A NEUROEVOLUTIONARY APPROACH TO DEVELOPING *SUPER MARIO BROS.* STRATEGIES

MITCHELL EITHUN

Abstract. The recent neuroevolutionary optimization scheme *Spectrum-diverse Unified Neuroevolution Architecture* (SUNA) developed by Vargas and Murata (2016) unifies current models for artificial neural networks and implements a diversity metric to encourage diverse networks to evolve. We use SUNA to evolve neural networks that learn to play the Nintendo video game *Super Mario Bros.* Motivated by neuroscience, we use a receptive field to encode game objects on screen. The receptive field data is given as input to neural networks, which output the actions of the game controller. Using default parameters and a small receptive field we found a neural network that completes WORLD 1-1 and has modest success on other levels. We also tested the idea of training networks on multiple levels, but encountered a local maximum.

Keywords. Evolutionary Algorithms, Super Mario Bros., Unified Neural Model, Artificial Neural Networks, Spectrum-diverse Unified Neuroevolution Architecture (SUNA)

1. Introduction

1.1. **The Domain.** The 1985 video game *Super Mario Bros.*[1] for the Nintendo Entertainment System (NES) is one of the best-selling video games of all time and in recent years has become a popular domain for artificial intelligence research. Researchers have worked generating new level content [1, 2], evolving strategies to play the game [3, 4, 5] and modeling player experience [6]. From 2009 to 2012, an internet contest was held to develop AI schemes in each of these categories [7]. There are also several YouTube videos explaining different approaches to playing *Mario* using AI strategies, including [8, 9, 10].

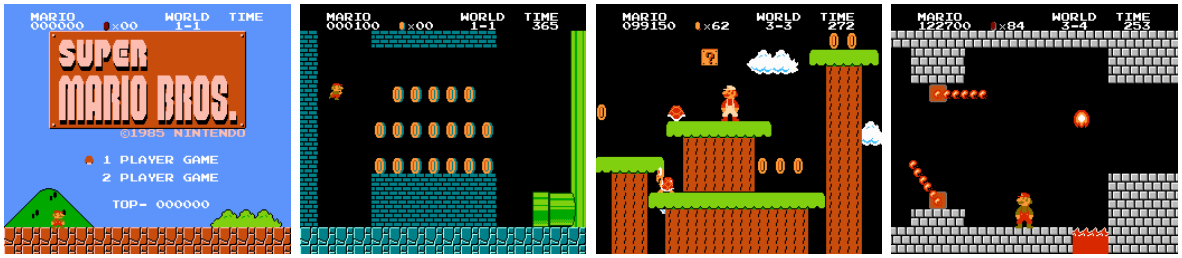


Figure 1. *Super Mario Bros.* gameplay.

Date: December 8, 2016.

[1] *Super Mario Bros.* is a trademark of Nintendo. Sprites and ROMs are presented here under Fair Use (17 U.S.C. § 107) for the educational purpose of demonstrating AI algorithms.

In the game *Super Mario Bros.* (SMB), the player guides Mario, an Italian plumber, through levels filled with platforms, obstacles and enemies and has the ability to make Mario jump, run and duck. Mario dies if he hits an enemy or falls in a pit. The goal of each level is to make it to the flagpole at the end without dying. Collecting power-ups and stomping enemies lessens the chance that Mario will lose a life and have to restart a level.

The simple goals of SMB make it a prime test subject for AI algorithms [11]. A generalized version of the game has been shown to be PSPACE-Complete, indicating that there is also some degree of computational difficulty required to complete the game [12]. This work focuses on using evolutionary algorithms to develop strategies that successfully complete SMB levels.

1.2. **Methods.** *Evolutionary algorithms* (also called *genetic algorithms*) are a general class of optimization algorithms motivated by the theory of biological evolution. Approaches to solve a problem, called *strategies* (or *chromosomes*), are repeatedly bred based on their problem-solving performance. Over time successful strategies emerge, beating out less effective strategies in a process like Darwinian natural selection.

We lay out the general scheme for an evolutionary algorithm (Figure 2). The algorithm is initialized with a set of random set of strategies, called a *population*. Each strategy in the population is evaluated to produce a *fitness* score, which indicates how well the strategy solves a given problem. In the reproduction phase, strategies with the highest fitness scores are *bred* (combined) using and random perturbations called *mutations* are added to produce a new *generation* of strategies. The process of evaluating fitness and breeding then repeats with the next generation. After many generations we expect the strategies to continually become more fit than their ancestors [13].

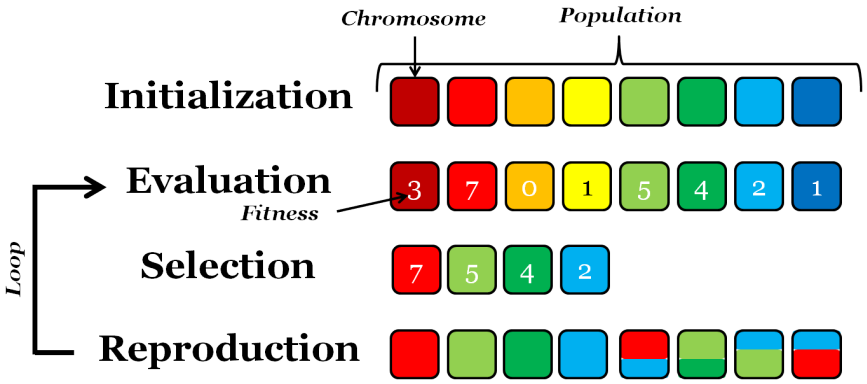


FIGURE 2. Evolutionary algorithm overview.

