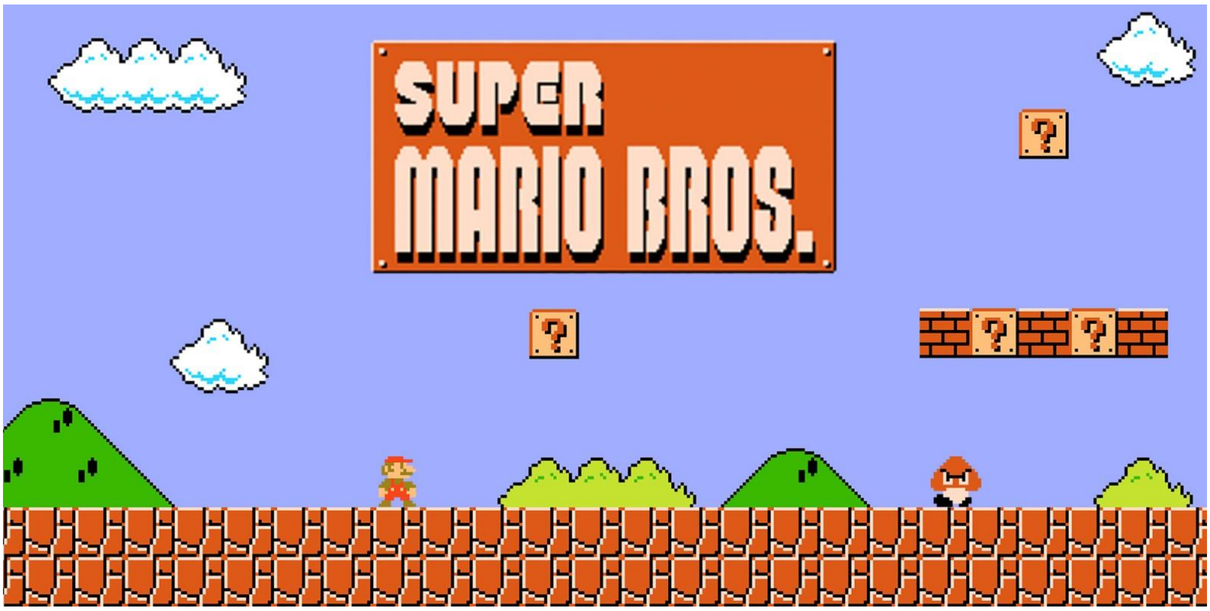
Can an AI learn to play Super Mario bros. for the NES?

Owen Crucefix

Candidate Number: 9479

Centre Number: 64395

Godalming College



Contents

Table of Figures

Research

Introduction

The aim of this investigation is to build an AI that can interact with a Super Mario Bros. game and find out whether or not it can learn to play the game successfully. This investigation was initially requested Alex Turner who has a long interest in Super Mario Bros. and knows how simple the game is and wanted to know if an AI could learn to play the game.

What needs to be researched?

I will be investigating whether or not an AI can learn to play Super Mario Bros so I need to know what Super Mario Bros is, how the game works and what kind of AI should be built to play it.

What is Super Mario Bros?

Super Mario Bros was a video game developed by Nintendo and launched for the Nintendo Entertainment System (NES) (also known as The Famicon in Japan) in 1985.

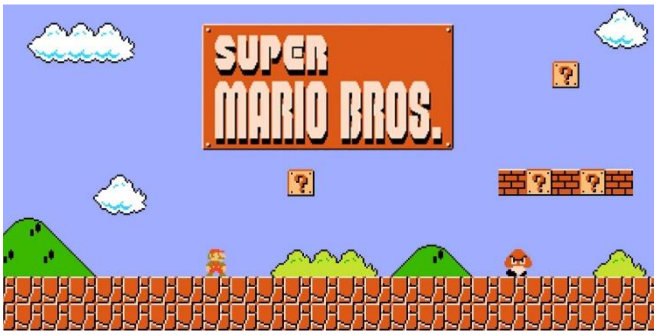


Figure 1 Promo Image of Super Mario Bros

In the game you play as Mario who must make it from one end of the stage to the other. Along the way you must overcome many obstacles ranging from platforming jumps to different enemies. There are also collectables and power-ups Mario can collect to help him beat each stage that can either give him more hits before death or increase the amount of lives Mario has, if Mario runs out of lives the game ends. There are 32 levels split across 8 worlds with 4 levels each.

Mario’s Move set

Movement

Mario can walk in both to the left and to the right. Mario can also run in these directions when holding down the run button.

Jumping

Mario’s main ability is the ability to jump, he can jump a variable height depending on how long the jump button is held down for. Mario’s jump is used to climb over obstacles, stomp on enemies and also to hit blocks in order to break them or gain an item from them.

Items/Powerups

Throughout Super Mario Bros. there are multiple different powerups and items. The main two are the Super Mushroom; which makes Mario larger, allows him to break blocks and lets him take one more hit before dying, and the Fire Flower, which does everything the Super Mushroom does, but also allows Mario to shoot fireballs at enemies by pushing the run button. There is also the Starman which allows Mario to become invincible so he can't be hurt by enemies.

What types of neural networks are there?

There are 3 main options for the type of neural networks that can be used: **Artificial Neural Network**, **Convolutional Neural Network** and **Recurrent Neural Network**

Artificial Neural Network

Artificial Neural Networks (ANN) are a type of network that is modelled after the human brain. It is comprised of neurons and neurons. It learns from past data in order to improve the network and give predictions. It is most commonly used for random function approximation, where it randomly applies a function to data but slowly learns by calculating errors between the predicated and actual results in order to produce the predicated result.

Artificial Neural networks are built up of three types of layers. The first layer is the input layer, this will receive data through various neurons that will then send the data through connected neurons to the next layer. The second type of layer is the hidden layer, there can be multiple hidden layers to a neural network all connected to each other in order. These layers will then perform various mathematical operations to the inputs and recognise the patterns that might be formed. Finally there is the output layer, this layer is just for returning the final output from the system. neuron

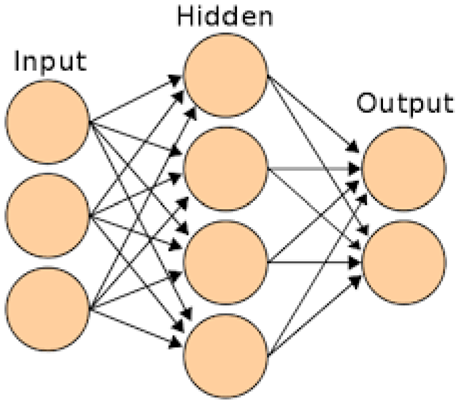


Figure 2 Example of Layers in an ANN

Each neuron in the system will have different weights, these weights specify how important each neuron is. A Transfer function will calculate the total weight of the input by adding up all the weights of the neurons that received an input.

Finally a method called Backpropagation is used to calculate what neurons might have contributed to the error the most and adjust their weights accordingly.

Convolutional Neural Network

Convolution Neural Networks (CNN) are a type of deep learning algorithm that are used for image recognition. They work by taking an image as an input and then learning the different features of the images using certain filters. These filters perform dot products on the image in order to break it down and find patterns.

Recurrent Neural Network

Recurrent Neural Networks (RNN) work over a period of time and are used to predict the next input based on the previous one. The general structure involves receiving the first input and applying it to the first layer in the network. Then the output from this layer is used as the input for the next state. Each layer is exactly the same so the all have the same weight. At the end all the information is joined together and the error is calculated from the actual result and the predicted result.

What types of Algorithms can neural networks use?

Gradient Descent

The idea behind a Gradient Descent algorithm is to minimise the error in each parameter. This is done by iterating over a parameter and slowly moving down the gradient of error in order to find the lowest point or the error where there is the least error. Each time the function iterates it calculates the new error and works out what the gradient of the error line is now based on the previous results. It then uses this to adjust the parameters and the direction in which the training will continue. It will stop once the error has reached a low enough point known as the stopping criteria. This algorithm is very useful for larger neural networks as it only needs to store gradient vectors for each parameter instead of storing the entire matrix of information. The downside to this algorithm is that it is very slow for long, narrow structures of error as it will take more iterations to find the actual minimum value.

Over time gradient descent has been improved with new algorithms such as the Quasi-Newton method, which builds up approximations for matrices composed of second derivatives of the loss function in order to approximate the smallest error margin, but it still has the same concept of the Gradient Descent algorithm which is to adjust parameters according to how large the error is and improve in the direction of the smallest error.

Evolutionary/Genetic Algorithm

An evolutionary algorithm is based off of the biological concept of "Survival of the fittest" and it works by having a population of networks that are randomly generated and have them perform a task. A fitness score is then calculated based on how well each network performed the task. The networks are then ranked from highest scoring to lowest scoring. Then a new generation of networks is created by taking two or more networks with similar scores and "breeding" them to create a network made up of the neurons from the parents. Mutation is then applied to some of the networks where they will receive or lose random additional neurons. The cycle will then repeat again with this generation until a network is produced that can perform the task nearly perfectly. Often the top scoring network of a generation will be in the next generation in case no new networks can out-perform it so the highest score for the generation is at least as good as the last generation. Evolutionary Neural Networks are very useful for more complicated problems that might have a changing environment as the network can adapt to its environment and change without the need for large amounts of new training data. The downside to this type of algorithm is that it is very slow to train as it requires the problem to be executed several times for each generation.

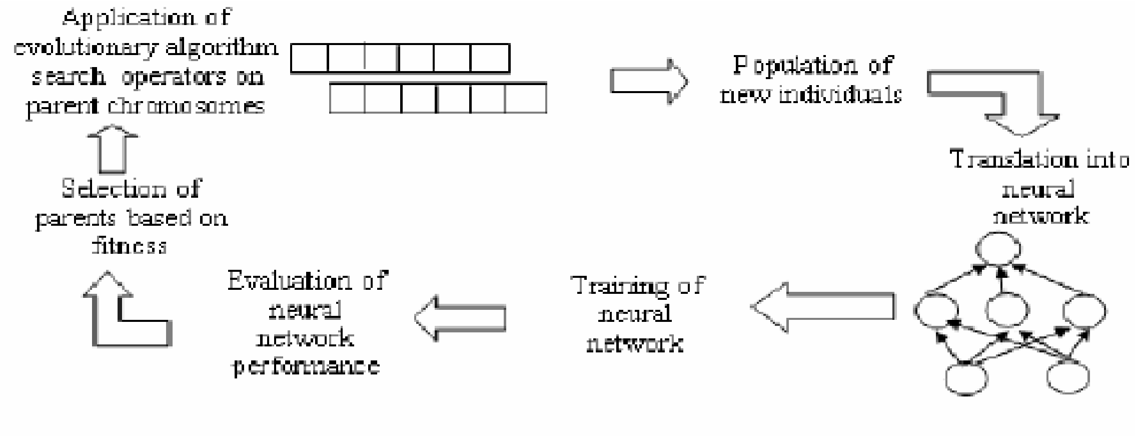

Application of evolutionary algorithm search operators on parent chromosomes
Population of new individuals
Translation into neural network
Training of neural network
Evaluation of neural network performance
Selection of parents based on fitness

Figure 3 Basic algorithm showing how an evolutionary neural network works

How will the problem be emulated/What programming language should be used?

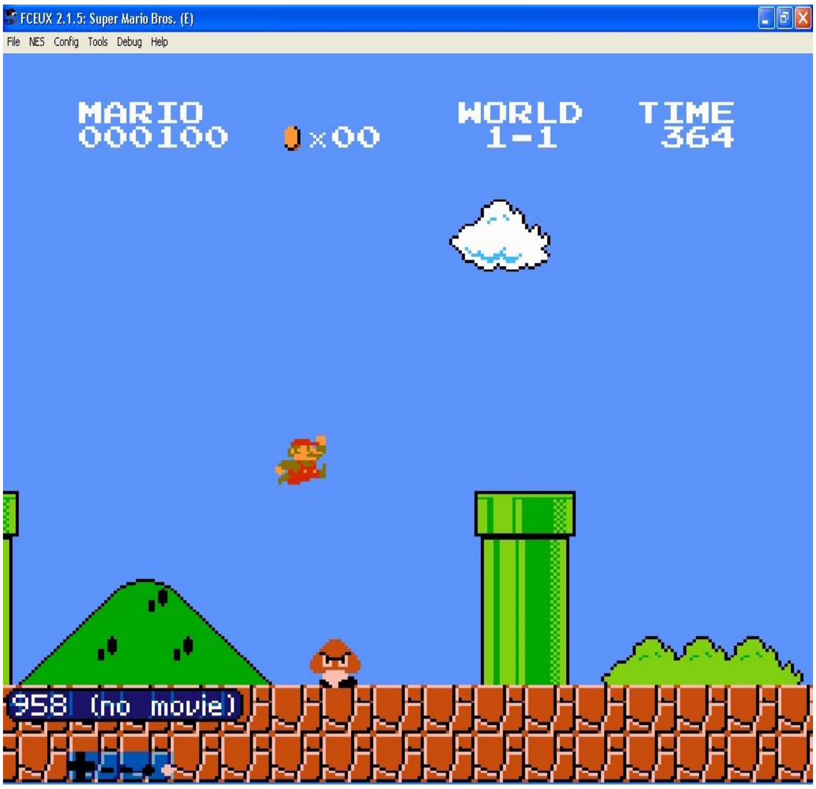
For emulating the problem, I have two options:

Image Recognition/Python

Use image recognition on a game of Super Mario Bros and send inputs to a controller when certain objects are detected on screen. This could be done easily using Python because there are libraries (for instance OpenCV) that can be used to detect different colours in an image and give the neural network an idea as to what the level looks like. This would make it easier to build up an image of everything on the screen at once without having to worry about creating a data structure to store the image as it can just be stored as colour values. Python would also be a great language to use as it supports Object-Oriented programming which would be a great way to handle the problem. The downside to this method is that certain objects in the game have very similar colours so the neural network might not be able to tell the difference between them, for example a Goomba (enemy) has a very similar colour to brick block so the neural network might not see Goombas as a threat or it might start to see brick blocks as enemies and try to avoid them. The other downside is that there might be input lag between the Python script and the controller so inputs might be late.

