to expand the total number of Mario Moments to 450 instances and also ensure that each level has exactly 30 Mario Moments in it. The benefits of this would be two fold, with one being that more instances would lead to the genomes seeing more similar looking challenges and therefore able to become more adept at solving them and, two, it would ensure that genomes wouldn't be biased by a particular level having more instances than others.

### 7.1.2 Application of Mario Moments

I mentioned in the results of training run #8 that one of the major challenges to neuroevolution is the need to apply a large number of training instances to every member of a population, at least according to the conventional wisdom. My above alterations would mean that a population of 300 genomes would each have to run through 450 training instances, leading to $300 * 450 = 135,000$ iterations every single generation. If each iteration takes thirty seconds to complete, that would mean each generation would take 47 days to complete. Luckily, in their paper *Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks* [18] Morse and Stanley propose a system that can dramatically reduce the training time for a generation.

The concept leverages a similar idea in "mini-batch" SGD, where the error gradient calculation used in backpropagation is calculated using a small set of training examples. This allows the training algorithm to make better use of parallelization techniques to speed up the training of a neural network model. Morse and Stanley propose a genetic algorithm that can also perform neuroevolution on a smaller batch of training examples, granting the algorithm both a sizable reduction in training time per generation while still maintaining the genetic stability of the population. Morse and Stanley are both keenly aware of the risks associated with selecting a random subset of a dataset for neuroevolution. Sometimes, genomes that perform well on the dataset overall may run into a situation where only training instances that they perform poorly on are selected for a generations training set. This is the very issue I experienced with training runs #4 and #7, where well performing genomes could encounter an unlucky string of training instances and suddenly find themselves eliminated from the population. Now the population has lost important knowledge that must be relearned.

In order to apply this same concept to my project and prevent this devastating loss of knowledge, the fitness calculation for genomes will have to be altered to include a factor Morse and Stanley called *Fitness Inheritance* [18]. Fitness inheritance is a system where a

genomes fitness is calculated by measuring it's fitness in the current generation and then is adjusted by the average fitness of it's parents. For this project, this would produce a fitness function of the form:

$$F = \frac{1}{N}\sum_{i=1}^{N} M_i^{end} - M_i^{start} - (t_i * \frac{1}{4}) + S_i + B_i^{goal} \tag{7.1}$$

$$F' = \frac{F_{p1} + F_{p2}}{2}(1 - d) + F \tag{7.2}$$

Where $F'$ is the adjusted fitness value for a genome for the current generation, $F_{pn}$ is the parent genome's fitness, $F$ is base fitness of the genome for the current generation, and $d$ is a decay hyperparameter that controls the weight of the fitness inheritance on the current generation.

By incorporating the fitness function modifications above and selecting batches of training instances such that at least one Mario Moment from each level are included, I will be able to increase my dataset size, ensure genetic diversity, and speed up the training process.

## 7.2 Conclusions

The results of training runs #4, #7, and #8 all show that the creation of an agent with a general understanding of how to effectively play SMB is entirely possible. As with most problems to be solved with machine learning, training time is one of the biggest limiting factors in the success or failure of an agent/model. However, taking time out of the equation, there are still other important factors to consider when developing a neuroevolution algorithm to train an agent, namely the fitness function that ranks the genomes and the dataset used in training. This is exemplified in the training runs with less than promising results because the misapplication of either one of these features directly resulted in the failure of those training runs. Training run #2 shows how exposing the genomes to only a single Mario Moment every generation had the effect of enforcing a divergence of behaviours so strongly that only two species were able to survive in the population. Training run #3 shows how the simple addition of a randomized list at the start of training is insufficient to properly train genomes. Training runs #5 and #6 exemplify rather succinctly how important it is to carefully craft the fitness function and avoid the temptation of focusing on specific behaviours when rewarding/punishing a genome. It is important that the fitness function *guide* the genomes to the proper behaviours by incentivising global goals like forward progress in a level or maintaining a high speed; otherwise, the genomes may unintentionally evolve to maximize a particular behaviour in a way that becomes ultimately detrimental to the success of the genome. Even though training runs #4 and #7 were successful in showing that genomes were able to create an amount of generalized gameplay knowledge, the way the Mario Moments dataset was used in those runs still needed to be altered to increase the efficacy of the training time. Reviewing Table 7.1, we can see that Training run #8 produced better results in only 169 generations by achieving the most progress in 50% of all levels tested vs 37% for training run #4 and 13% for training run #7. When compared to an average of 2150 generations in training run #4 and #7, this results in a 92% reduction in the number of generations.