One specific type of evolutionary algorithm is *neuroevolution*, in which each strategy is a neural network. A *neural network* is a weighted directed graph that maps input stimulation to output actions through a series of functions, called *neurons*. The edges in the network are like synapses in a brain send signals across dendrites in a human brain. After input signals propagate through the neurons in the network, the network produces output signals [14]. The *topology* of a network refers to the graph theoretic qualities of the neurons and their connections. A neural network is shown in Figure 3.

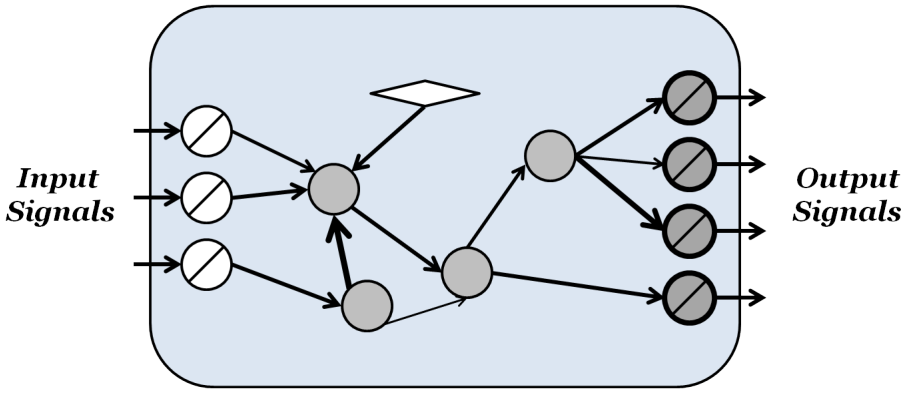


FIGURE 3. A neural network receives input signals from input neurons which are sent along weighted directed edges to other neurons. After signals propagate through the network, output neurons produce output signals.

To evolve neural networks in a neuroevolutionary algorithm we initialize a population of networks with different numbers of neurons and connections, test their performance on a problem and then breed them to produce better networks [13].

One popular formulation of neuroevolution is *Evolving Neural Networks Through Augmenting Topologies* (NEAT) [15]. Important features of NEAT include:

(1) Breeding strategies with different neural network topologies.
(2) Grouping similar strategies into *species* to control breeding and avoid mixing incompatible networks.
(3) Growing a network from minimal starting conditions.

NEAT has previously been implemented to evolve strategies to play *Super Mario Bros.* and *Super Mario World* [10].

Earlier this year Vargas and Murata developed an extension of NEAT called *Spectrum-diverse Unified Nueroevolution Architecture* (SUNA) [16]. Notable features of SUNA include:

(1) A small number of parameters (only 8, compared to the 33 required by NEAT).
(2) A "unified" representation of neurons which encompasses several different neural network schemes.
(3) An emphasis on preserving the diversity of different species in the evolution phase.

The developers of SUNA show that it outperforms NEAT in several small applications [16]. Accordingly, we evaluate the effectiveness of SUNA at evolving strategies that learn to play SMB.

1.3. **Evolving Strategies to Play Mario.** Inspired by the application of NEAT to play *Super Mario Bros.* and *Super Mario World* games in [10], we apply SUNA to play SMB. Each strategy is represented by a neural network. We construct a *repcetive field* to encode objects on the screen, which the networks uses as input. Then the network produces outputs that are interpreted as pressing buttons on the game controller. Through the evolution process strategies learn how to react to the surrounding environment to maximize how far they can make it through SMB levels.

We used two sets of training data to test the fitness of strategies. First we trained networks on WORLD 1-1, the first level of the game. A strategy's fitness is defined as far to the right the strategy was able to make it in the level before dying or standing still. After 722 generations we found a strategy that successfully completes WORLD 1-1. We found that this strategy had some success on other levels but it can only complete WORLD 1-1. The second set of training data contains four different levels: WORLDs 1-1, 1-2, 3-2 and 5-1. Strategies was randomly spawned into one of these levels during each fitness test. After 1400 generations the algorithm encountered a persistent local maximum and no strategy was able to complete any of the training levels.

1.4. **Outline.** This paper is organized as follows. In Section 2 we touch on the key features of SUNA. In Section 3 we explain how we represent SMB strategies using neural networks. In Section 4 we discuss the results of our evolution experiments. In Section 5 we discuss the performance of the algorithm, limitations of our strategy representation and considerations for future work.

2. The SUNA Scheme

Spectrum-diverse Unified Nueroevolution Architecture (SUNA) is a recent neuroevolutionary learning algorithm developed by Vargas and Murata [16]. In many respects it is an extension of *Neuroevolution through Augmenting Topologies* (NEAT), developed by Stanley and Miikkulainen [15]. SUNA fits into the taxonomy of optimization algorithms as follows:

$$\text{SUNA} \subset \text{Neurovolution} \subset \text{Evolutionary Algorithms} \subset \text{Learning Algorithms}.$$

We outline the structure of SUNA and explain some of its key features and algorithms, including a unified neural model and a speciation process to encourage diverse strategies to evolve.

2.1. **Unified Neural Model.** SUNA uses a general neuron representation scheme to capture the ways in which neural networks vary in size and topology. This *unified neural model* encapsulates a variety of different schemes in the literature by using different types of neurons, different neuronal timescales and neuromodulation.