Using an Emulator/Lua

Using an emulator and a ROM file for the game I can have direct access to the RAM for a game of Super Mario Bros. My chosen emulator would be FCEUX because it allows for custom Lua scripts to be run and is well documented for the different commands that can be used by the emulator. Lua is an easy to use language which while does not explicitly support Object-Oriented programming can simulate objects and classes to the same effect. If I used an emulator it would make it really easy to access all of the data in the game and could create an image of the environment with a 100% accuracy as it wouldn't depend on any image recognition. The downside to this is that I would have to create a class and method to extract the data from the RAM and build the map to use which would take more time. One

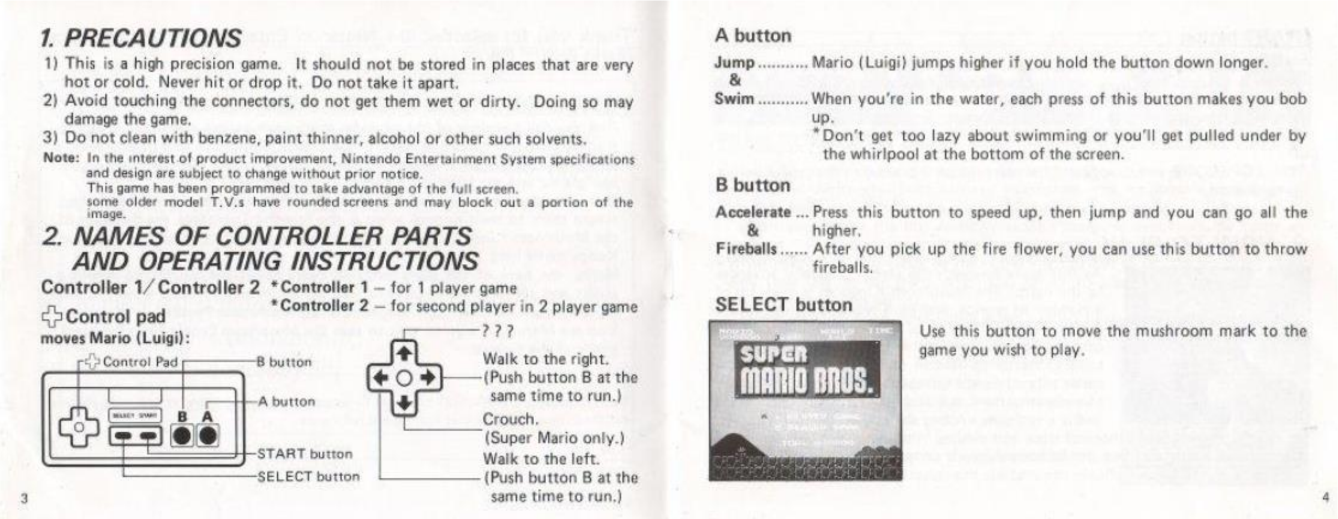

FCEUX 2.1.5: Super Mario Bros. (E)
File NES Config Tools Debug Help
MARIO
000100
×00
WORLD
1-1
TIME
364
958 (no movie)

other downside to Lua is that it’s random function doesn’t always work and that would mean building my own random function from the ground up, however this would not take too long to do as random functions are quite simple.

Figure 4 Example Image of FCEUX running Super Mario Bros

Documents

Super Mario Bros Instruction Manual (US Version)



1. PRECAUTIONS

1) This is a high precision game. It should not be stored in places that are very hot or cold. Never hit or drop it. Do not take it apart.
2) Avoid touching the connectors, do not get them wet or dirty. Doing so may damage the game.
3) Do not clean with benzene, paint thinner, alcohol or other such solvents.

Note: In the interest of product improvement, Nintendo Entertainment System specifications and design are subject to change without prior notice.
This game has been programmed to take advantage of the full screen.
some older model T.V.s have rounded screens and may block out a portion of the image.

2. NAMES OF CONTROLLER PARTS AND OPERATING INSTRUCTIONS

Controller 1/Controller 2 ***Controller 1** – for 1 player game
***Controller 2** – for second player in 2 player game

✚ Control pad
moves Mario (Luigi):

? ? ?
Walk to the right.
(Push button B at the same time to run.)
Crouch.
(Super Mario only.)
Walk to the left.
(Push button B at the same time to run.)

3

A button

Jump .... Mario (Luigi) jumps higher if you hold the button down longer.
&
Swim .... When you're in the water, each press of this button makes you bob up.
*Don't get too lazy about swimming or you'll get pulled under by the whirlpool at the bottom of the screen.

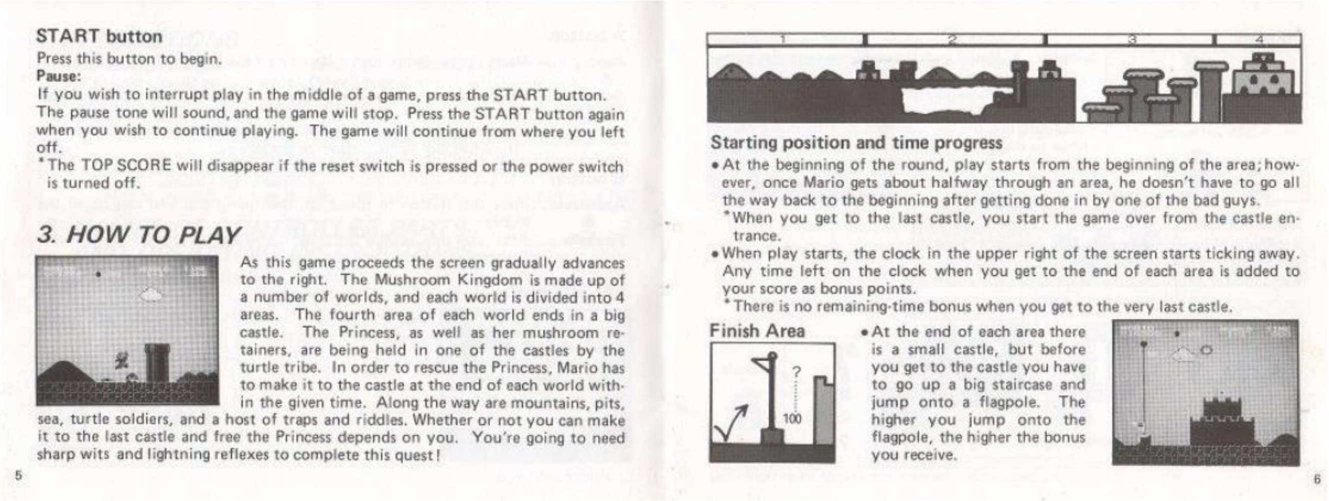
B button

Accelerate ... Press this button to speed up, then jump and you can go all the higher.
&
Fireballs ...... After you pick up the fire flower, you can use this button to throw fireballs.

SELECT button

Use this button to move the mushroom mark to the game you wish to play.

4



START button

Press this button to begin.
Pause:
If you wish to interrupt play in the middle of a game, press the START button. The pause tone will sound, and the game will stop. Press the START button again when you wish to continue playing. The game will continue from where you left off.
*The TOP SCORE will disappear if the reset switch is pressed or the power switch is turned off.

3. HOW TO PLAY

As this game proceeds the screen gradually advances to the right. The Mushroom Kingdom is made up of a number of worlds, and each world is divided into 4 areas. The fourth area of each world ends in a big castle. The Princess, as well as her mushroom retainers, are being held in one of the castles by the turtle tribe. In order to rescue the Princess, Mario has to make it to the castle at the end of each world within the given time. Along the way are mountains, pits, sea, turtle soldiers, and a host of traps and riddles. Whether or not you can make it to the last castle and free the Princess depends on you. You're going to need sharp wits and lightning reflexes to complete this quest!

5

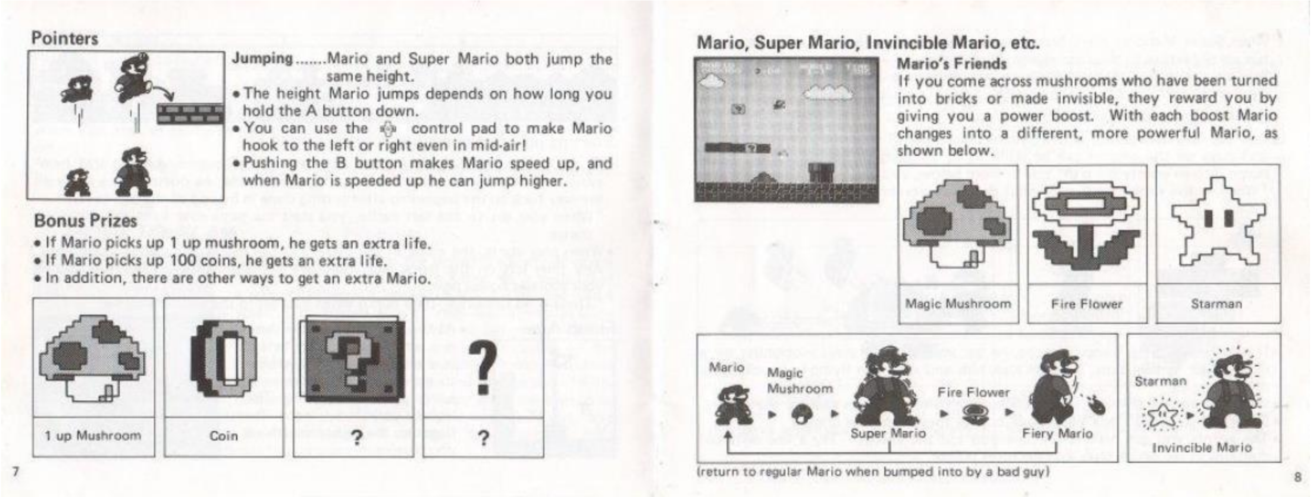
Starting position and time progress

- At the beginning of the round, play starts from the beginning of the area; however, once Mario gets about halfway through an area, he doesn't have to go all the way back to the beginning after getting done in by one of the bad guys.
  *When you get to the last castle, you start the game over from the castle entrance.
- When play starts, the clock in the upper right of the screen starts ticking away. Any time left on the clock when you get to the end of each area is added to your score as bonus points.
  *There is no remaining-time bonus when you get to the very last castle.

Finish Area

- At the end of each area there is a small castle, but before you get to the castle you have to go up a big staircase and jump onto a flagpole. The higher you jump onto the flagpole, the higher the bonus you receive.

6



Pointers

Jumping ....Mario and Super Mario both jump the same height.
- The height Mario jumps depends on how long you hold the A button down.
- You can use the ✚ control pad to make Mario hook to the left or right even in mid-air!
- Pushing the B button makes Mario speed up, and when Mario is speeded up he can jump higher.

Bonus Prizes

- If Mario picks up 1 up mushroom, he gets an extra life.
- If Mario picks up 100 coins, he gets an extra life.
- In addition, there are other ways to get an extra Mario.

7

Mario, Super Mario, Invincible Mario, etc.

Mario's Friends
If you come across mushrooms who have been turned into bricks or made invisible, they reward you by giving you a power boost. With each boost Mario changes into a different, more powerful Mario, as shown below.

(return to regular Mario when bumped into by a bad guy)

8

Table of Sources

Item	Source	Link/Reference
Super Mario Bros Promo Image	Official Nintendo Store page	https://www.nintendo.co.uk/Games/NES/Super-Mario-Bros--803853.html
ANN Diagram	Analytics Vidhya - How does Artificial Neural Network (ANN) algorithm work? Simplified! By Tavish Srivastava	https://www.analyticsvidhya.com/blog/2014/10/ann-work-simplified/
Super Mario Bros Instruction Manual	Original by Nintendo Images from Legendsoflocalization.com	https://legendsoflocalization.com/media/super-mario-bros/manuals/Super-Mario-Bros-Manual-US.pdf
FCEUX Documentation	FCEUX.com	http://www.fceux.com/web/help/fceux.html
FCEUX Example image	emulatorsplay.com	http://emulatorsplay.blogspot.com/2012/08/fceux-emulator-for-playing-nintendo.html
Evolutionary Neural	Electricity Price Forecasting	https://www.researchgate.net/publication/4315571_Electricity_Price_Forecasting_Using_Evolved_Neural_Networks

Network diagram	Using Evolved Neural Networks By D. Srinivasan	

Summary of research

From my research I have found out that Super Mario Bros is a simple game with easy to learn controls that rewards the player for collecting items and beating levels but punishes the player for not avoiding enemies or performing platforming correctly. I have also found out about several types of neural networks so that I can find a model that can solve the problem presented to me by Alex in order to get an AI to learn to play and beat Super Mario Bros. Another area I have researched is the method of emulation for the problem, I have identified two different methods and the programming languages that would be needed for both of them.

Analysis

What type of AI should be used?

From looking at the problem on the surface I have decided that an evolutionary algorithm would be best suited to the problem. This is because the environment constantly changes in Super Mario Bros so the AI will need to be able adapt to its environment easily. An evolutionary algorithm makes this easier because it doesn't have defined inputs and slowly evolves over time creating the inputs based on its environment.

What method of emulation should be used?

From my research I have decided that the system will be created using an emulator and written in Lua. This is because I believe it will be easier to create a map of the tile data directly from the game's RAM than to use some form of image recognition to detect what is on screen. I also believe that it won't take too long to create the map and as such that isn't much of a drawback. The fact that I will have access to the entire RAM will also allow for different options when calculating fitness and rewards for the network as I can use data such as the player's score, their exact position in a level and whether or not they have gained a powerup in order to give bonuses to a network. The emulator I will be using is FCEUX Version 2.2.3 for Windows and has a built-in compiler for Lua scripts along with a few built in functions for things like accessing memory and creating inputs. FCUEX also allows you to have control over every single frame meaning there will be no input lag for the controller and I will be able to analyse everything that happens in the game without missing any data.

Class diagram of a single neural network

When analysing the problem, I decided to make a small diagram to give an outline as to the components that would need to be created for a single neural network in the system.

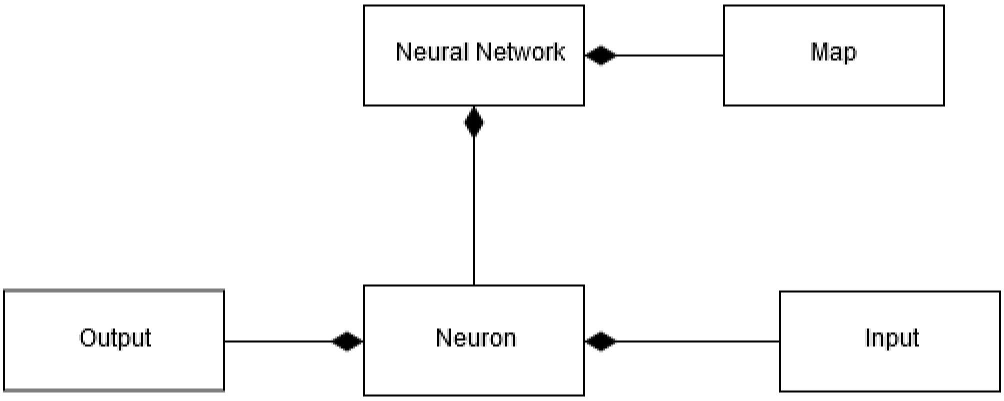


Figure 5 An outline of a class diagram for a neural network

The neural network would be composed of neurons and a map. The map will store the data for each of the current tiles on screen and will provide and easy way of locating what type of tile is in each coordinate. The neurons will have an input and an output, the input will have a coordinate and a tile type associated with it so that it can send information when its assigned tile type is found at its coordinates. The output will then refer to an input on the controller and when data is sent from the input in the neuron it will send an input to the controller.

After this analysis of the original diagram I decided that the input and the output might not need to be their own classes and could simply be properties of a neuron simplifying the class diagram to this:

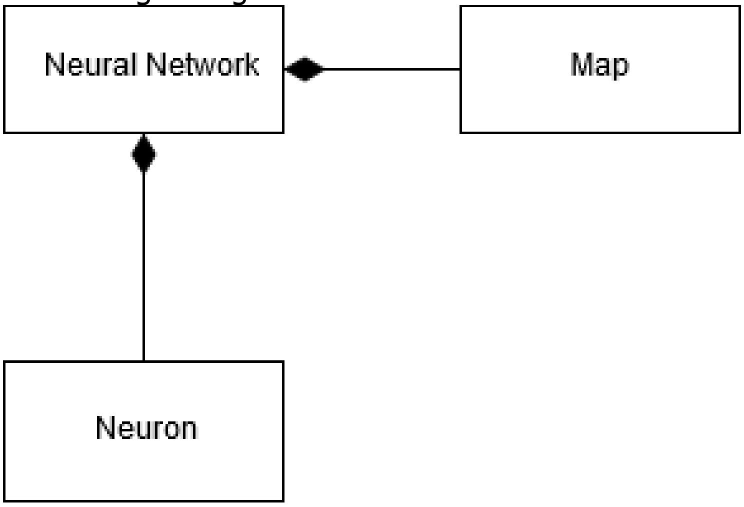


Figure 6 Simplified outline of a class diagram for a neural network

Data-Flow Diagram

In order to help understand how an evolutionary neural network would work for this system I have broken down how the data would be handled by the system in a data-flow diagram.

