

# Implementing Deep Q Learning for Sonic The Hedgehog 2

---



---

Cardiff University  
School of Computer Science and Informatics

**CM3203 - One Semester Individual Project - 40 Credits**

Author:

Dion James Watts-Evans

Supervisor:

Federico Liberatore

Moderator:

Charith Perera

# Table of Contents

<b>Table of Contents</b>	<b>1</b>
<b>Table of Figures</b>	<b>2</b>
<b>Abstract</b>	<b>3</b>
<b>Acknowledgements</b>	<b>3</b>
<b>1. Introduction</b>	<b>4</b>
<b>2. Background</b>	<b>5</b>
2.1. Project Context	5
2.2. Reinforcement Learning	5
2.3. Q learning and Deep Q learning	6
2.4. The Gym-Retro Emulator Environment	8
2.5. Technologies and Constraints	9
2.6. Project Aims	10
<b>3. Approach</b>	<b>11</b>
3.1. The Approach in theory	11
3.2. The First Approach: Bizhawk	11
3.3. The Second Approach: Bizhawk with Colab	12
3.4. The Final Approach: Colab	14
<b>4. Implementation</b>	<b>15</b>
4.1. Creating Scenario Files	15
4.1.1. Data Files	15
4.1.2. Reward Function	15
4.1.3. Done Function	16
4.2. Creating the Deep Q Network Agent	18
4.2.1. Implementation	18
4.2.2. The Network Class	18
4.2.3. The Agent Class	19
4.3. Bringing Everything Together	19
<b>5. Results and Evaluation</b>	<b>21</b>
5.1. Initial Implementations	21
5.2. Experimentation	21
5.2.1. The Experiments	21
5.2.2. Better Than Random	22
5.2.3. The Effects of the Scenario Files	24
5.2.4. The Effects of Memory Size	25
5.2.5. Generality	27
5.3. Other Findings and Evaluation	29

<b>6. Future Work.....</b>	<b>31</b>
6.1. Improvements and Further Experimentation.....	31
6.2. Implementing for Other Levels.....	31
6.3. System for Predicting Future Reward .....	33
<b>7. Conclusions .....</b>	<b>34</b>
<b>8. Reflection on Learning.....</b>	<b>35</b>
<b>9. Bibliography .....</b>	<b>36</b>
<b>10. Appendix.....</b>	<b>37</b>
<b>Appendix A – Video Examples .....</b>	<b>37</b>

## Table of Figures

<b>[Figure 1.1]</b> The Pictures demonstrate that should the player approach certain obstacles without sufficient momentum, the player will not make forward progress.	<b>4</b>
<b>[Figure 2.1]</b> Diagram showing the relationship between the Agent, Environment, and Interpreter.	<b>6</b>
<b>[Figure 2.2]</b> The Bellman Equation explained.	<b>7</b>
<b>[Figure 2.3]</b> An extremely simple Environment The agent (Sonic) wants to reach the goal state (signpost).	<b>7</b>
<b>[Figure 2.4]</b> the initial Q table for the environment in figure 2.3.	<b>7</b>
<b>[Figure 3.1]</b> Diagram showing how the systems interacted at this point in the approach, interactions in red had yet to be implemented.	<b>13</b>
<b>[Figure 3.2]</b> diagram showing the final state of the project and how various systems are interacting.	<b>14</b>
<b>[Figure 4.1]</b> A steep hill that Sonic would struggle to get up from his current position.	<b>17</b>
<b>[Figure 4.2]</b> Slightly back from the hill in figure 4.1 is a spring that can propel Sonic up the hill.	<b>17</b>
<b>[Figure 5.1]</b> Initial Experiment plan.	<b>22</b>
<b>[Figure 5.2]</b> Generality Experiment plan.	<b>22</b>
<b>[Figure 5.3]</b> A graph showing the maximum x position over time for a random agent.	<b>23</b>
<b>[Figure 5.4]</b> A graph showing the maximum x position over time for our control agent.	<b>23</b>
<b>[Figure 5.5]</b> A graph that compares the progression of the maximum X position achieved by the scenario file agents over time.	<b>24</b>
<b>[Figure 5.6]</b> A graph comparing act clears obtained by both agents with loose done functions.	<b>25</b>
<b>[Figure 5.7]</b> A graph comparing the maximum X position achieved by each memory experiment agents over time.	<b>26</b>
<b>[Figure 5.8]</b> A graph comparing act clears obtained by the control agent and both altered memory agents.	<b>26</b>
<b>[Figure 5.9]</b> A graph comparing the maximum X position achieved by the trained and the untrained agent in act 2.	<b>27</b>
<b>[Figure 5.10]</b> A map of Emerald Hill Zone act 1.	<b>28</b>
<b>[Figure 5.11]</b> A map of Emerald Hill Zone act 2.	<b>28</b>
<b>[Figure 5.12]</b> A graph comparing act clears obtained by Trained and Untrained agents on act 2 of Emerald Hill Zone.	<b>29</b>
<b>[Figure 6.1]</b> A map of act 1 of the second level of the game, Chemical Plant Zone.	<b>32</b>
<b>[Figure 6.2]</b> A map of act 1 of the Third level of the game, Aquatic Ruin Zone.	<b>32</b>

# Abstract

Machine learning implementations for video games have become somewhat of niche online in recent time, with models being trained on a variety of video games for education and entertainment purposes. This report covers an implementation of a reinforcement learning algorithm known as Deep Q Learning for the video game Sonic the hedgehog 2. The project aims to create a machine learning model that can complete the first level of the game, with speed of completion being a secondary target. The model will also be tested on its generality or ability to beat levels not previously seen by it.