Recall that a neural network is a weighted directed graph in which the vertices are neurons. Incoming edges are *inputs* and outgoing edges carry an *output*. Each neuron applies an *activation function* to the sum of all inputs weighted by their connection weights to produce an output. The activation functions used in SUNA are identity, sigmoid, random and threshold. Along with using different activation functions, the model allows for different *neuronal timescales*. Instead of updating at the same rate, each neurons has its own *adaptation speed* which makes it possible for signals to propagate through neurons at different rates.

More formally, consider a neuron with activation function f, and inputs $x_1, \ldots, x_n$ coming along edges weighted by $\omega_1, \ldots, \omega_n$. The network sees the composite input

$$\text{input} = f\left(\sum_{i=1}^{n} \omega_i x_i\right).$$

The internal state of the neuron is updating according to the rule

$$\text{state}_t = \text{state}_{t-1} + \frac{1}{\text{adaptationSpeed}}(\text{input} - \text{state}_{t-1}).$$

The current state (state_t) is then used as output. Notice that the neuron updates instantly if adaptionSpeed $= 1$, but older outputs will persist if adaptionSpeed > 1.

Control neurons are also introduced to implement *neuromodulation*, which allows neurons to enable or disable other neurons from firing. The outputs from a control neuron are called *control signals*. If a neuron has control signals $\text{cs}_1, \ldots \text{cs}_n$ coming in along edges weighted by $\omega_1, \ldots, \omega_n$, then the neuron activates (produces an output) if

$$\sum_{i=1}^{n} \omega_i \text{cs}_i \geq \text{excitation threshold}.$$

The excitation threshold is a fixed parameter for all neurons.

2.2. **Speciation.** To evolve more diverse strategies with SUNA, parents are picked through a process called *speciation*. Speciation uses the *spectrum* of a network, an array that tracks certain numeric features of a network, including

(1) the number of identity neurons,
(2) the number of sigmoid neurons,
(3) the number of threshold neurons,
(4) the number of random neurons,
(5) the number of control neurons, and
(6) the number of neurons with adaption speed greater than one.

Using a *novelty metric* to determine the distance between two spectrums, networks are sorted into *subpopulations* of similar networks. (More precise details are available in [16].) After fitness testing, the best individual from each subpopulation becomes a parent in the next generation and all other strategies are discarded.

2.3. **Evolution.** In the evolution phase, the best individuals in each subpopulation become parents in the next generation, forming the basis for the rest of the population. A child is created by choosing a random parent and randomly applying several of the following mutations.

(1) **Adding a neuron.** Two random connections are added to this neuron: one from the neuron and one to the neuron. The *control neuron probability* parameter is the chance that the neuron will be a control neuron. Otherwise its type is chosen randomly.
(2) **Removing a neuron.** All connections to and from the neuron are also removed.
(3) **Adding a connection.** A connection is made between two random neurons and initial weights are 1 or -1.
(4) **Removing a connection.** No other modifications are made.

The number of mutations we perform is a parameter called "number of step mutations". The probability of performing mutation $i \in \{1, 2, 3, 4\}$ is given by the i^{th} entry of the *mutation probability array* parameter.

After mutations have been performed, all connection weights in the network have a 50% chance of being perturbed by a value in $[-\omega, \omega]$, where is ω is the current value of the connection.

2.4. **Initialization.** The number of networks in each generation is a parameter. Each network in the initial population has a set of input and output neurons in accordance with the application of the algorithm. A parameter number of "initial mutations" are applied to each individual to create some initial diversity. No connection weights are perturbed in this step.

3. *Super Mario Bros.* As a Learning Domain

In this section we explain how we represented SMB strategies using neural networks and how we ran the C++ implementation of SUNA inside an emulator. Motivated by [10, 3], we encoded game objects on the screen using a receptive field to model what a strategy can see on screen. Each neural network strategy maps these inputs to button presses on the controller.

3.1. **Environment.** Our build was done on Ubuntu 16 running inside Oracle VM VirtualBox on a Windows PC. We used the C++ implementation of SUNA provided by the authors on Github as the `Physis-Shard` library (`github.com/zweifel/Physis-Shard`). This library has several dependencies, notably the learning library `zweifel` (`github.com/zweifel/zweifel`), which in turn required the `boost` library.

To play the game we used the FCEUX 2.2.3 NES emulator (`fceux.com/web/home.html`), built on the `SDL` multimedia library. To speed up trials we made a small hack to FCEUX that prevents SDL showing video output on the screen. The emulator API is exposed to Lua 5.1, which we used to read from and write to NES memory while the game is running and to realize the training environment. We used the Lua package `Alien` (`github.com/mascarenhas/alien`) as a Foreign Function Interface to access shared library functions written in C++ that encapsulate the functionality of the SUNA library. Some procedures to find game elements in memory were borrowed from [10]. Figure 4 provides an overview of the software used.