Level 0

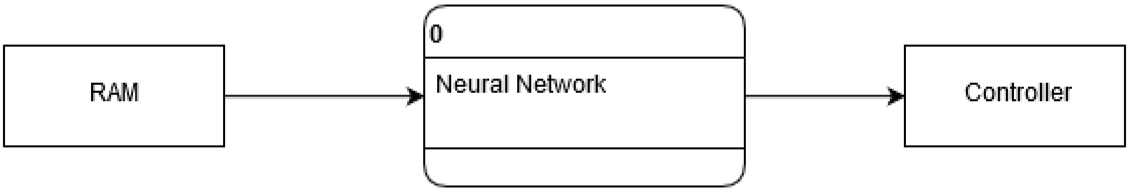


Figure 7 Level 0 Data Flow Diagram for the system

Level 1

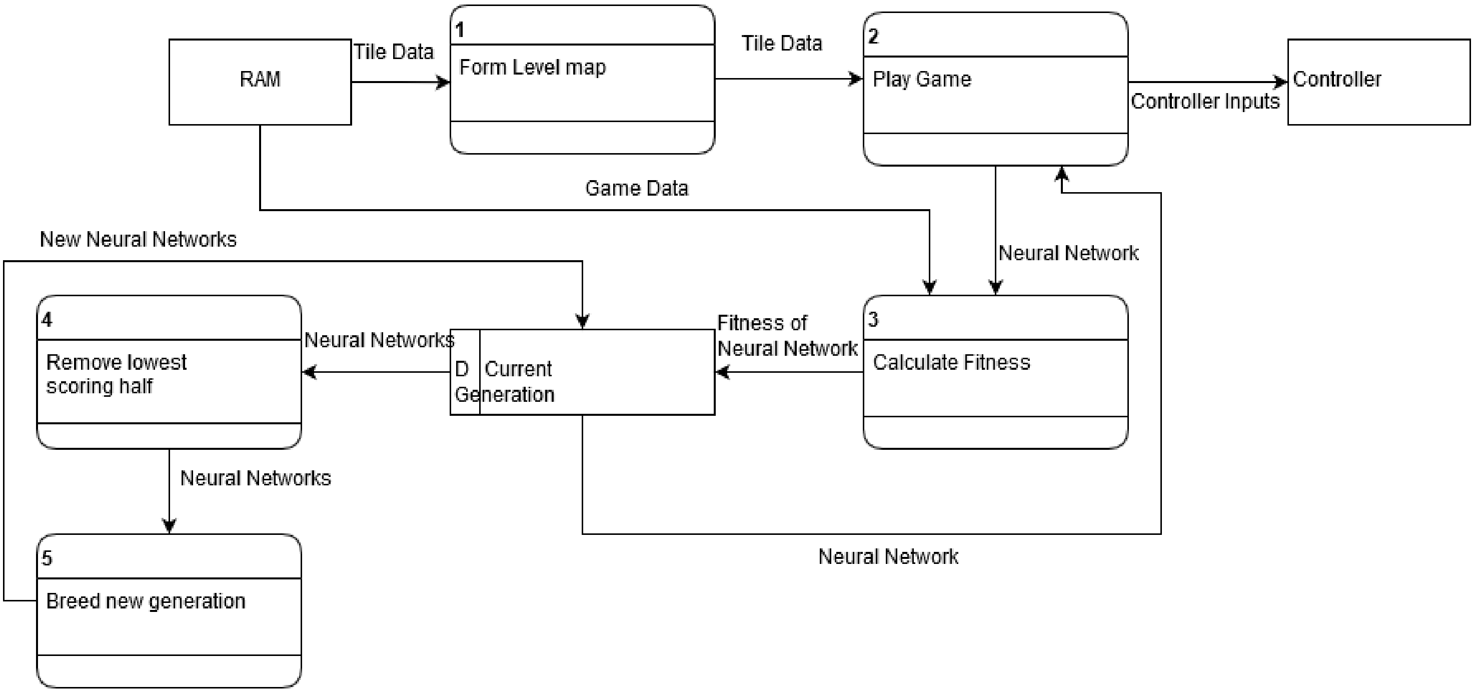


Figure 8 Level 1 Data Flow Diagram for the system

Volumes of Data

The system I will produce does not need to store much data. The population of the current generation of neural networks will need to be stored. This will be done by storing the coordinates of every neuron in the network in relationship to the map along with the type of input it is. The fitness value will also be stored with a neural network if it has already been calculated. This will allow easy access to view an individual network as well as allowing training to be handled over multiple sessions as it will save the current generation at the end of a session and will be able to load the same generation upon the next start-up.

Data Dictionary for what is stored

Reference	Name	Data Type	Length	Occurrence	Source
1	CoordinateX	Integer	0-32	1x per neuron	Neuron

2	CoordinateY	Integer	0-32	1x per neuron	Neuron
3	Output	String	6	1x per neuron	Neuron
4	TileType	Integer	1-2	1x per neuron	Neuron
5	Fitness	Integer	Unlimited	1x per network	Neural Network
6	NetworkID	Integer	0-29	1x per network	Neural Network

Flow Chart of Proposed System

Now that I know what data will be transferred and what needs to be stored and need to think about how the system will function. This can be easily outlined with a simple flow chart so I created a flow chart to show how the system would handle the first generation.

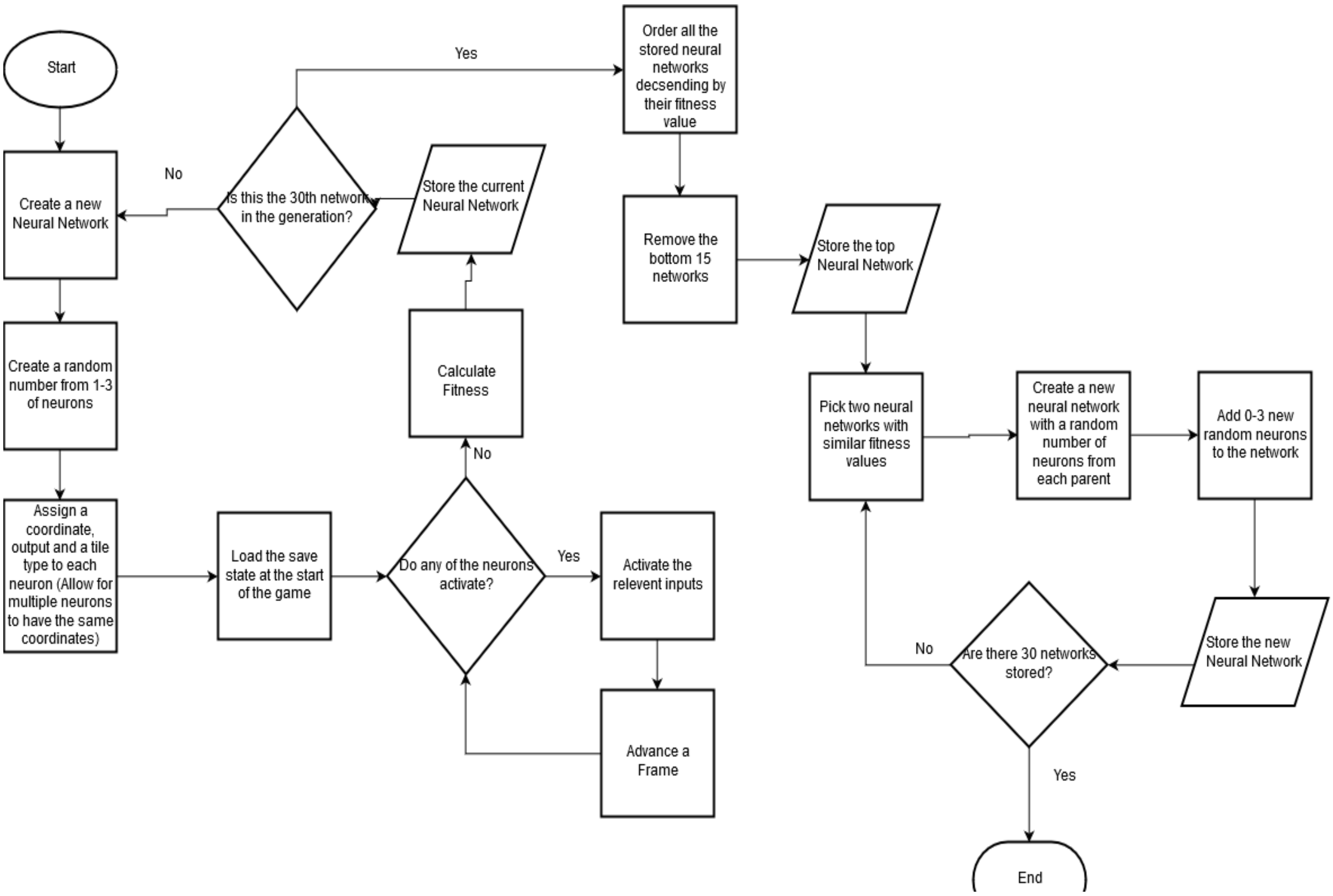

Start
Create a new Neural Network
Create a random number from 1-3 of neurons
Assign a coordinate, output and a tile type to each neuron (Allow for multiple neurons to have the same coordinates)
Load the save state at the start of the game
Do any of the neurons activate?
Yes
Activate the relevent inputs
Advance a Frame
No
Calculate Fitness
Store the current Neural Network
Is this the 30th network in the generation?
No
Yes
Order all the stored neural networks decsending by their fitness value
Remove the bottom 15 networks
Store the top Neural Network
Pick two neural networks with similar fitness values
Create a new neural network with a random number of neurons from each parent
Add 0-3 new random neurons to the network
Store the new Neural Network
Are there 30 networks stored?
No
Yes
End

Figure 9 Flow Chart of the first generation of neural networks in the system

OOP in Lua

Lua is not explicitly an OOP language however it does support the OOP paradigm through the use of tables and meta tables. It is very confusing syntax in order to set it up but once it is setup for a class it functions like any other OOP language. In order to help understand how the syntax works I will have an example of class instantiation below.

local ExClass = {Property1 = 0} –This creates the initial table that the class will be built off of with its properties listed inside.

ExClass.__index = ExClass – This line means if a value that is not in table is attempted to be accessed it will be redirected back to itself and create the entry.

Function ExClass.New() –Start of the constructor

> local self = setmetatable({},ExClass) –This is the line that does most of the magic; it creates a blank metatable or object that then inherits all of the properties and methods from the table ExClass and the assigns the keyword "self" to be used to access it.
>
> self.Property1 = 1 –Standard contructor code goes here making sure to use the self prefix to ensure you are changing the property of the object
>
> return self –Returns the metatable to the object

end –Closes the function

--Then for creating a method for the class

Function method1:() –Methods use a ":" instead of a "." That would be normally used because it means that the self parameter is automatically added so the metatable can be accessed.

> --method code

end

How the emulator interacts with the lua script

Because lua is a scripting language it is generally not made to be compiled but instead interpreted by a program that wants to run the script. The FCEUX emulator has a built in lua interpreter meaning the script will have direct access to the emulators memory and will be able to execute commands within the emulator.

Controller input

FCEUX allows you to perform inputs automatically through the use of input tables. An input table states all the possible inputs from a controller and whether or not they are being held down. In theory this allows for inputs to be changed every single frame however in practice because the inputs need a frame to refresh, they can only be done every other frame. For example, A is held down on frame 1 but if A is held down on frame 2 it is not registered as a new input therefore nothing changes, however if I is not held down on frame 2 and then is held down on frame 3 it will register as two separate inputs.

Mario's movement capabilities

Jumping

Mario can change his jump height by holding down the jump button for longer. Through testing I have found out Mario can reach his maximum jump height of 67 pixels after 29 frames of uaholding down the button.

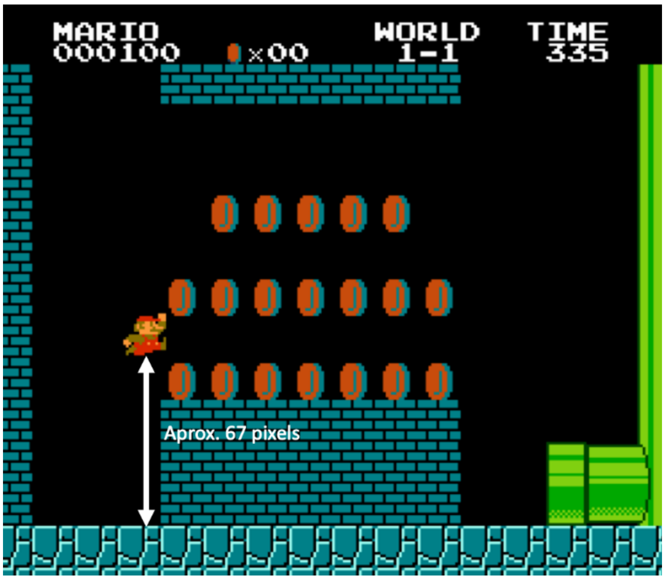


Figure 10 Example image of Mario's maximum jump height (Actual measurements were taken using the game's memory location for height)

Running

Mario can run by holding down the run button and a direction but he has to accelerate to his maximum speed. Through testing I have discovered that it takes Mario 18 frames to accelerate to his maximum speed.

List of Requirements

0. Form a map of the level

0.1. Read the RAM for tile data

0.2. Form an ordered array from the tile data

0.3. Make the map move as Mario moves

1. Create neural network

1.1. Create Neurons with an input, made of coordinates and a tile type, and an output

1.2. Store data for a neural network as a single string in a uniform fashion to make it easier to read when accessing previous networks

1.3. Interpret the data string for a neural network

2. Evolve Generation

2.1. Create an ordered list of all neural networks for the current generation based on fitness values

2.2. Remove the bottom 50% scoring neural networks

2.3. Keep the top scoring neural network for the next generation

2.4. Breed remaining neural networks until the desired number of networks for the next generation exist

2.4.1 Pick out two neural networks that will be used as parents

2.4.2 Pick random neurons from the two networks for the new network

2.4.3 Mutate the new network by adding a random amount of new neurons

3. Play Game

3.1. Load a pre-existing save state at the start of the selected level from a list

3.2. Check for any inputs from neurons

3.3. Perform inputs into the game

3.4. Check if the game is ready to end via either no inputs or Mario dying

3.5. Calculate Fitness value for a neural network based on the different scoring methods: Distance, Speed and Coins collected

4. Random Function

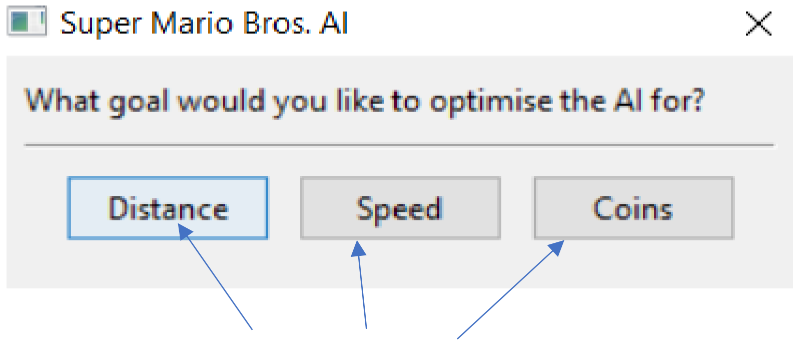
4.1. Random function generates a new seed every time it is used

4.2 Random function generates integers between an upper and lower bound

Design

User Interface

Upon Running the program, the following Interface should be displayed to the user

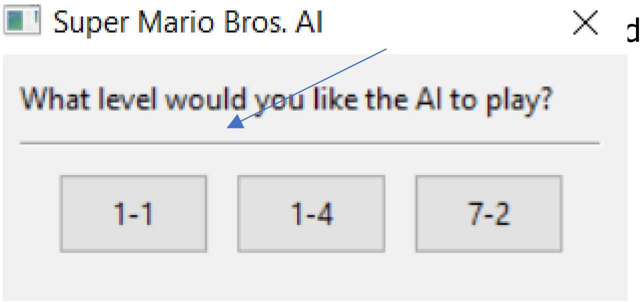


Super Mario Bros. AI

What goal would you like to optimise the AI for?