| | Training Run 4 | Training Run 7 | Training Run 8 | Best Score |
|---|---|---|---|---|
| World 1-1 | 683 | 554 | 556 | 683 |
| World 1-2 | 849 | 348 | 640 | 849 |
| World 1-3 | 398 | 404 | 724 | 724 |
| World 1-4 | 509 | 416 | 1163 | 1163 |
| World 2-1 | 491 | 304 | 975 | 975 |
| World 2-2 | 0 | 0 | 0 | 0 |
| World 2-3 | 1297 | 1042 | 1038 | 1297 |
| World 2-4 | 378 | 335 | 1227 | 1227 |
| World 3-1 | 567 | 460 | 890 | 890 |
| World 3-2 | 620 | 617 | 2650 | 2650 |
| World 3-3 | 292 | 280 | 520 | 520 |
| World 3-4 | 476 | 310 | 812 | 812 |
| World 4-1 | 1804 | 398 | 1815 | 1815 |
| World 4-2 | 284 | 285 | 308 | 308 |
| World 4-3 | 353 | 378 | 369 | 378 |
| World 4-4 | 587 | 382 | 586 | 587 |
| World 5-1 | 418 | 650 | 1371 | 1371 |
| World 5-2 | 698 | 219 | 347 | 698 |
| World 5-3 | 400 | 374 | 393 | 400 |
| World 5-4 | 378 | 395 | 309 | 395 |
| World 6-1 | 605 | 467 | 507 | 605 |
| World 6-2 | 252 | 252 | 394 | 394 |
| World 6-3 | 787 | 356 | 651 | 787 |
| World 6-4 | 410 | 418 | 413 | 418 |
| World 7-1 | 808 | 396 | 682 | 808 |
| World 7-2 | 0 | 0 | 0 | 0 |
| World 7-3 | 1041 | 1041 | 751 | 1041 |
| World 7-4 | 1241 | 382 | 384 | 1241 |
| World 8-1 | 507 | 508 | 783 | 783 |
| World 8-2 | 300 | 544 | 331 | 544 |
| World 8-3 | 928 | 490 | 1860 | 1860 |
| World 8-4 | 131 | 938 | 1157 | 1157 |
| Total Best | 11 | 4 | 15 | |

Table 7.1: Best progress score for Training Runs #4, #7, & #8

# References


[1] Raya Alshammri, Ghaida Alharbi, Ebtisam Alharbi, and Ibrahim Almubark. Machine learning approaches to identify parkinson's disease using voice signal features. *Frontiers in Artificial Intelligence*, 6:1084001, March 2023.

[2] N. Altice. *I Am Error: The Nintendo Family Computer / Entertainment System Platform*. Platform Studies. MIT Press, 2015.

[3] P.J. Angeline, G.M. Saunders, and J.B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65, 1994.

[4] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[5] Seth Bling. Mari/o - machine learning for video games, June 2015.

[6] Mia Consalvo. Console video games and global corporations: Creating a hybrid culture. *New Media & Society*, 8(1):117–137, 2006.

[7] Andrew Cunningham. The nes: How it began, worked, and saved an industry, December 2021.

[8] J. E. Darnell and W. F. Doolittle. Speculations on the early course of evolution. *Proceedings of the National Academy of Sciences of the United States of America*, 83(5):1271–1275, March 1986.