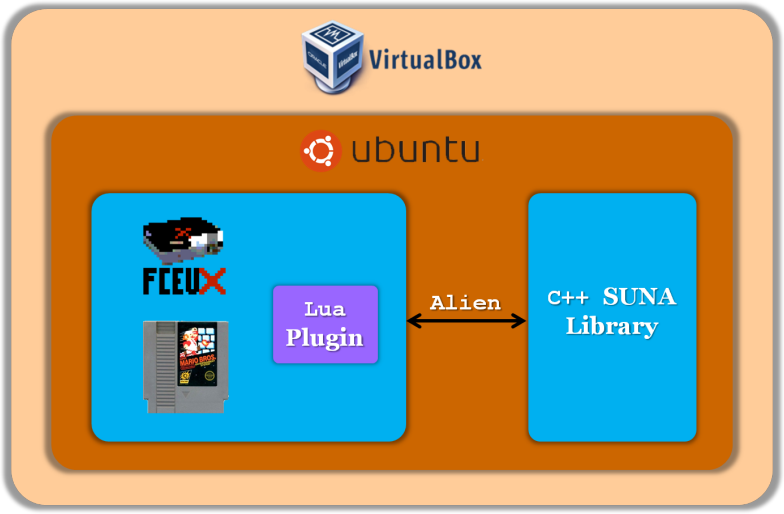


Figure 4. The development scheme. The Alien package allows Lua to call functions from shared libraries (such as .so). This allowed us to simply use the existing C++ implementation of SUNA.

3.2. **The Neural Network.** The neural network maps screen objects to controller inputs. (See Figure 5.) We consider objects in a box centered on Mario. Borrowing from neuroscience, we call this neighborhood the *receptive field*, representing the environmental observations that a strategy makes while trying to play the game. The network outputs six real numbers that are interpreted as Boolean values that represent the actions of buttons on the NES controller. Since SUNA uses random neurons, the neural network is not a well-defined function in the mathematical sense and is nondeterministic in the computing sense.

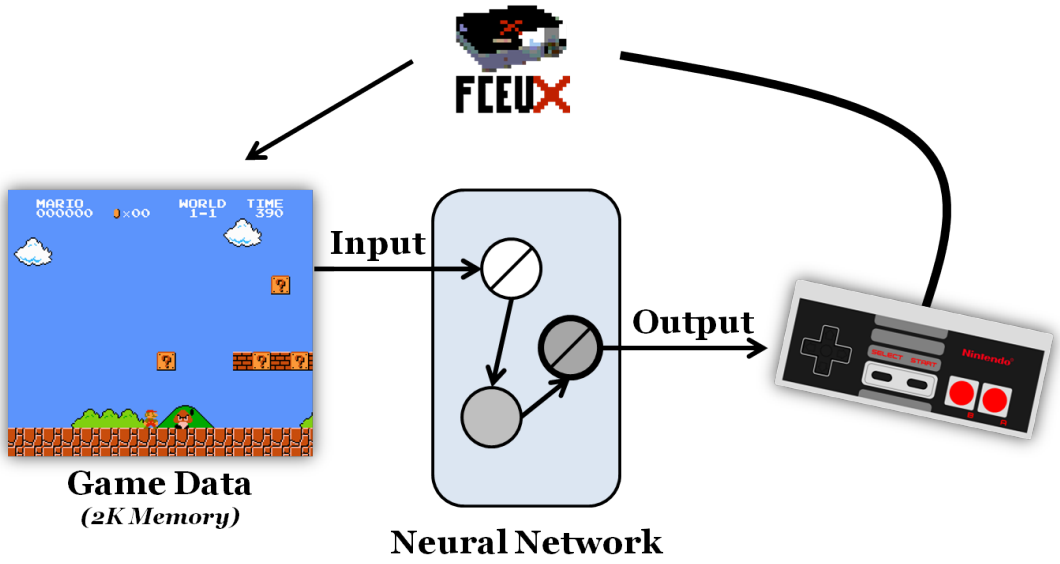


FIGURE 5. Each neural network interacts with the game by taking game data as input and producing output that represents the state of the NES controller.

3.2.1. *Network Input.* In *Super Mario Bros.*, the screen is drawn using a 16×14 grid of square *tiles*, illustrated in Figure 6.

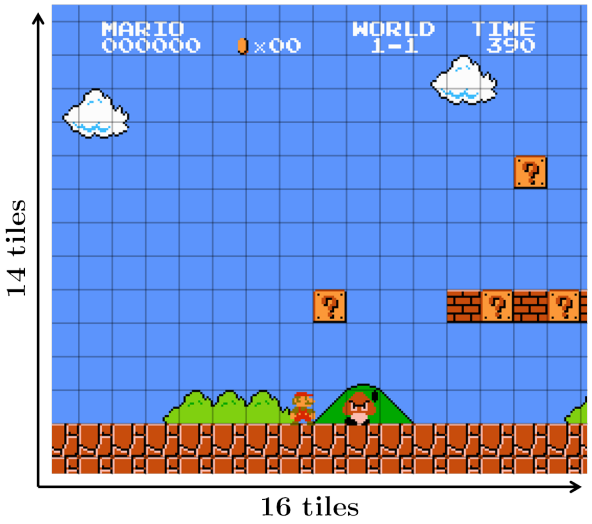


FIGURE 6. The tile boundaries in SMB. Each tile is further divided into four square *sprites* that are woven together to produce each object.

Our receptive field is a square subgrid along tile boundaries centered at Mario. Fix a positive integer radius r. We define the *input grid of radius* r as the $(2r+1) \times (2r+1)$ subgrid in which Mario is at the center. Each tile in the receptive field is mapped to an input neuron and so there are $(2r+1)^2$ input neurons in the network.

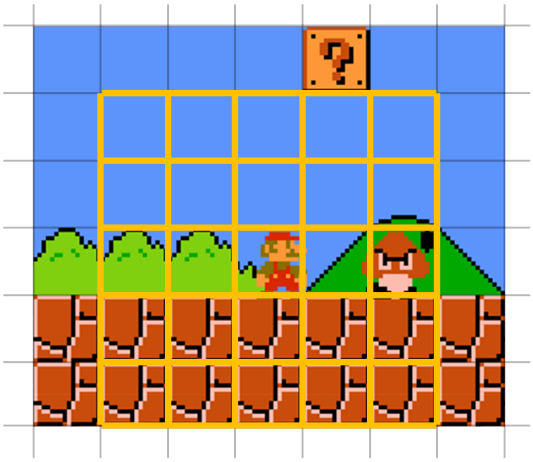


FIGURE 7. A receptive field of radius 2.