Depending on which button is chosen the method in which fitness is calculated will be changed

Once one of the options is selected the next displayed

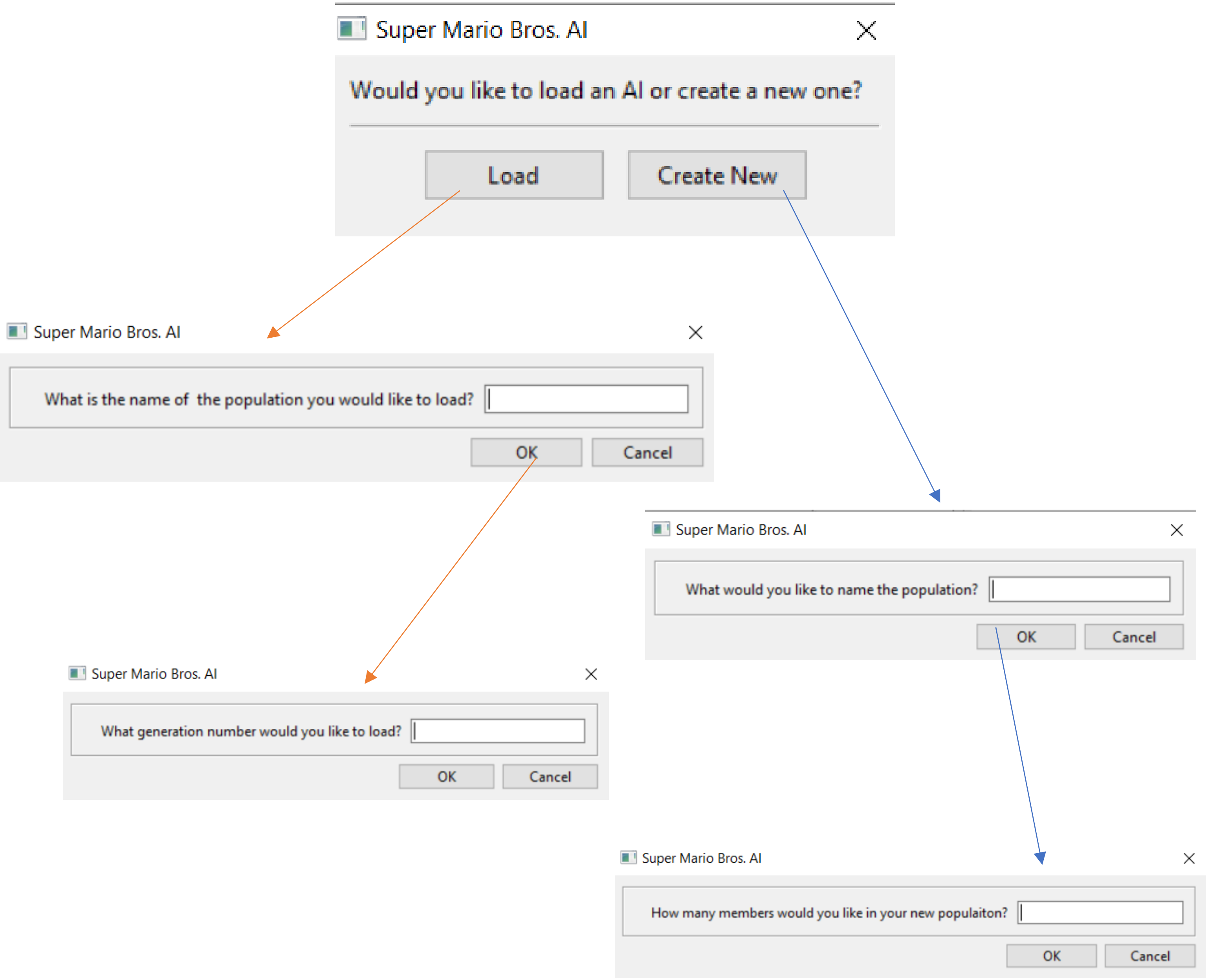


Super Mario Bros. AI

What level would you like the AI to play?

Depending on which button is chosen the level that will be loaded for the AI to play on will be changed

Once the parameters have been chosen the user should be prompted to either load a population or create a new one and will be asked for the required inputs for both and a cancel button to send them back to the menu.



Super Mario Bros. AI

Would you like to load an AI or create a new one?

Super Mario Bros. AI

What is the name of the population you would like to load?

Super Mario Bros. AI

What would you like to name the population?

Super Mario Bros. AI

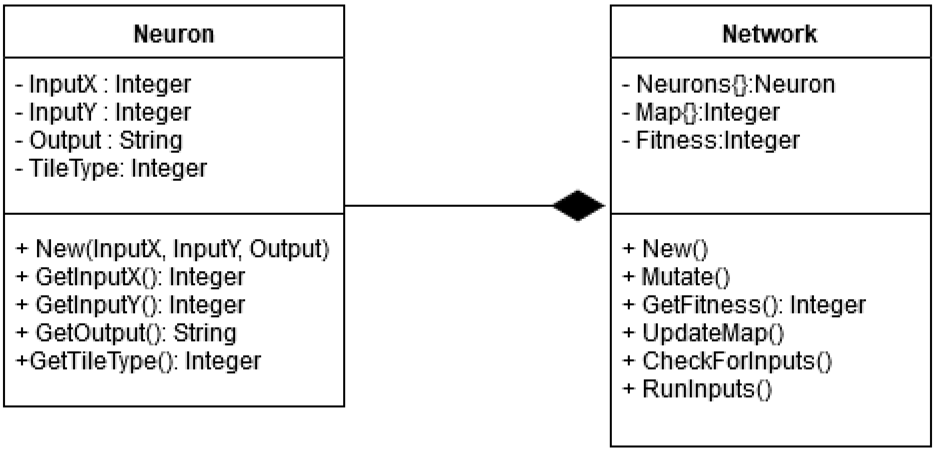
What generation number would you like to load?

Super Mario Bros. AI

How many members would you like in your new populaiton?

UML Class Diagram

After I started the design process, I realised that the map class would be redundant so I changed it to simply be a property of the Network Class.



FCEUX Functions

The FCEUX emulator that I'm using to run the game and the program comes with its own built in functions for managing the emulator.

The functions I will be using are listed below along with a description of what they do:

- Memory.ReadByte(MemoryAddress) – Returns an integer value from 0-255 that represents the data stored in RAM at the specified memory address
- Joypad.Write(Controller,InputTable) – Writes the inputs specified in the InputTable to the controller specified. Allows for spoofing inputs from a controller.
- Emu.FrameAdvance() – Advances the game to the next frame
- Gui.Box(X1,Y1,X2,Y2,Colour) – Draws a box on the UI from the coordinates (X1,Y1) to (X2,Y2) in the colour specified where Colour is a hex representation of a colour
- savestate.load(savestate.object(state slot)) – Loads a save state from the slot specified.

Data Storage

The data that needs to be stored by the program is:

- The data for each neural network
    - The Input and output of neurons
    - The fitness scores
- The name of the population
- The number of generations passed

The data will be stored in text files with the following format

The name of the population will be the file name

The first line of the file will be an integer which will refer to the number of generations that have passed for that population.

The second line will refer to the number of networks in the population

Each line after that will refer to a Network

A network will be stored as string of numbers.

The first 2 digits will refer to the number for neurons in the network.

Then all the neurons will be listed in order with 6 digits for each neuron.

The neurons are stored with the first 2 digits for the X coordinate of the input and the next 2 digits as the Y coordinate of input and the 5th digit refers to the "ID" of the output according to this table.

1	"A"
2	"B"
3	"Up"
4	"Down"
5	"Left"
6	"Right"

The 6th and final digit of a neuron will refer to the tile type that it detects, either a block/ground as a 1 or an empty space as a 2.

Then after the neurons the Fitness value will be stored as a 5 digit integer from 00000 – 99999.

For Example A network might be stored as: 401152113113203033105096100104

Which would represent a network with a fitness value of 104 and the following neurons:

InputX	InputY	Output	Tile Type
1	15	“B”	Block
13	11	“Up”	Empty
3	3	“Up”	Block
5	9	“Right”	Block

Whilst this method of storage might seem complicated at first it allows for an easy and compact storage system using a simple decoding algorithm:

```
File ← OPENFILE(PopulationName & ".txt","r")

GenerationNumber ← File.Read()

NoOfNetworks ← File.Read()

OutputTable ← ["A","B","Up","Down","Left","Right"]

FOR i = 1, NoOfNetworks DO

    Network ← File.Read()

    NoOfNeurons = Network[1] //Indexing from 1 as that's what lua does

    PosCounter ← 2

    For j = 1, NoOfNeurons DO

        CurrentGeneration[i].Neurons[j].InputX ←
        Network[PosCounter,PosCounter +1]

        PosCounter += 2

        CurrentGeneration[i].Neurons[j].InputY ←
        Network[PosCounter,PosCounter +1]

        PosCounter += 2

        CurrentGeneration[i].Neurons[j].Ouput ←
        OutputTable[Network[PosCounter]]

        PosCounter += 1

        CurrentGeneration[i].Neurons[j].TileType ← Network[PosCounter]
```

```
                PosCounter += 1

        NEXT

        CurrentGenerations[i].Fitness = Int(Network[PosCounter,PosCounter+4])

NEXT
```

PseudoCode of Different Functions

Reading data for a map

Creates the image of the level that represents what the AI can "see"

```
InternalDisplay ← Array[0,12][0,31]

VisualDisplay ← InternalDisplay

CurrentTile ← 1280

FOR i = 0,1 DO

        FOR j = 0,12 DO

                FOR k = (16*i),(16*(i+1)-1) DO

                        InternalDisplay[j][k] = ReadMemory(CurrentTile)

                        //Built in function that reads the memory location specified

                        CurrentTile += 1

                NEXT

        NEXT

NEXT

//Creating an offset for showing where Mario is

Offset ← 0 : INTEGER

FOR i = 0,12 DO

        FOR j = 0,15 DO

                Offset = CalculateOffset()
```

```
//A function that will read different memory address and perform Maths on them to find the position of Mario in relation to the edge of the map

                    VisualDisplay[i][j] -> InternalDisplay[i][Offset]

          NEXT

NEXT
```

Creating a new neuron

```
SUB New(InputX,InputY,Output,TileType)

          Self.InputX ← InputX

          Self.InputY ← InputY

          OutputTable ← ["A","B","Up","Down","Left","Right"]

          Self.Output ← OutputTable[Output]

          Self.TileType ← TileType

END SUB
```

Creating a new neural network

```
Neurons ← ARRAY : Neuron

Fitness ← 0

FOR i = 1,RandInt(1,3) DO

          Neurons[i] ← NEW
          Neuron(RandInt(1,16),RandInt(1,13),RandInt(1,6),RandInt(1,2))

NEXT
```

Playing the game

Has each network in a generation play the game until it either dies or gets stuck

```
FOR i =1,Length(CurrentGeneration) DO

          PlayingNetwork ← CurrentGeneration[i]

          LoadSaveState("State1")
```

```
Playing ← True

Inputs ← {}

NoInputCounter ← 0

JumpCounter ← 0

FramCounter ← 0

DO WHILE Playing

    PlayingNetwork.UpdateMap()

    Inputs ← PlayingNetwork.CheckForInputs()

    HasInputs ← False

    FOR j = 1,6 DO

        If Inputs[i] == True THEN

            HasInputs ←True

            NoInputCounter ← 0

        END IF

    NEXT

    IF NOT HasInputs THEN

        IF NoInputCounter >= 20 THEN

            Playing ← False

        ELSE

            NoInputCounter += 1

        END IF

    ELSE

        IF Inputs[“A”] == True THEN

            JumpCounter += 1

        ELSE

            JumpCounter ← 0
            END IF

            PlayingNetwork.RunIntputs(Inputs,JumpCounter)

        END IF

        FrameCounter += 1

        AdvanceFrame() //Built in functions that advances to the next frame

    LOOP

    PlayingNetwork.Fitness ← CalculateFitness(FrameCounter)

NEXT
```

Checking for Inputs

Checks to see if any inputs should be run

```
FUNCTION CheckForInputs()

    OutputTable = {“A” = False,”B” = False,”Up” = False,”Down” = False, ”Left” = False, ”Right” = False}

    FOR i = 1,Length(self.Neurons) DO

        IF VisualDisplay[Neurons[i].GetInputX][Neurons[i].GetInputY] == 0
        AND Neurons[i].GetTileType  == 2 THEN

            OutputTable[Neurons[i].GetOutput()] = True

        ELSEIF VisualDisplay[Neurons[i].GetInputX][Neurons[i].GetInputY] !=
        0 AND Neurons[i].GetTileType  == 1 THEN

        END IF

    NEXT

    RETURN OutputTable

END FUNCTION
```

Running inputs

Puts any inputs that have been detected into the controller

```
SUB RunInputs(Inputs, ByRef JumpCounter)

    IF JumpCounter .> 29 THEN

        Inputs{“A”} =← False
        JumpCounter ← 0
    END IF
    Joypad.Write(1,Inputs) //Built in function that writes the inputs in the table to the controller
END SUB
```

Calculate Fitness

Scores each network based on their performance

```
//Names used in reading memory in this function are placeholders for what the actual memory addresses used will be referring to
FUNCTION CalculateFitness(FrameCounter,Goal)
    Fitness ← 0
    IF Goal == 1 THEN
        Distance ← ReadMemory(CurrentScreen)*60 + ReadMemory(XPositionOnCurrentScreen)

    ELSEIF Goal == 2 THEN
        Distance ← ReadMemory(CurrentScreen)*60 + ReadMemory(XPositionOnCurrentScreen)
        Speed ← Distance*20 / FrameCounter
        Fitness += Speed

    ELSEIF Goal == 3 THEN
        Fitness += 10*ReadMemory(CoinCount)
    END IF

    RETURN Fitness

END FUNCTION
```