[9] Manuel Eberl. Fisher–yates shuffle. *Archive of Formal Proofs*, September 2016.

[10] David Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 1 edition, 01 2002.

[11] Jason Gauci and Kenneth O. Stanley. Autonomous Evolution of Topographic Regularities in Artificial Neural Networks. *Neural Computation*, 22(7):1860–1898, 07 2010.

[12] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3–4):317–342, January 1997.

[13] Faustino J Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, page 356–1361, 1999.

[14] IGN. Ign's topp 100 games. https://web.archive.org/web/20100301132404/http://top100.ign.com/2000/010.html, 2006.

[15] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, page 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[16] David E. Moriarty and Risto Miikkulainen. Forming Neural Networks Through Efficient and Adaptive Coevolution. *Evolutionary Computation*, 5(4):373–399, 12 1997.

[17] David E. Moriarty and Risto Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1):11–32, January 1996.

[18] Gregory Morse and Kenneth O. Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*, page 477–484, Denver Colorado USA, July 2016. ACM.

[19] H Mühlenbein. Limitations of multi-layer perceptron networks - steps towards genetic neural networks. *Parallel Computing*, 14(3):249–260, 1990.

[20] News and Features Team. The top 25 videogame franchises. https://web.archive.org/web/20080228062503/http://ps3.ign.com/articles/749/749069p5.html, 2006.

[21] Dr. Tom Murphy VII Ph.D. The first level of super mario bros. is easy with lexicographic orderings and time travel ...after that it gets a little tricky.

[22] C. M. Radding. Homologous pairing and strand exchange in genetic recombination. *Annual Review of Genetics*, 16:405–437, 1982.

[23] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8. Spartan Books, 1962.

[24] N. Saravanan and D.B. Fogel. Evolving neural control systems. *IEEE Expert*, 10(3):23–27, 1995.

[25] J.D. Schaffer, D. Whitley, and L.J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37, 1992.

[26] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *CoRR*, abs/1912.10944(arXiv:1912.10944), December 2019. arXiv:1912.10944 [cs].

[27] N. Sigal and B. Alberts. Genetic recombination: the nature of a crossed strand-exchange between two homologous dna molecules. *Journal of Molecular Biology*, 71(3):789–793, November 1972.

[28] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*, 15(2):185–212, 04 2009.

[29] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *In Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 2002.

[30] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, June 2002.

[31] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[32] J.D. Watson and Tina A. Baker. *Molecular Biology of the Gene Fourth Edition*. Addison-Wesley, 4 edition, 1987.

[33] A.P. Wieland. Evolving neural network controllers for unstable systems. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume ii, pages 667–673 vol.2, 1991.

[34] Xin Yao and Yong Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90, 1998.

[35] Byoung-Tak Zhang, Heinz Muhlenbein, et al. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex systems*, 7(3):199–220, 1993.

[36] Corentin Zone, Robin Van Oirbeek, and Andrea Pennisi. Deep reinforcement learning: An introduction through video games. *Université catholique de Louvain*, 2020.

# Vita

The author was born in Baton Rouge, LA in 1988. He attended high school at the Mississippi School for Mathematics and Science and secured a full scholarship to Mississippi State University to pursue a Bachelor of Fine Arts, with a Concentration in Painting and Sculpture, which was achieved in 2012. After working as a World Wide Compliance Analyst for Electronic Arts, he joined the University of New Orleans in 2018 as an undergrad studying Computer Science. He later joined the Computer Science Masters program to pursue a Master of Science and Expert Certification in Machine Learning and AI in 2020. In 2021, he joined Dr. Ben Samuel's Light Lab as a Graduate Research Assistant, researching the application of Virtual Reality (VR) technology to the classroom and worked in conjunction with Dr. Matthew Tarr and Top Right Corner to develop VR training labs for the Chemistry Department.