We can define this region mathematically using the Chebyshev topology. For two points $\vec{x} = (x_1, x_2)$, $\vec{y} = (y_1, y_2)$ in the Cartesian plane, the *Chebyshev metric* is defined as

$$d(\vec{x}, \vec{y}) := \max\{|x_1 - x_2|,\ |y_1 - y_2|\}.$$

Hence, the receptive field is the closed ball of radius r with respect to the Chebyshev metric.

The tiles that Mario can interact with in WORLD 1-1 include landscape and enemies, which are shown in Figure 8.

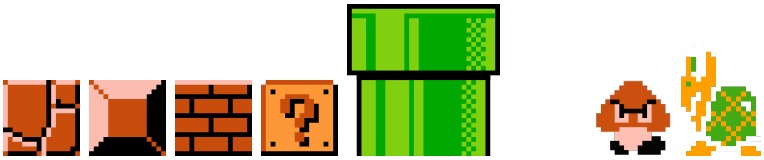


FIGURE 8. Landscape tiles (ground, bumper, brick, question block, pipe) and enemy sprites (goomba and green koopa) in WORLD 1-1.

Each tile in the input grid is encoded by a number in the interval $[-1, 1]$ based on the map in Table 1. If an enemy straddles a gridline, it is mapped to where most of its sprite is located.

Tile Class	**Input Value**
Landscape tiles	1
Enemies	-1
Empty	0

TABLE 1. How tiles from the receptive field are mapped to neural network inputs. Tiles below the ground are also mapped to 0.

3.2.2. *Network Outputs.* At each frame the neural network produces six real number output values each of which corresponds to a button on the controller: LEFT, RIGHT, UP, DOWN,

B, or A. Each controller button is in a boolean state (ON or OFF) and so we map each output x to a boolean by the map

$$x \mapsto \begin{cases} \text{OFF}, & \text{if } x \in (-\infty, 0] \\ \text{ON}, & \text{if } x \in (0, \infty) \end{cases}.$$

Note that the joypad on the NES does not allow UP and DOWN nor LEFT and RIGHT to be ON at the same time.

3.2.3. *GUI.* To see the neural network while trials are active we created a visualization of the network in the emulator using Lua. The GUI is shown in Figure 9

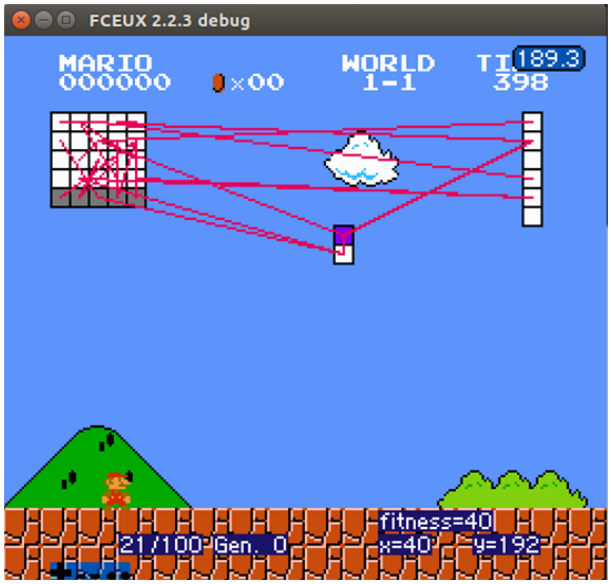


FIGURE 9. Running a trial in the FCEUX emulator. The neural network GUI shows the receptive field, intermediate neurons and connections and boolean controller outputs.

The GUI represents different types of neurons with the symbols in Table 2.

GUI Symbol	Neuron Type	Function
	Identity	$x \mapsto x$
	Sigmoid	$x \mapsto 1/(1+e^{-x})$
	Random	$x \mapsto x + \text{rand}()$
	Threshold	$x \mapsto x$ if $x \geq c$
	Control	*(Activation)*

TABLE 2. Types of neurons allowed in the unified neural model and their symbol in the GUI. Control neurons use an activation threshold.

Having a graphical representation of the network makes it easier to see how each strategy is interacting with the game.

3.3. **Fitness Testing.** Our fitness function is simply how far forward a strategy travels in the x-direction measured in pixels. If a strategy does not move in the x-direction for 120 frames (2 seconds), then its fitness test ends and we move on to the next individual.

3.4. **Evolution Parameters.** The parameters for SUNA are discussed in Section 2. We use the default parameters, listed in Table 3.

Parameter	**Value**
Number of initial mutations	200
Number of step mutations	5
Population size	100
Maximum novelty map population	20
Mutation probability array	{0.01, 0.01, 0.49, 0.49}
Neuromodulation probability	0.1
Control neuron probability	0.2
Excitation threshold	0.0

TABLE 3. Parameters for SUNA.

4. Experiments and Results

We conducted several experiments using SUNA to evolve neural networks to play SMB. Specifically, we trained strategies on different sets of levels and tested high-performing strategies on other levels for validation.