Sorting the neural networks by fitness values

A quick sort based off of the fitness values of each network in a generation

```
QuickSortByFitness( 1, Length(CurrentGeneration))

SUB QuickSortByFitness(LO, HI)

x ← LO

y ← HI

mid ← CurrentGeneration((x + y) / 2).getFitness()

DO WHILE x <= y

        DO WHILE CurrentGeneration(x).getFitness() > mid

                        x ← x + 1

        LOOP

        DO WHILE mid > CurrentGeneration(y).getFitness()

                y ← y - 1

         LOOP

         IF x <= y THEN

                TempFitness <- CurrentGeneration(x).getFitness()

                CurrentGeneration(x).setFitness(CurrentGeneration(y).getFitness())

                CurrentGeneration(y).setFitness(TempFitness)

                x ← x + 1

                y ← y - 1

        END IF
```

```
LOOP

IF LO < y THEN

        QuickSortByFitness(LO, y)

END IF

If HI > x Then

        QuickSortByFitness(x, HI)

END IF

END SUB
```

Evolving a Generation

Creates a new generation of networks by breeding two from the previous generation together

```
NumOfNetsInGen <- Length(CurrentGeneration)

NextGeneration <- ARRAY[NumOfNetsInGen]

Fitnesses <- ARRAY[1,Length(CurrentGeneration)]

FOR i = 1 to Length(CurrentGeneration)

        Fitnesses[i] = CurrentGeneration[i].GetFitness()

NEXT

QuickSortByFitness(Fitness, 1, Length(Fitnesses), CurrentGeneration)

FOR i = Length(CurrentGeneration)/2, Length(CurrentGeneration) DO

        CurrentGeneration[i] <- NULL

NEXT

NextGeneration[1] <- CurrentGeneration[1]

FOR i = 2, NumOfNetsInGen DO

        FirstPartner = CurrentGeneration[Random(1,Length(CurrentGeneration))]

        PartnerFound <- False
```

```
    SecondPartner : NeuralNetwork
    DO
        SecondPartner = CurrentGeneration[Random(1,Length(CurrentGeneration))]
        IF SecondPartner = FirstPartner THEN
            PartnerFound ←False
        ELSE IF CurrentGeneration(SecondPartner) – CurrentGeneration(FirstPartner) < 3 AND CurrentGeneration(SecondPartner) – CurrentGeneration(FirstPartner) > -3 THEN
            PartnerFound <- True
        ELSE
        PartnerFound <- False
    LOOP UNTIL PartnerFound
    NextGeneration[i] ← NEW NeuralNetwork
    FOR j = 1,Length(NextGeneration[i].Neurons DO
    NextGeneration[i].Neurons[j] ← NULL
    NEXT
    NextGeneration[i].Neurons ← FirstPartner.GetNeurons() + SecondPartner.GetNeurons())
    NextGeneration[i].Mutate()
NEXT
```

Mutating a network

Adds or removes a small amount of neurons in a network, creates the variation between generations.

```
SUB Mutate()
        FOR i = 1,RandInt(1,4)
                MutationType ← RandInt(1,4)
                IF MutationType = 4 THEN
                        Self.Neurons[RandInt[1,Length(Neurons)] ← NULL
                ELSE
                        Self.Neurons[Length(Neurons)+1] ← NEW
                        Neuron(RandInt(1,16),RandInt(1,13),RandInt(1,6),RandInt(1,2)
                        )
                END IF
        NEXT
END SUB
```

Random Function

Generates a random integer value between two bounds

```
Seed = (Seed * 722233  + 3459329) MOD (2^31 -1)
GeneratedValue = Seed MOD((UpperBound+1)-LowerBound)+LowerBound
RETURN Generated Value
```

Testing Strategy

Table of tests (includes unit tests and IO tests)

No.	Description	Data Type	Expected result
1	Data has been read from RAM. Test using an easy to understand variable e.g Coin count.		The data found in the specified memory location will be displayed in either the console or on screen
2	A normal level can be displayed on the UI	Typical	A small 32*13 grid with Grey squares for empty spaces in the level, white squares for spaces with a tile in it
3	The map is empty when no level is loaded	Erroneous	A small 32*13 grid is displayed but all squares are grey.

4	The map loads correctly for an unused level in the game's files	Extreme	32*13 grid is displayed and has the correct colours and therefore works for any level.
5	The map display moves as Mario moves left to right		Every 16 pixels or 1 tile Mario moves the map moves 1 grid space in the direction he moves.
6	Creation of a neuron in a network. Test by running the network creation subroutine and then output the neurons' properties.		The Neuron should have an Xcoordinate from 1-32, a Y coordinate from 1-13, a valid output and a valid tile type for detection.
7	Networks can be stored in a single string.		Network is created and is saved into a text file as a single string
8	Networks can be decoded from the string they're stored as.		A text file that contains a network is opened and a new network is created that is the same as the previous one that was saved.
9	Sort algorithm that sorts networks based on a single value in the network works		An unsorted list of networks is inputted and a sorted list is outputted.
10	A generation can have the bottom 50% removed		The list of networks is outputted being half the length it was originally.
11	Top scoring network is saved for the next generation		Before the evolving algorithm is run the next generation should contain 1 network that is identical to the top scoring one from the previous generation

12	Breeding networks works based on the two parent networks		After one iteration off breeding the new generation contains a network with neurons chosen from the two parent networks (up to 3 more neurons maybe added due to mutation)
13	A save state can be loaded	Typical	A new save state can be created and loaded
14	Check for neuron inputs that are active		Any inputs from neurons that are active will be output to the console.
15	Automatic controller inputs will work		Mario will move will the controller inputs that have been specified
16	Next network is loaded when Mario dies		When Mario dies the current network is closed and saved and the save state is loaded with a new network active.
17	Next network is loaded when there are no inputs for 60 frames/1 second		After 60 frames of doing nothing the current network is closed and saved and the save state is loaded with a new network active.
18	Fitness is calculated after a network's game ends (Based on Distance)		When a game ends a fitness value is outputted to the console.
19	Fitness is calculated after a network's game ends (Based on Speed)		When a game ends a fitness value is outputted to the console.

20	Fitness is calculated after a network's game ends (Based on Coins collected)		When a game ends a fitness value is outputted to the console.
21	The cancel button on any input sends the user back to the Creating/Loading menu (As all inputs come from that menu)		The user is sent back to the Creating/Loading menu when pushing cancel when asked for any input (New Pop. Name, Pop. Size, Loading name, Generation to load)
22	Loading a population from a file	Typical	The population saved in the file with the name selected is loaded
23	Trying to load a population from a file that doesn't exist	Erroneous	If the file does not exist then the user is given an error message telling them it doesn't exist and
24	Leaving the field blank	Erroneous	The user is asked to input a name again
25	Loading a file that only has 1 generation (Generation 0)	Extreme	The population saved in the file with the name selected is loaded.
26	Generation Number Input exists for a population	Typical	The population is loaded correctly
27	Generation does not exist for the population inputted	Erroneous	The user given an error message telling them that the population doesn't exist or the generation doesn't exist for that population
28	The Generation input is left	Erroneous	The user is asked to input a the number for the generation they would like to load again

	blank or is not a number		
29	The Generation input is 0	Extreme	The population is loaded correctly
30	A new population will be created with the name the user inputs into the input box	Typical	The population is created with the selected name
31	The new population box is left blank	Erroneous	The user is asked to input the name again.
32	The name inputted is the same as an already existing population	Extreme	A new population is created overwriting the old one with the same name (The user is warned about this from the input box)
33	The Size of a new population is inputted	Typical	A new population with the specified size is created.
34	A size of 2 or less is entered, the input is not a string or the field is left blank	Erroneous	The user is asked to input the size again
35	A size of 3 is entered	Extreme	A new population with a size of 3 is created
36	The top fitness of a generation is saved to a population’s “stats” file		There is a file saved with the population’s files that contains 1 fitness value per line and is as long as there are generations of that population
37	The random function generates a new random seed each time it		Each time a random number is generated the new seed is printed and none of them should be identical

	generates a random number.		
38	Each of the 3 levels can be loaded		When a level is chosen that level is then loaded for the population
39	Random Function generates random numbers between an upper and lower bound		Random numbers are generated that are between the upper and lower bound and can be both

Testing

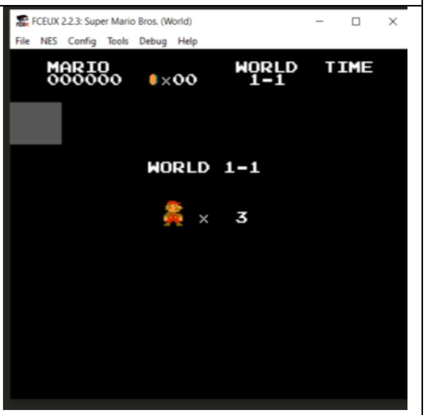
Testing Playlist

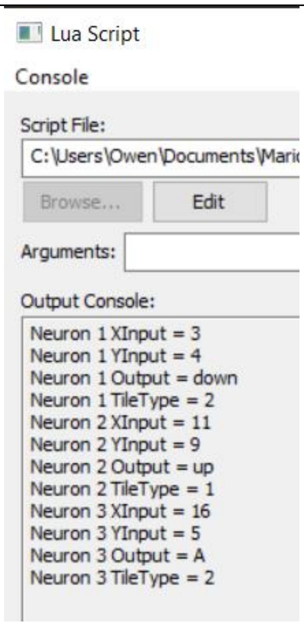
Many of the tests required video evidence and these have been compiled into a YouTube playlist for easy viewing.

https://www.youtube.com/playlist?list=PLLZ_69SDcy1JXYc2mnDZzuaB0wv2PEt-E

Testing Table

No.	Description	Data Type	Expected result	Pass/Fail	Evidence
1	Data has been read from RAM. Test using an easy to understand variable e.g Coin count.		The data found in the specified memory location will be displayed in either the console or on screen	Pass	Video "Test #1" from testing playlist

2	A normal level can be displayed on the UI	Typical	A small 32*13 grid with Grey squares for empty spaces in the level, white squares for spaces with a tile in it	Pass	Video "Test #2" from testing playlist
3	The map is empty when no level is loaded. Test by using loading screen as no level is loaded on during loading screens.	Erroneous	A small 32*13 grid is displayed but all squares are grey.	Pass	 FCEUX 2.2.3: Super Mario Bros. (World) File NES Config Tools Debug Help MARIO 000000 ×00 WORLD 1-1 TIME WORLD 1-1 × 3
4	The map loads correctly for an unused level in the game's files	Extreme	32*13 grid is displayed and has the correct colours and therefore works for any level.	Pass	Video "Test #4" from testing playlist
5	The map display moves as Mario moves left to right		Every 16 pixels or 1 tile Mario moves the map moves 1 grid space in the direction he moves.	Pass	Video "Test #2" from testing playlist

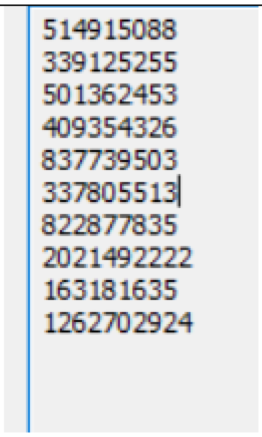
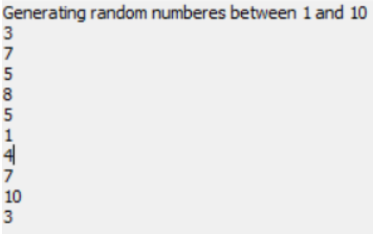
6	Creation of a neuron in a network. Test by running the network creation subroutine and then output the neurons' properties.		The Neuron should have an Xcoordinate from 1-32, a Y coordinate from 1-13, a valid output and a valid tile type for detection.		 Script File: C:\Users\Owen\Documents\Mari Arguments: Output Console: Neuron 1 XInput = 3 Neuron 1 YInput = 4 Neuron 1 Output = down Neuron 1 TileType = 2 Neuron 2 XInput = 11 Neuron 2 YInput = 9 Neuron 2 Output = up Neuron 2 TileType = 1 Neuron 3 XInput = 16 Neuron 3 YInput = 5 Neuron 3 Output = A Neuron 3 TileType = 2
7	Networks can be stored in a single string.		Network is created and is saved into a text file as a single string	Pass	Video "Tests #7 and #8" from testing playlist
8	Networks can be decoded from the string they're stored as.		A text file that contains a network is opened and a new network is created that is the same as the previous one that was saved.	Pass	Video "Tests #7 and #8" from testing playlist
9	Sort algorithm that sorts networks based on a single value in the network works		An unsorted list of networks is inputted and a sorted list is outputted.	Pass	Video "Test #9" from testing playlist
10	A generation can have the bottom 50% removed		The list of networks is outputted being	Pass	Video "Test #10" from testing playlist

			half the length it was originally.		
11	Top scoring network is saved for the next generation		Before the evolving algorithm is run the next generation should contain 1 network that is identical to the top scoring one from the previous generation	Pass	Video “Test #11” from testing playlist
12	Breeding networks works based on the two parent networks		After one iteration of breeding the new generation contains a network with neurons chosen from the two parent networks (up to 3 more neurons maybe added due to mutation)	Pass	Video “Test #12” from testing playlist
13	A save state can be loaded		A Save state will load	Pass	Video “Test #13” from testing playlist
14	Check for neuron inputs that are active		Any inputs from neurons that are active will be output to the console.	Pass	Video “Test #14” from testing playlist
15	Automatic controller inputs will work		Mario will move will the controller inputs	Pass	Video “Test #15” from testing playlist

			that have been specified		
16	Next network is loaded when Mario dies		When Mario dies the current network is closed and saved and the save state is loaded with a new network active.	Pass	Video “Tests #16 and #17” from testing playlist
17	Next network is loaded when there are no inputs for 60 frames/1 second		After 60 frames of doing nothing the current network is closed and saved and the save state is loaded with a new network active.	Pass	Video “Tests #16 and #17” from testing playlist
18	Fitness is calculated after a network’s game ends (Based on Distance)		When a game ends a fitness value is outputted to the console.	Pass	Video “Test #18” in the testing playlist
19	Fitness is calculated after a network’s game ends (Based on Speed)		When a game ends a fitness value is outputted to the console.	Pass	Video “Test #19” in the testing playlist
20	Fitness is calculated after a network’s		When a game ends a fitness value is	Pass	Video “Test #20” in the testing playlist

	game ends (Based on Coins collected)		outputted to the console.		
21	The cancel button on any input sends the user back to the Creating/Loading menu (As all inputs come from that menu)		The user is sent back to the Creating/Loading menu when pushing cancel when asked for any input (New Pop. Name, Pop. Size, Loading name, Generation to load)	Pass	Video “Test #21” in the testing playlist
22	Loading a population from a file	Typical	The population saved in the file with the name selected is loaded	Pass	Video “Tests #25 and #29” in the testing playlist
23	Trying to load a population from a file that doesn’t exist	Erroneous	If the file does not exist then the user is given an error message telling them it doesn’t exist and	Pass	Video “Tests #23 and #24” in the testing playlist
24	Leaving the field blank	Erroneous	The user is asked to input a name again	Pass	Video “Tests #23 and #24” in the testing playlist
25	Loading a file that only has 1 generation (Generation 0)	Extreme	The population saved in the file with the name selected is loaded.	Pass	Video “Tests #25 and #29” in the testing playlist
26	Generation Number Input	Typical	The population is loaded correctly	Pass	Video “Tests #22 and #26” in the testing playlist

	exists for a population				
27	Generation does not exist for the population inputted	Erroneous	The user given an error message telling them that the population doesn't exist or the generation doesn't exist for that population	Pass	Video "Test #27" in the testing playlist
28	The Generation input is left blank or is not a number	Erroneous	The user is asked to input a the number for the generation they would like to load again	Pass	Video "Test #28" in the testing playlist
29	The Generation input is 0	Extreme	The population is loaded correctly	Pass	Video "Tests #25 and #29" in the testing playlist
30	A new population will be created with the name the user inputs into the input box	Typical	The population is created with the selected name	Pass	Video "Tests #30 and #33" in the testing playlist
31	The new population box is left blank	Erroneous	The user is asked to input the name again.	Pass	Video "Test #31" in the testing playlist
32	The name inputted is only 1 character long	Extreme	A new population is created with a 1 character name	Pass	Video "Test #32" in the testing playlist
33	The Size of a new population is inputted	Typical	A new population with the specified size is created.	Pass	Video "Tests #30 and #33" in the testing playlist

34	A size of 2 or less is entered, the input is not a string or the field is left blank	Erroneous	The user is asked to input the size again	Pass	Video "Test #34" in the testing playlist
35	A size of 3 is entered	Extreme	A new population with a size of 3 is created	Pass	Video "Test #35" in the testing playlist
36	The top fitness of a generation is saved to a population's "stats" file		There is a file saved with the population's files that contains 1 fitness value per line and is as long as there are generations of that population	Pass	Video "Test #36" in the testing playlist
37	The random function generates a new random seed each time it generates a random number.		Each time a random number is generated the new seed is printed and none of them should be identical	Pass	 514915088 339125255 501362453 409354326 837739503 337805513 822877835 2021492222 163181635 1262702924
38	Each of the 3 levels can be loaded		When a level is chosen that level is then loaded for the population	Pass	Video "Test #38" in the testing playlist
39	Random Function generates random numbers		Random numbers are generated that are between the upper and lower	Pass	 Generating random numberes between 1 and 10 3 7 5 8 5 1 4 7 10 3

	between an upper and lower bound		bound and can be both		

Beta Testing

I asked a third party to use the program for me and then interviewed them about their experience.