During fitness testing, a trial ended if one of the following conditions was met:

(1) Mario's x-position did not change after 120 frames (about 2 seconds).
(2) Mario died by hitting an enemy or falling in a pit.
(3) Mario reached the end of the level. In this case, training was ended at the end of the current generation.

4.1. **Training on WORLD 1-1.** We started by training strategies on WORLD 1-1, the first level of the game.[2] As explained earlier, a strategy's fitness was evaluated by spawning at the start of the level and seeing how far the strategy made it to the left. The radius of receptive field was fixed at $r = 2$.

After 722 generations, a strategy emerged that finished WORLD 1-1, which ended training. The fitness scores recorded from this trial are summarized in Figure 10. The strategy that completed WORLD 1-1 has 86 connections and 4 neurons aside from input and output neurons.

[2]Video of this trial can be found at `youtu.be/nBSbmBE2kKA`.

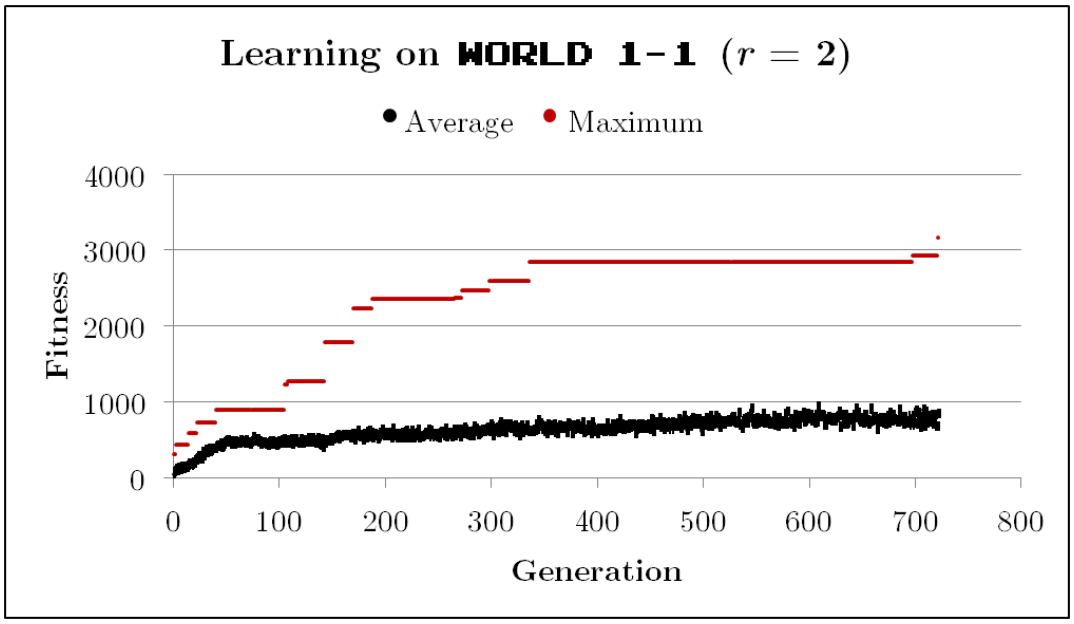


FIGURE 10. The average and maximum fitness score achieved in each generation during training on WORLD 1-1.

Notice that the the fitness scores of the best performing members of each generation form plateaus. These local maxima reflect several challenging parts of the level, shown in Figure 11.

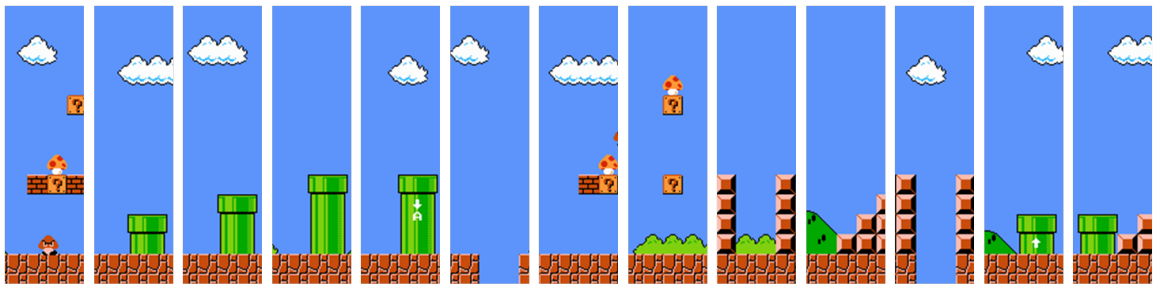


FIGURE 11. Local maxima encountered in WORLD 1-1. Common obstacles include: jumping over pipes: (b), (c), (d), (e), (l); overcoming enemies: (a), (g), (h) and avoiding pits: (f), (k).

4.2. **Portability.** To test the effectiveness of the strategy that beat WORLD 1-1 at playing the game more generally, we had the strategy attempt to play all 32 levels in SMB. The strategy consistently beat WORLD 1-1, but did not complete any other levels in the game. The best performances on other levels are listed in Table 4.

Level	x-distance
WORLD 8-4	1304
WORLD 3-2	946
WORLD 8-1	945
WORLD 3-1	898
WORLD 4-1	671

TABLE 4. Validation results: the best performances on other levels in the game by the winning strategy. The average performance over all levels besides WORLD 1-1 was 524.

Interestingly the strategy performed best on WORLD 8-4, the last level of the game. However, this level starts with a long corridor with few enemies for Mario to dodge. The strategy seemed to be well-suited to play levels that started without tall objects to jump over. The strategy's worst performance was on WORLD 2-2 and WORLD 7-2. These are water levels and require the player to use a completely different gameplay mechanic than the strategy learned while training on WORLD 1-1.