Did you find the program easy to use?

"The program was clear and easy to understand but it could have used some more explanation as to what some of the more technical terms mean, for instance, What does it mean by the number of members in a population?"

Was the UI easy to understand?

"The UI was very simple and easy to understand but a bit bland and boring to look at."

Did you have any problems with the program when you tried to use it?

"The program worked perfectly as intended however it was very slow to train and I had to leave it running for a long time to get any results. I did try to break the program by putting garbage data into any inputs but it didn't break at all when I tried to break it."

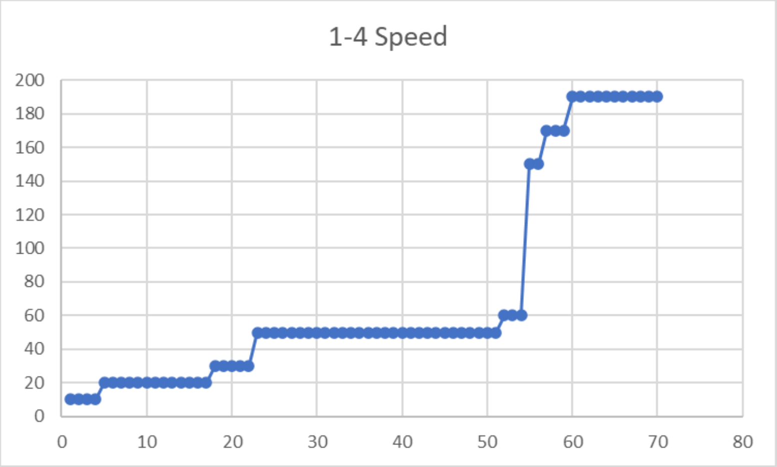
How well did the AI work?

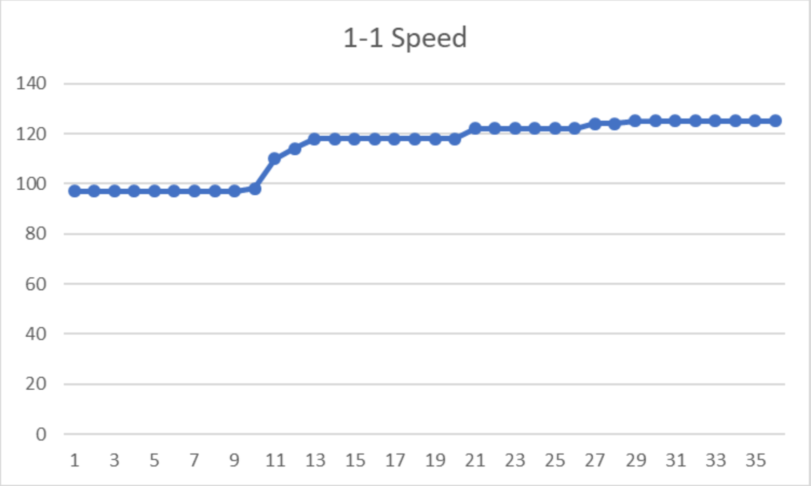
To test whether or not an AI would be able to learn to play Super Mario Bros. I trained 9 different AIs, 1 of each reward type on 3 vastly different levels, and created a graph for each detailing their performance over time. (Note the Y axis is an arbitrary fitness value and the actual numbers don't reflect how the AIs compared to each other.)

The 3 levels were: 1-1 – a standard overworld level with nothing special about it, 1-4 – A castle level with lots of traps and a boss fight at the end, and 7-2 – an underwater level filled with enemies and where the game controls very differently.

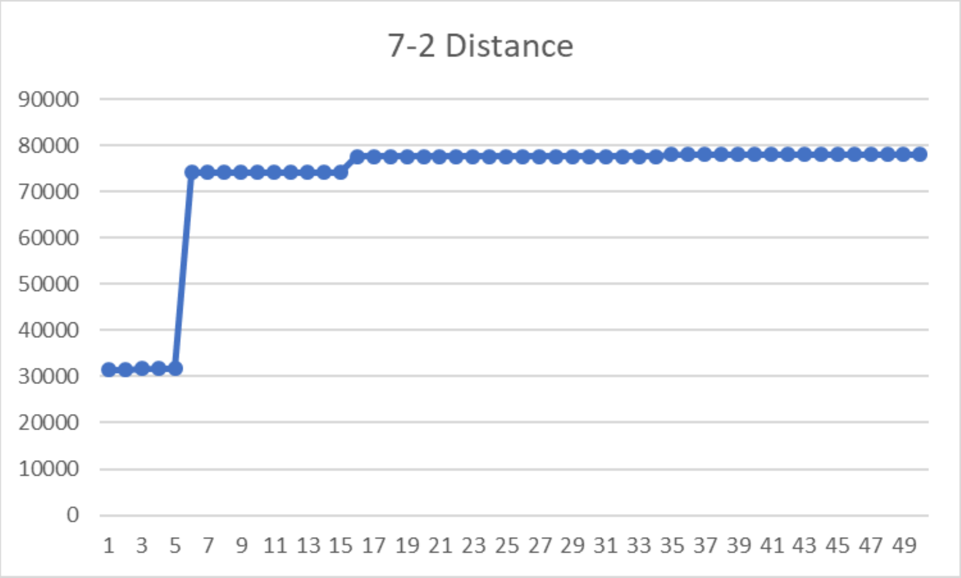
The 3 reward types were: Distance – To test if the AI would be able to learn to reach the goal (the standard objective), Speed – To test if the AI could learn to do it fast, and Coins Collected – To see if the AI could learn the layout of a level and find the most amount of coins.

In general, each AI was able to learn to perform its task to a high standard given a enough time to train. The main exception to this being the AIs trained on speed. All of the speed AIs never learned to finish the level they were playing, usually making it about three quarters of the way through the level before deciding that it took too long to do the last quarter and giving up. Speed however was shown to have one of the more gradual increases in performance over time.

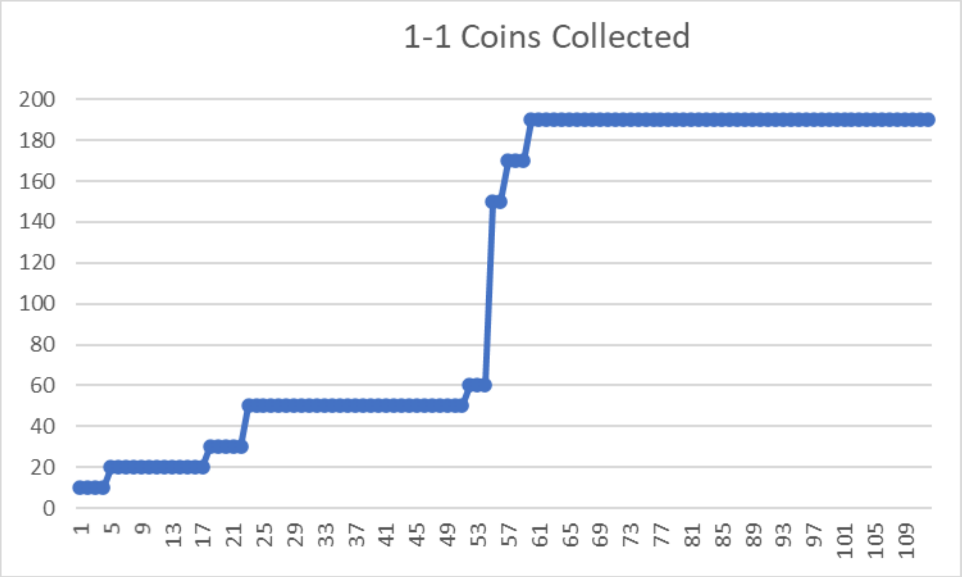


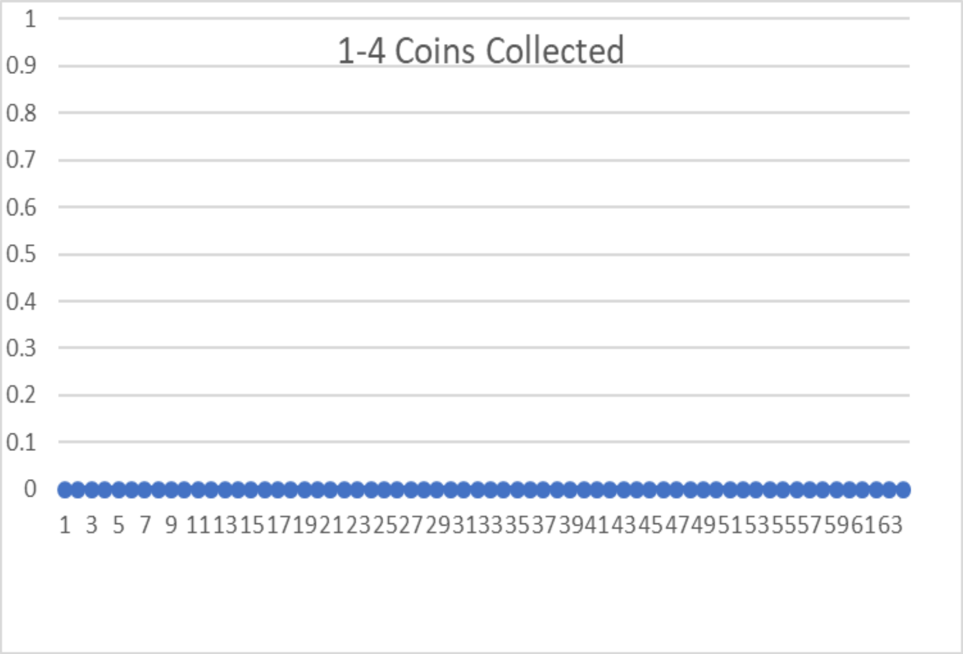


The level the AIs had the hardest time with was the underwater level. In this level Mario controls differently when the A button is pushed instead of jumping Mario will swim upwards a small amount and can do this multiple times without touching the ground. This led the AI to swimming up to the top of the level and just staying up there most of the time as it couldn't quite get the hang of swimming properly, this meant the AI had a hard time beating the level as it got stuck one walls that blocked the top of the level. Despite this the AI was still able to improve greatly on this stage, when being rewarded for distance or coins collected, and made a lot of progress being able to avoid the enemies and water currents that make the level difficult.



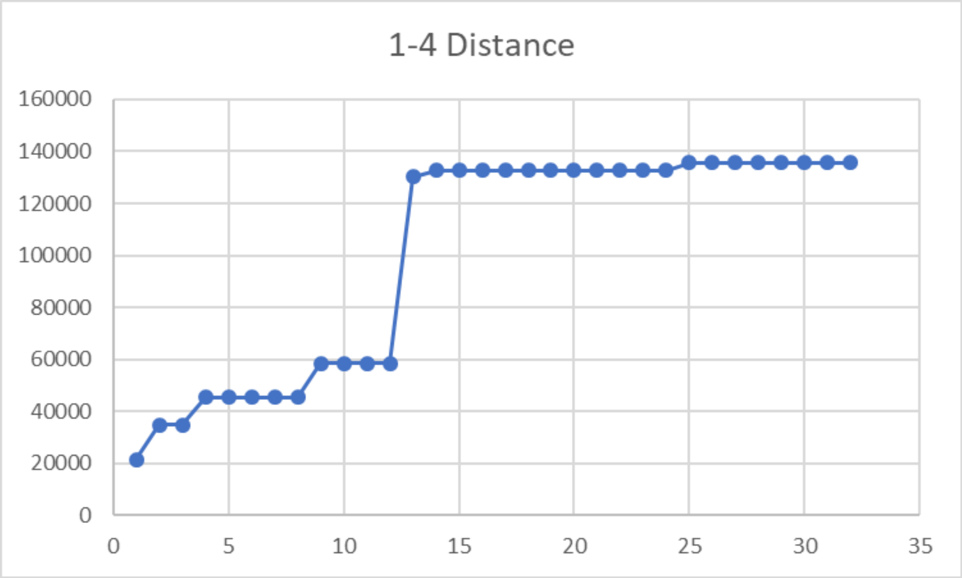
The most interesting reward type was the coins collected. As coins are stored in the game's memory as just another object the AI was not able to tell the coins apart from anything else. This meant that instead of reacting to coins and collecting them the AI would instead learn the layout of a level and find the best route to collect the most amount of coins in a level. For example, in 1-1 the AI would hit all of the blocks early on in the level, collecting the coins within them, and would then move onto a hidden room that contains a large amount of coins. This reward system worked best in levels that had more coins.