4.3. **Training on Multiple Levels.** In the second trial we used a larger set of training data. At the start of each fitness test a strategy spawned into a random level chosen from the following: WORLD 1-1, WORLD 1-2, WORLD 3-2 and WORLD 5-1. These levels were chosen since they are relatively easy and have different enemy placements. We trained strategies on multiple levels in hopes of evolving strategies that are more successful at playing arbitrary levels of the game. We set the radius of the receptive field at $r = 5$ to give strategies a better chance at using input to play the game. Fitness data for this trial is shown in Figure 12.

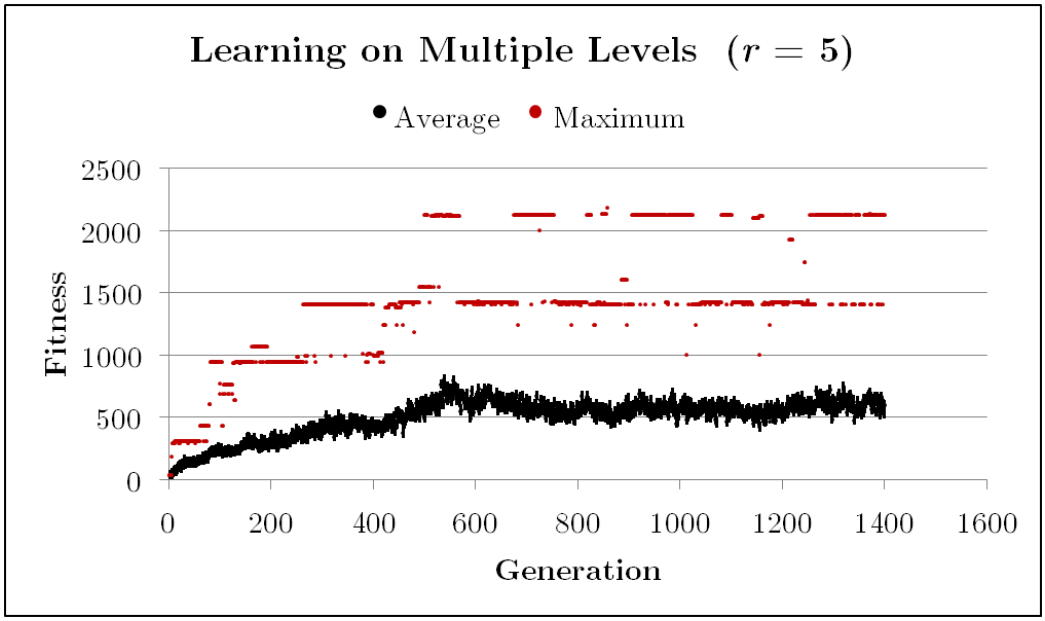


FIGURE 12. The average and maximum fitness score achieved in each generation during training on multiple levels.

After 1400 trials, no strategy was to able to complete any of the training levels. Furthermore, the fitness function staggered about a local maximum and so training was ended.

To see how well strategies from this trial perform on other levels, we tested the best strategy from the last generation on the other 29 levels of the game not used during training. The average fitness score was 467. However, the best strategy in Section 4.1 averaged 525 on this same set of levels. Unfortunately this indicates that the best strategy from this section was not as successful at adapting to other levels as the strategy trained solely on WORLD 1-1. This could be attributed to the fact that for a strategy to be successful in this scheme it had to be able to perform well on a random training level. Hence, strategies that perform well on some, but not all, of the levels may not survive if they have to play a more difficult level. Also, during training on multiple levels a persistent local maximum was encountered that the algorithm had difficulty overcoming.

5. Discussion

Here we discuss some strengths and weakness of how we used SUNA to play *Super Mario Bros.* and ideas for further work.

5.1. **The Input Model.** The receptive field is a simple and effective way to encode the visual elements of the game to use as input for a neural network. However, a receptive field of radius r has $(2r+1)^2$ tiles, meaning that the number of input neurons is quadratic in r. Hence, the space of networks widens as the receptive field is made larger and so it becomes more difficult for successful solutions to emerge. This suggests that a trade-off of having a finer representation of game objects could be that the algorithm will take longer to find a satisfactory solution. To test this hypothesis we could vary the radius of the receptive field and see how quickly the algorithm produces satisfactory results. A similar analysis could be done to study the effects of shifting the receptive field so it is not centered on Mario.

There are also many possible refinements that could be made to the receptive field to create a better representation of visual elements of the game. For example we could make finer distinctions between how tiles map into the neural network by specifying different types of blocks and enemies. Notably, since our training levels had few coins, our input scheme did not distinguish between coins and blocks. Mapping tiles with coins to a different value would allow strategies to recognize coins in other levels. There are also receptive fields in the literature that use different size blocks to model input for *Mario* games [3]. Near Mario the blocks are smaller to make fine distinctions between objects and farther way the blocks are bigger and so less information is encoded.

Furthermore, the input model could be extended to include values that describe Mario's current state. For example, Mario's current velocity in the x and y directions could be used to indicate how Mario is moving at each frame. (This idea comes from YouTube user SethBling, who has Mario's velocities commented out in his source code [10].)

5.2. **The Algorithm.** We choose to use the default SUNA parameters in our setup. However, the parameters of the model could likely be perturbed to increase of the speed of convergence. Notably, the default probability that a neuron will be added to a network during the reproduction phase is only 0.01. Increasing this rate could encourage more diverse networks to appear, perhaps helping the algorithm overcome local maxima. Furthermore, it could be advantageous to rig members of initial population in a meaningful way to ensure that every output has an incoming connection, for example. (For a discussion about choosing an initial population for an evolutionary algorithm, see [17].)

We also did not modify the C++ implementation of SUNA in any way to play SMB. However, the relatively long length of time that it took to complete the end of WORLD 1-1 could suggest that the algorithm would be more effective a if it changed during the evolution process. For example, more random mutations could take place after realizing that fitness scores have been stagnant.

We can also ask what population size is best for this problem. There are numerous case studies on this problem for evolutionary algorithms, including [18, 19]. However, the infamous "No Free Lunch Theorem" ensures that there is no best choice of parameters in an optimization algorithm [20].

5.3. **Gameplay Mechanics.** One notable aspect of gameplay that no neural network recognized is that Mario can go down some pipes to secret areas. These pipes appear about once in any particular level. (The pipe in FIgure 11e is the only such pipe in WORLD 1-1.). It makes sense that these pipes went unnoticed considering that using pipes mechanic is very different from the "platforming" tasks such as running and jumping in most other levels. By modifying the fitness function strategies could be incentivized to go down pipes but it seems unlikely that an individual would learn to push DOWN at the correct moment to go down one of these pipes.

There are also two (very similar) water levels in SMB, which represent a completely different style of gameplay. We did not train any strategies on water levels, but it would be interesting to see if some scheme could evolve a strategy that learns how to play both water levels and other levels.

5.4. **Future Work.** Future work could focus on refining what inputs are presented to the neural network and changing the parameters of the model, both of which are described above.

We could also implement a more sophisticated fitness function to test strategies. Current approaches have usually consider how far Mario is able to move to the right before stopping or dying. While the ultimate goal of the game is simply to finish, it would be interesting to also consider other factors important to human players. These include: how many coins and extra lives Mario has, how quickly Mario moves through a level, how many enemies Mario kills and Mario's power-up state. (Similar approaches are used in [3].) Perhaps implementing some linear combination of these variables as the fitness function to encourage "meta-goals" would help strategies learn quicker and produce more effective results.

In Section 4.3 we explained how we trained strategies by subjecting them to a random training level during fitness testing. It would also be interesting to average a strategy's performance over all training levels as a composite fitness score. This would allow a strategy to train on multiple levels and without contending with the randomness of the scheme in Section 4.3. We could compare the performance of these two schemes to see which one evolves more robust Mario-playing strategies.

Acknowledgments

This paper is part of the author's Senior Seminar project in partial fulfillment of the Computer Science major at Ripon College. The author thanks Dr. William Retert, Dr. McKenize Lamb and members of the TASvideos forum (`tasvideos.org/forum`) for useful discussions. Map data used in Figure 11 is from Ian Albert's map website [21].

References

[1] Adam Summerville and Michael Mateas. Super Mario as a string: Platformer level generation via lstms. *CoRR*, abs/1603.00930, 2016.

[2] Noor Shaker, Georgios Yannakakis, and Julian Togelius. *Towards automatic personalized content generation for platform games*, pages 63–68. 2010.

[3] Lars Solvoll Tnder and Ole-Petter Olsen. Multi-objective neuroevolution in Super Mario Bros. *Institutt for datateknikk og informasjonsvitenskap*, 2013.

[4] Julian Togelius, Sergey Karakovskiy, Jan Koutnik, and Jrgen Schmidhuber. Super Mario evolution. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 156–161, 2009.

[5] N. C. Hou, N. S. Hong, C. K. On, and J. Teo. Infinite Mario Bross AI using genetic algorithm. In *Sustainable Utilization and Development in Engineering and Technology, 2011 IEEE Conference on*, pages 85–89, Oct 2011.

[6] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):54–67, March 2010.

[7] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N. Yannakakis. The Mario AI championship 2009-2012. *AI Magazine*, 2013.

[8] Tom Murphy. The first level of Super Mario Bros. is easy with lexicographic orderings and time travel. *The Association for Computational Heresy (SIGBOVIK)*, April 2013.

[9] Michael Roberts. Super Mario Bros. - neural network with genetic algorithm. YouTube, https://youtu.be/05rEefXlmhI., 2015.

[10] SethBling. MarI/O - machine learning for video games. YouTube, https://youtu.be/qv6UVOQ0F44., 2015.

[11] Jordan Pearson. Why artificial intelligence researchers love 'Super Mario Bros.', October 2015.

[12] Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. 2016.

[13] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *CoRR*, abs/1410.7326, 2014.

[14] Carlos Gershenson. Artificial neural networks for beginners. *CoRR*, cs.NE/0308031, 2003.

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

[16] D. V. Vargas and J. Murata. Spectrum-diverse neuroevolution with unified neural models. *IEEE Transactions on Neural Networks and Learning Systems*, PP(99):1–15, 2016.

[17] Pedro A Diaz-Gomez and Dean F Hougen. Initial population for genetic algorithms: A metric approach. In *GEM*, pages 43–49, 2007.

[18] Stanley Gotshall Bart Rylander. Optimal population size and the genetic algorithm. *Population*, 100(400):900, 2002.

[19] Ting Hu, Simon Harding, and Wolfgang Banzhaf. Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2):205–225, 2010.

[20] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), April 1997.

[21] Ian Albert. Super Mario Bros. maps. http://ian-albert.com/games/super_mario_bros_maps/.