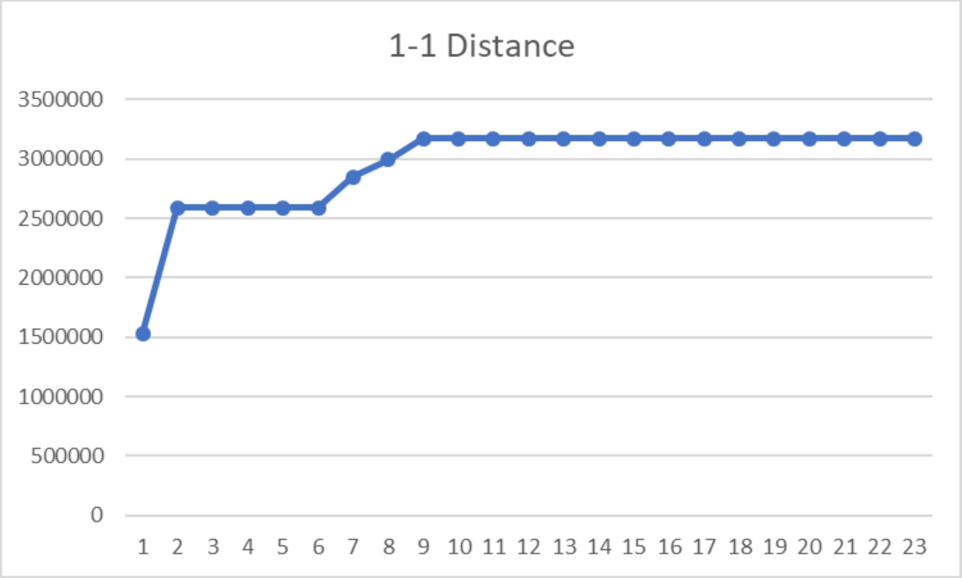


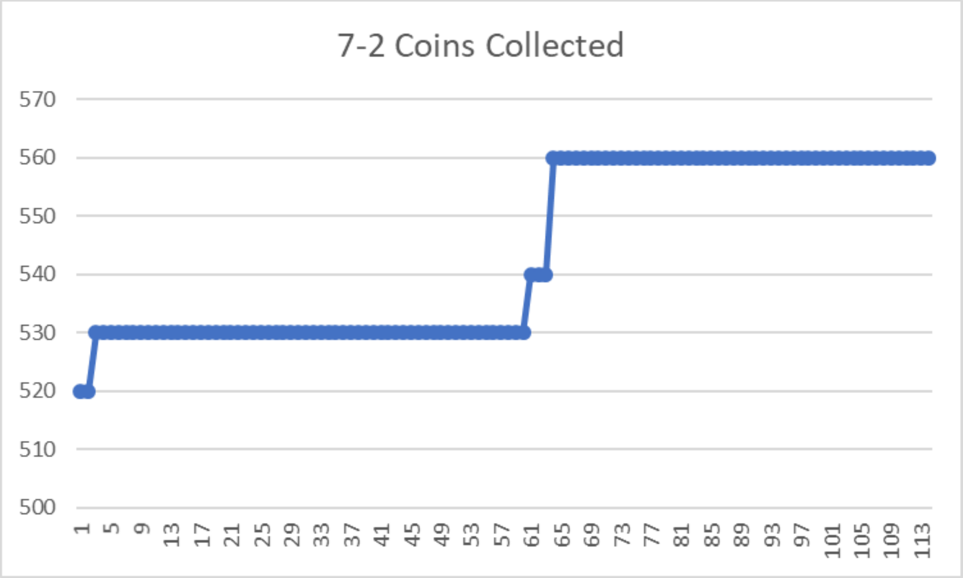


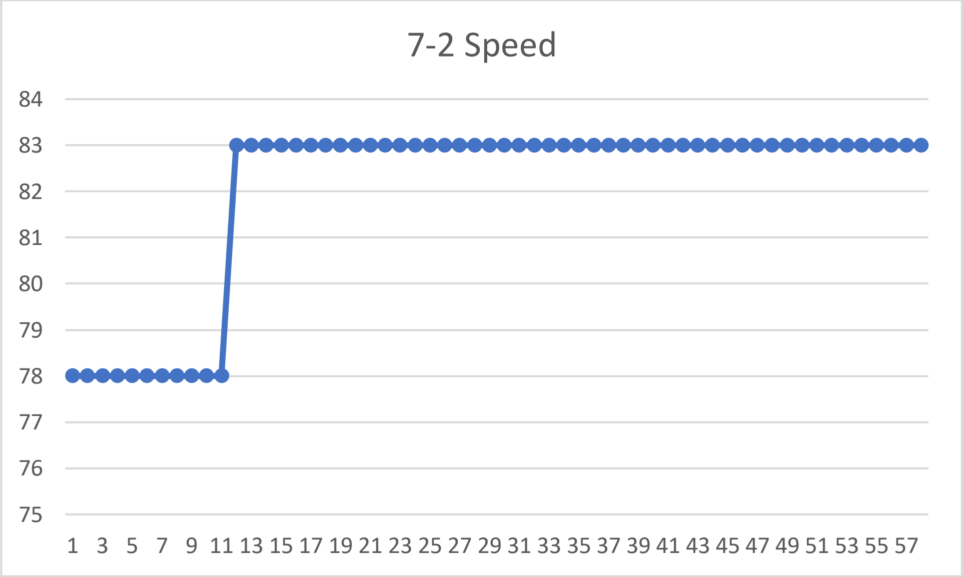
However, the AI broke when trying to learn to collect coins in 1-4 as there are only 4 coins in the whole level and they are all at the very end. This mean that the AI never learnt anything instead opting to create the most minimal network possible that didn’t do anything. This demonstrates that the AI does need to be rewarded in any capacity to be able to learn.

Below are the graphs for the rest of the AIs not shown above.






7-2 Coins Collected
570
560
550
540
530
520
510
500
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97
101
105
109
113


7-2 Speed
84
83
82
81
80
79
78
77
76
75
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57

Evaluation

Have the requirements been met?

Req. #	Requirement	Is the requirement met?	Testing Ref.
0.1	Read the RAM for tile data	Yes	Test #1
0.2	Form an ordered array from the tile data	Yes	Test #2
0.3	Make the map move as Mario moves	Yes	Test #5
1.1	Create Neurons with an input, made of coordinates and a tile type, and an output.	Yes	Test #6
1.2	Store data for a neural network as a single string in a uniform fashion to make it easier to read when accessing previous networks	Yes	Test #7
1.3	Interpret the data string for a neural network	Yes	Test #8
2.1	Create an ordered list of all neural networks for the current generation based on fitness values	Yes	Test #9
2.2	Remove the bottom 50% scoring neural networks	Yes	Test #10
2.3	Keep the top scoring neural network for the next generation	Yes	Test #11
2.4	Breed remaining neural networks until the desired number of networks for the next generation exist	Yes	Test #12

3.1	Load a pre-existing save state at the start of the selected level from a list	Yes	Test #13
3.2	Check for any inputs from neurons	Yes	Test #14
3.3	Perform inputs into the game	Yes	Test #15
3.4	Check if the game is ready to end via either no inputs or Mario dying	Yes	Tests #16 and #17
3.5	Calculate Fitness value for a neural network based on the different scoring methods: Distance, Speed and Coins collected	Yes	Tests #18-#20
4.1	Random function generates a new seed every time it is used	Yes	Test #37
4.2	Random function generates integers between an upper and lower bound	Yes	Test #39

How could the outcome be improved if the problem was revisited?

The program and algorithm are able to create an AI that can learn to play Super Mario Bros. and can easily adapt to any scenario that is thrown at it. However there is one main constraint with the program and that is the amount of time it takes to train the AI, so if the problem were to be revisited would focus on finding solution to make the training shorter. This could be done by making it so multiple Networks in a generation are able to play at the same time and are all able to feed back to the program at the same time so that a whole generation could be finished in the time it would normally take a single network to play. Another method of approaching this problem would be to add more complexity to the networks. This would mean that they would be able to handle more complex problems with less training therefore reducing the time taken to train.

Feedback from the end user

I gave Alex a copy of the program and also showed him some of the AIs I had created or testing and then asked him what he thought of the program and whether it met his requirements.

"The program was clear and easy to understand with a simple and easy to use UI. Whilst the AIs take a very long time to train it was easy to see clear improvements happening slowly over time and from the examples you showed me I could see the AI would develop. There were some technical terms that were hard to understand and could have used some explaining like explaining what a member in a population is. The program also never broke when I used it. Overall this program has helped a lot with my investigation to see if an AI can learn to play Super Mario Bros."

Code Dump

```
require("iuplua") --Library that controls the GUI



local RandomGen = {} --Start of class definition of RandomGen class - RandomGen is a class that is used for creating random integers

RandomGen.__index = RandomGen


function RandomGen.New() --Constructor for RandomGen class

  local self = setmetatable({},RandomGen)

  self.seed = 0

  return self

end

function RandomGen:setSeed(NewSeed)

  self.seed = NewSeed

end

function RandomGen:generateRandom(LowerBound,UpperBound) --Method for RandomGen objects that generates a random integer between the two bounds (inclusive)
```

```
  local NextSeed = 0


  if LowerBound == nil then -- Allows for not having a lower bound
    LowerBound = 1
  end
  if UpperBound == nil then -- Allows for not having an upper bound
    UpperBound = 2
  end


  NextSeed = (self.seed * 722233 + 3459329)  % (2^31-1) --Generates a new seed based on the previous seed
  self:setSeed(NextSeed)
  NextSeed = NextSeed%((UpperBound+1) - LowerBound) + LowerBound --Calculates the integer that is to be generated between the two bounds
  return NextSeed
end
--End of the class definition for RandomGen class
RandGen = RandomGen.New() --Creates a Random numbe generating function
RandGen:setSeed(os.time()) --Sets the starting seed based off of the system clock
```

```
local Neuron = {} --Start of the class definition for the Neuron class
Neuron.__index = Neuron
function Neuron.New(X,Y,Output,Type) --Constructor for the Neuron class
  local self = setmetatable({}, Neuron)


  local PossibleOutputs = {"A","B","up","down","left","right"}
  self.InputX = X --Sets up the Object's properties based off of the values given at object creation
  self.InputY = Y
  self.Output = PossibleOutputs[Output]
  self.TileType = Type
  return self
end
--Accessor methods for the Neuron class
function Neuron:getInputX()
  return self.InputX
end
```

```
function Neuron:getInputY()
  return self.InputY
end
function Neuron:getOutput()
  return self.Output
end
function Neuron:getTileType()
  return self.TileType
end
--End of the class definition for Neuron class


local Network = {} -- Start of the class definition for the Network class
Network.__index = Network
function Network.New() -- Constructor for the Network class
  local self = setmetatable({},Network)
  self.Neurons = {}
  self.Map = {}
  self.Fitness = 0


  self:setUpMap()

  return self

end

--Accessor method for Network class

function Network:getFitness()

  return self.Fitness

end


function Network:genZeroSetUp() --Used when creating a new population and there are no parent networks that have been bred

  local X

  local Y

  local Out

  local Type


  --Generates random values for each of the properties of the 3 neurons that a network will start with
  for i = 1,3 do
    X = RandGen:generateRandom(1,16)


    Y = RandGen:generateRandom(1,13)


    Out = RandGen:generateRandom(1,6)


    Type = RandGen:generateRandom(1,2)


    self.Neurons[i] = Neuron.New(X,Y,Out,Type)
    --print("Neuron ".. i .. " created" .. X .. Y .. Out .. Type)
  end
end


function Network:setUpMap() --Lua doesn't allow tables to have a pre-defined size so this function does that for the map table
  TempMap = {}
  for i = 1,13 do
```

```
    TempMap[i] = {}

    self.Map[i] = {}

    for j = 1,32 do

      TempMap[i][j] = 0

    end

    for j = 1,16 do

      self.Map[i][j] = 0

    end

  end

end


function Network:UpdateMap() --Updates what Mario can see each frame based on his posistion and then displays it on screen

  local NextTileAdress = 1280

  local PosOffset = 0

  local Colour = ""


  for i = 0,1 do
```

```
    for j = 1,13 do
      for k = ((16*(i))),(16*(i+1)-1) do
        TempMap[j][k] = memory.readbyte(NextTileAdress) --Reads the memory addresses for the tiles
        NextTileAdress = NextTileAdress + 1
      end
    end
  end
  for i = 1,13 do
    for j = 0,15 do
      PosOffset = (((j + math.floor((memory.readbyte(0x0086)-40)/16))%32)+(16*(memory.readbyte(0x006D)%2)))%32 --Does the maths to calculate offset based on Mario's posistion
      self.Map[i][j+1] = TempMap[i][PosOffset]
      Colour = self:squareColour(i,PosOffset)
      gui.box(j*2,i*2+40,(j+1)*2,(i+1)*2+40,Colour)

    end
  end
end
```

```
function Network:squareColour(Row,Column) --Determines the colour of the next tile to be displayed
  if (TempMap[Row][Column] == nil) then --If there is an error loading data for the map the tiles that could not be loaded will be displayed red
    return "#FF0000"
  elseif (TempMap[Row][Column] ~= 0) then --Wherever there is not an empty tile it is displayed as being white on the map
    return "#FFFFFF"
  else --If a tile is empty it is displayed as being grey on the map
    return "#505050"
  end
end


function Network:Mutate() --The function that controls the mutation of a network where a random amount of neurons are either added or removed from a network
  local NeuronAmount = #self.Neurons --Stores the length of the Neurons table to be able to keep track of the changing amount of neurons when mutating
  local MutationAmount = RandGen:generateRandom(0,3)
  local MutationType
  while MutationAmount > 0 do

    MutationType = RandGen:generateRandom(1,2) --1 = Add Neuron, 2 = Remove Neuron

    if MutationType == 1 then

      self.Neurons[NeuronAmount] = Neuron.New(RandGen:generateRandom(1,16),RandGen:generateRandom(1,13),RandGen:generateRandom(1,6),RandGen:generateRandom(1,2))

      NeuronAmount = NeuronAmount + 1

    else

      if #self.Neurons > 1 then --Prevents mutation making a network useless by removing all neurons

        self.Neurons[RandGen:generateRandom(1,NeuronAmount)] = nil

        for i = 1,NeuronAmount do --Removes nil values from the table as Lua assumes a table terminates where the first nil value is even if there are more values after it

          if self.Neurons[i] == nil then

            self.Neurons[i] = self.Neurons[i+1]

            self.Neurons[i+1] = nil


          end

        end
      end

      NeuronAmount = NeuronAmount - 1

    end

    MutationAmount = MutationAmount - 1

  end

end


function Network:checkForInputs() --The function that checks if any of the neurons in a network have activated on a frame

  --The table of inputs that is possible for the controller. Note: some of the inputs are not possible to be used be the AI as they do not affect gameplay

  local OutputTable = {up = false, down = false, right = false, left = false, start = false, select = false, start = false, A = false, B = false}

  for i = 1,#self.Neurons do


    if self.Map[self.Neurons[i]:getInputY()][self.Neurons[i]:getInputX()] ~= 0 and self.Neurons[i].TileType == 1 then --If "TileType" is 1 then it checks if there is anything at the location the neuron points to

      OutputTable[self.Neurons[i]:getOutput()] = true

    elseif self.Map[self.Neurons[i]:getInputY()][self.Neurons[i]:getInputX()] == 0 and self.Neurons[i].TileType == 2 then --If "TileType" is 2 then it checks if the location the neuron points to is empty
```

```
      OutputTable[self.Neurons[i]:getOutput()] = true
    end
  end
  return OutputTable
end


function Network:runInputs(Inputs,JumpCounter) --Runs the inputs for a particular frame
  --Mario's maximum jump height takes 29 frames to reach and if the A button is held down he cannot jump again
  --so every 30 frames that the A button is held down for there is one frame where it is not held down to allow Mario to jump multiple times
  if JumpCounter > 29 then
    Inputs["A"] = false
    JumpCounter = 0
  end

  joypad.write(1,Inputs)

  return JumpCounter
```

```
end


function Network:calculateFitness(FrameCounter,Distance,Goal) --Calculates the fitness value of a network once it has finished playing
  local Fitness = 0
  local Speed = 0


  if Goal == 1 then --Trained for distance
    Fitness = math.floor((Distance*60))
  elseif Goal == 2 then --Trained for speed
    Fitness = math.floor((Distance*60/FrameCounter))
  elseif Goal == 3 then --Trained for coins collected
    Fitness = 10*memory.readbyte(0x075E)
  else
    print("Error calculating fitness: Unknown Goal")
  end


  self.Fitness = Fitness
```

```
end

--End of the class definition of the Network class


Population = {} --Start of the class definition for the Population class

Population.__index = Population

function Population.New(GenNo,PopulationSize,Name,Goal) --Constructor for the Population Class


  local self = setmetatable({},Population)

  self.CurrentGeneration = {}

  self.PreviousGeneration = {}

  self.GenerationNo = 0

  self.PopulationName = ""

  self.GenerationNo = GenNo

  self.PopulationName = Name

  self.Goal = Goal

  if GenNo == 0 then --If the generation of a population is zero then there will be no networks to load so new ones have to be created

    self:createGenZero(PopulationSize)
  end


  return self

end


function Population:createGenZero(PopSize)--Creates new networks for a new population


  for i = 1,PopSize do

    self.CurrentGeneration[i] = Network.New()

    self.CurrentGeneration[i]:genZeroSetUp()


  end


  StatsFile = io.open(self.PopulationName .. "Stats.txt","w") --Creates the stats file where fitness values are saved so they can be evaluated later

  io.output(StatsFile)

  io.close(StatsFile)

end
```

```
function Population:playGame() --The function that controls all the actions that occur when the AI is playing
  for i = 1,#self.CurrentGeneration do --Repeats for all the networks in a population
    local PlayingNetwork = self.CurrentGeneration[i]
    local Playing = true
    local Inputs = {}
    local NoInputCounter = 0
    local JumpCounter = 0
    local FrameCounter = 0
    local PossibleInputs = {"A","B","up","down","left","right"}
    local LastXPos = 0
    local Rightmost = 1
    memory.writebyte(0x075E,0) --Sets the coin count in memory to 0 as in some of the savestates used the coin count does not start at 0
    savestate.load(LevelStart) --Loads the savestate

    while (Playing) do --Each iteration of this loop is 1 frame of gameplay
      PlayingNetwork:UpdateMap()
```

```
Inputs = PlayingNetwork:checkForInputs()


HasInputs = false
for j = 1,8 do
  if Inputs[PossibleInputs[j]] == true then

    HasInputs = true



  end
end

---Failure conditions, if there are no inputs or Mario is dead or Mario has not moved horizontally then the counter increments,
--if the counter reaches 60 (Equivalent to 1 second as the game runs at 60 FPS) then the network stops playing
if not HasInputs or memory.readbyte(0x000E) == 11 or LastXPos == memory.readbyte(0x0086) then
  if NoInputCounter >= 60 then
```

```
      Playing = false
      print("Stop playing")
    else
      NoInputCounter = NoInputCounter + 1
    end
  else
    NoInputCounter = 0
  end
  --Checks how many frames Mario has been jumping for
  if Inputs["A"] == true then
    JumpCounter = JumpCounter + 1
  else
    JumpCounter = 0
  end

  JumpCounter = PlayingNetwork:runInputs(Inputs,JumpCounter)
  FrameCounter = FrameCounter + 1
```

```
      LastXPos = memory.readbyte(0x0086)

      emu.frameadvance()


      if (memory.readbyte(0x006D)*256 + memory.readbyte(0x0086)) > Rightmost  then --Calculates the furthest to the right Mario has gotten, used for Distance and Speed based fitness

        Rightmost = memory.readbyte(0x006D)*256 + memory.readbyte(0x0086)

      end

    end

    print("Network ".. i .. " done")


    PlayingNetwork:calculateFitness(FrameCounter,Rightmost,self.Goal)

  end

end


function Population:sortByFitness(LO,HI) --Does a quick sort to sort the list of networks in a population by their fitness values

  local LowPointer = LO

  local HighPointer = HI

  local Mid = self.CurrentGeneration[math.ceil((LowPointer +HighPointer)/2)]:getFitness()
  local TempNetwork

  while LowPointer <= HighPointer do

    while self.CurrentGeneration[LowPointer]:getFitness() > Mid do

      LowPointer = LowPointer + 1

    end

    while self.CurrentGeneration[HighPointer]:getFitness() < Mid do

      HighPointer = HighPointer - 1

    end

    if LowPointer <= HighPointer then

      TempNetwork = self.CurrentGeneration[LowPointer]

      self.CurrentGeneration[LowPointer] = self.CurrentGeneration[HighPointer]

      self.CurrentGeneration[HighPointer] = TempNetwork

      LowPointer = LowPointer + 1

      HighPointer = HighPointer - 1

    end

  end

  if LO < HighPointer then
```

```
    self:sortByFitness(LO,HighPointer)

  end

  if HI > LowPointer then

    self:sortByFitness(LowPointer,HI)

  end

end


function Population:evolvePopulation() --Evolves the population based on fitness scores

  local NextGeneration = {}

  local FirstPartner = Network.New()

  local SecondPartner = Network.New()

  local FirstPartnerNum

  local SecondPartnerNum

  local PartnerNotFound = true

  local NeuronCounter = 1

  local BadNet1 --Two lower scoring networks are kept for breeding to help reduce the odds of achieving a local maximum

  local BadNet2
  local BadNet1Num = RandGen:generateRandom(math.ceil((#self.CurrentGeneration)/2),#self.CurrentGeneration) --Gets a random integer from that is greater than half the size of the population
  local BadNet2Num = RandGen:generateRandom(math.ceil((#self.CurrentGeneration)/2),#self.CurrentGeneration)
  local HasNoNeurons = true
  local OutputToNum = {A = 1, B = 2, up = 3, down = 4, left = 5, right = 6}
  while BadNet1Num == BadNet2Num do
    BadNet2Num = RandGen:generateRandom(math.ceil((#self.CurrentGeneration)/2),#self.CurrentGeneration)
  end
  BadNet1 = self.CurrentGeneration[BadNet1Num]
  BadNet2 = self.CurrentGeneration[BadNet2Num]


  self:sortByFitness(1,#self.CurrentGeneration) --Sorts the population


  for i = math.ceil((#self.CurrentGeneration)/2),#self.CurrentGeneration do --Removes the bottom 50% of networks
    self.CurrentGeneration[i] = nil
  end
  self.CurrentGeneration[#self.CurrentGeneration + 1] = BadNet1
  self.CurrentGeneration[#self.CurrentGeneration + 1] = BadNet2
  NextGeneration[1] = self.CurrentGeneration[1] --Keeps the top scoring network from the previous generation so that the
population doesn't evolve backwards


  for i =        2,(#(self.CurrentGeneration)-1)*2 do


    FirstPartnerNum = RandGen:generateRandom(1,#(self.CurrentGeneration))
    PartnerNotFound = true
    while PartnerNotFound do
      SecondPartnerNum = RandGen:generateRandom(1,#(self.CurrentGeneration))
      if FirstPartnerNum == SecondPartnerNum then
        PartnerNotFound = true


      else
        PartnerNotFound = false
      end


    end
    FirstPartner = self.CurrentGeneration[FirstPartnerNum] --Take 2 random networks from the remaining 50% of the population to breed together

    SecondPartner = self.CurrentGeneration[SecondPartnerNum]

    NextGeneration[i] = Network.New()

    HasNoNeurons = true

    NeuronCounter = 1

    while HasNoNeurons do

      for j = 1,#FirstPartner.Neurons do --Saves a random number of the neurons from the first partner for the new network

        if RandGen:generateRandom(1,2) == 2 then


         NextGeneration[i].Neurons[NeuronCounter] = Neuron.New(FirstPartner.Neurons[j].InputX, FirstPartner.Neurons[j].InputY, OutputToNum[FirstPartner.Neurons[j].Output], FirstPartner.Neurons[j].TileType)


         NeuronCounter = NeuronCounter + 1

         HasNoNeurons = false

        end

      end
      for j = 1,#SecondPartner.Neurons do --Saves a random number of the neurons from the second partner for the new network
        if RandGen:generateRandom(1,2) == 2 then

         NextGeneration[i].Neurons[NeuronCounter] = Neuron.New(SecondPartner.Neurons[j].InputX, SecondPartner.Neurons[j].InputY, OutputToNum[SecondPartner.Neurons[j].Output], SecondPartner.Neurons[j].TileType)

         NeuronCounter = NeuronCounter + 1
         HasNoNeurons = false
        end
      end
    end

    NextGeneration[i]:Mutate()

   end
   self.CurrentGeneration = NextGeneration
   StatsFile = io.open(self.PopulationName .. "Stats.txt","a") --Saves the best fitness score of a generation to the stats file for that population
```

```
  io.output(StatsFile)

  io.write(self.CurrentGeneration[1]:getFitness().."\n")

  io.close(StatsFile)

  self.GenerationNo = self.GenerationNo + 1

end


function Population:savePopulation() --Saves the data of a population to a text file

  local File = io.open(self.PopulationName .. self.GenerationNo .. ".txt", "w")

  local OutputToNum = {A = 1, B = 2, up = 3, down = 4, left = 5, right = 6}


  io.output(File)


  io.write(#(self.CurrentGeneration) .. '\n') --Saves the amount of networks in the population as the top line of the file

  --For each network they are saved in format, Amount of neurons in the network, For each neuron, XCoordinate, YCoordinate, Output, TileType

  --Each network takes up 1 line in the text file

  for i = 1,#(self.CurrentGeneration) do

    if #(self.CurrentGeneration[i].Neurons) <= 9 then
```

```
      io.write("0" .. #(self.CurrentGeneration[i].Neurons))

    else

      io.write(#(self.CurrentGeneration[i].Neurons))

    end




    for j = 1,#(self.CurrentGeneration[i].Neurons) do


      if self.CurrentGeneration[i].Neurons[j].InputX <= 9 then

        io.write("0".. self.CurrentGeneration[i].Neurons[j].InputX)

      else

        io.write(self.CurrentGeneration[i].Neurons[j].InputX)

      end

      if self.CurrentGeneration[i].Neurons[j].InputY <= 9 then

        io.write("0".. self.CurrentGeneration[i].Neurons[j].InputY)

      else

        io.write(self.CurrentGeneration[i].Neurons[j].InputY)
      end


      io.write(OutputToNum[self.CurrentGeneration[i].Neurons[j].Output])
      io.write(self.CurrentGeneration[i].Neurons[j].TileType)
    end
    io.write('\n')
  end
  io.close(File)
end



function Population:loadPopulation() -- Loads a population from a text file
  local File = io.open(self.PopulationName .. self.GenerationNo .. ".txt","r")
  io.input(File)
  local NumOfNetworks = io.read() --Stores the amount of the networks in the population
  local NumOfNeurons = 0
  local NetworkAsString = ""
```

```
 local PosCounter

 local X

 local Y

 local Output

 local Type

 --Loads each network 1 at a time based on how they are stored above

 for i = 1,NumOfNetworks do

   self.CurrentGeneration[i] = Network.New()

   PosCounter = 1

   NetworkAsString = io.read()


   NumOfNeurons = tonumber(NetworkAsString:sub(PosCounter,PosCounter+1))

   PosCounter = PosCounter +2

   for j = 1,NumOfNeurons do

     X = tonumber(NetworkAsString:sub(PosCounter,PosCounter+1))

     PosCounter = PosCounter + 2

     Y = tonumber(NetworkAsString:sub(PosCounter,PosCounter+1))
```

```
      PosCounter = PosCounter + 2

      Output = tonumber(NetworkAsString:sub(PosCounter,PosCounter))

      PosCounter = PosCounter + 1

      Type = tonumber(NetworkAsString:sub(PosCounter,PosCounter))

      PosCounter = PosCounter + 1

      self.CurrentGeneration[i].Neurons[j] = Neuron.New(X,Y,Output,Type)

    end

  end

  io.close(File)

end
```

```
Goal = iup.Alarm("Super Mario Bros. AI","What goal would you like to optimise the AI for?","Distance","Speed","Coins") --User chooses what the goal of the AI is
```

```
Level = iup.Alarm("Super Mario Bros. AI","What level would you like the AI to play?","1-1","1-4","7-2") --User chooses what level they want the AI to learn to play (Overworld,Castle,Underwater)
if Level == 3 then --Due to the way the savestates are stored if the underwater level is chosen the savestate that needs to be loaded is 5 instead of 3
  Level = 5
end


LevelStart = savestate.object(Level) --Creates the savestate object so that it can be loaded later
savestate.load(LevelStart)
Loading = true
while Loading do
  Loading = false
  Load = iup.Alarm("Super Mario Bros. AI","Would you like to load an AI or create a new one?","Load","Create New") --Usr chooses if they ant to load a population or make a new one
  if Load == 1 then
    err, filename = iup.GetParam("Super Mario Bros. AI", filename,"What is the name of  the population you would like to load? %s\n","")


    if err == false then --User pushes the cancel buttton, so it goes back to the last menu
```

```
      Loading = true
    elseif filename == "" or filename == nil then --Checks if the filename is invalid
      iup.Message("Super Mario Bros. AI","Invalid name")
      Loading = true
    else
      local LoadingGen = true
      while LoadingGen do
        err, gennum = iup.GetParam("Super Mario Bros. AI", gennum,"What generation number would you like to load? %s\n","")
        gennum = tonumber(gennum)
        LoadingGen = false
        if err == false then
          Loading = true
        elseif type(gennum) ~= "number" then
          LoadingGen = true
          iup.Message("Super Mario Bros. AI","Please input a number ")
        elseif gennum < 0 then
          LoadingGen = true
          iup.Message("Super Mario Bros. AI","Please input a number that is greater than or equal to zero")

        else

          TestFile = io.open(filename..gennum..".txt","r")

          if TestFile == nil then

            Loading = true

            iup.Message("Super Mario Bros. AI","Could not find the saved population with the name: "..filename..gennum..".txt") --If the file does not exist then the user recieves this message

          else

            io.input(TestFile)

            local size = io.read()

            size = tonumber(size)

            io.close(TestFile)

            local MainAI = Population.New(gennum,size,filename,Goal)


            MainAI:loadPopulation()

            while true do --The main loop for a loaded population

              MainAI:playGame()

              MainAI:evolvePopulation()
```

```
              MainAI:savePopulation()

            end

          end


        end




      end

    end


  elseif Load == 2 then

    local InvalidName = true


    while InvalidName do

      err, AIName = iup.GetParam("Super Mario Bros. AI", AIName,"What would you like to name the population? %s\n","") --User
inputs a name

      InvalidName = false

      if err == false then --If cancel is pushed, goes back a menu
```

```
        Loading = true

      elseif AIName == "" or AIName == nil then

        InvalidName = true

      else

        local NewSize = true

        while NewSize do

          err, Size = iup.GetParam("Super Mario Bros. AI", Size,"How many members would you like in your new populaiton?
%s\n","") --User inputs a number for the size of the population they want

          Size = tonumber(Size)

          NewSize = false

          if err == false then

            Loading = true

          elseif type(Size) ~= "number" then

            NewSize = true

            iup.Message("Super Mario Bros. AI","Please input a number ")

          elseif Size < 3 then

            NewSize = true

            iup.Message("Super Mario Bros. AI","Please input a number that is greater than or equal to 3")
          else


            local MainAI = Population.New(0,Size,AIName,Goal)

            MainAI:savePopulation()

            while true do--The main loop for a new population

              MainAI:playGame()

              MainAI:evolvePopulation()

              MainAI:savePopulation()

            end

          end

        end

      end

    end

  end

end
```