The report will cover the implementation of this model, from the basic ideas needed to implement it, to initial attempts for a working solution, and then finally going into detail about the implementations describing how various functions enable and affect the performance of the model, We will discuss the results, the successes of the project and the failures. This report will cover how these results came about by adjusting parameters that manage how the agent performs.

Results will be analysed and discussed to determine if the solution provided is successful, ideas to expand the solution in future will be put forward, and we will reflect on the learning implementing this project has caused. Overall, this report discusses the journey of the implementing a machine learning model, the positives and the negatives.

# Acknowledgements

I would like to thank my partner Ffion, for her unfaltering support during the pandemic and this project, without her help I don't know what state this project would be in.

I would like to thank all the members of Duct Tape House and my Friends for the memories and the support they have given me these last four years.

Finally, I would like to thank my supervisor Federico Liberatore for all the insight and guidance he has provided me with.

# 1. Introduction

Machine learning for video games has been a good indicator of forward progress in the accessibility of machine learning field. Understanding large data sets and how a model could be used to process that data can be hard even for experts who are trained to do so. Using video games can make for a good platform for explaining and developing interests in machine learning technology.

This project aimed to implement a machine learning model for the video game Sonic the Hedgehog 2 for the Sega Mega Drive<sup>1</sup>. The game provides several unique challenges for a machine learning model that other games might not.

- Levels are long horizontal obstacle courses as opposed to single screen challenges meaning the model will have to adapt to rapidly changing observations.
- The game features branching level design; thus, decisions early in the level can severely impact performance later in the level.
- Several sections of the level require significant forward momentum to overcome meaning the player needs to approach at speed to progress.  
(see Figure 1.1)



[Figure 1.1] The pictures demonstrate that should the player approach certain obstacles without sufficient momentum, the player will not make forward progress.

- Levels often feature complex geometrical patterns which may impact the AI's judgement of similar situations.

The project scope was limited to the first level or zone of the game, Emerald Hill Zone. Each zone is split up into two acts, the project will focus on the first of these two acts, but some consideration will be given to second act when discussing if the AI can achieve some level of generality.

The implementation of this project shows some hopeful results, but leaves open some interesting questions, and provides several avenues for future improvement, due to some unfortunate issues. This will be covered in more detail in sections 4,, 5 and 6, discussing the implementation and the problems with it.

This report will cover said implementation of the machine learning model from beginning to end. Starting with research into the implementation of the chosen machine learning algorithm, Q Learning. Following the initial discoveries and trials for the basic

---

<sup>1</sup> Known more commonly as the Sega Genesis internationally

implementation project. It will then follow on to compare several of the methods I used by examining the data gathered over multiple training sessions.

## 2. Background

### 2.1. Project Context

This project exists in the wider context of video game machine learning projects. Several Sonic the Hedgehog AIs already exist, Most commonly using the Neuroevolution of Augmenting Topologies algorithm, or NEAT algorithm. The NEAT algorithm is relevant for this use case as it keeps several distinct topologies<sup>2</sup>, and evolves by augmentation. Meaning several possibilities are considered. A good example of the NEAT algorithm for Sonic the Hedgehog is demonstrated by Lucas Thompson in his Implementation tutorial.[1]

While Q Learning for Sonic the Hedgehog 2 has been done such as the implementation by Alex Kaplan[2], there are extra considerations this project will make when implementing. Firstly, we want our implementation to play the game quickly. A key part of Sonic the Hedgehog games is fast movement and preservation of momentum; we want to reward our AI for speed. Secondly, we want to test the generality of such an implementation. We will test our trained AI on the second level and compare it to a new AI to see if the later AI performs better.

### 2.2. Reinforcement Learning

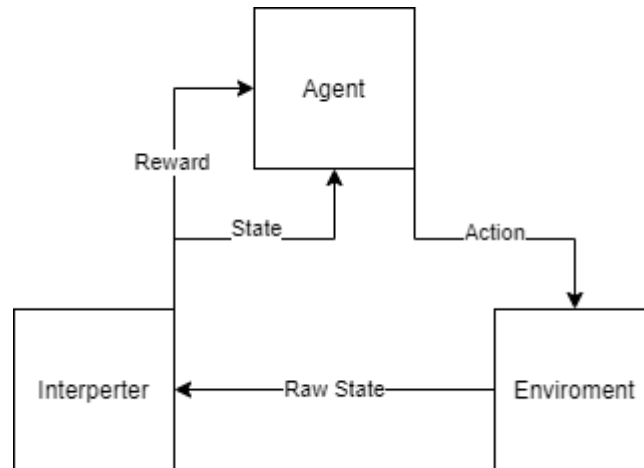
The project uses a Deep Q Learning neural network to select actions for the computer player. Q Learning is a Reinforcement Learning technique. This section will go into detail about Reinforcement Learning as a machine learning paradigm, whereas Q learning will be covered in section 2.2.

Reinforcement Learning is a machine learning paradigm that seeks to optimize a given problem using a defined set of actions. The computer player, referred to as the agent takes an action and receives a reward based off the state following the action. The agent does not know which move to make at first. It learns which moves yield a high reward by first taking the move and noting the reward obtained[3].

The way the agent works in theory is quite simple. Given a state, the agent performs an action. The environment is then changed according to this action, The new state of the environment is passed to the interpreter which provides the agent with the reward for this action and the parts of the state that are relevant to the agent.

---

<sup>2</sup> A topology is a specific neural network, kind of like a genome in biology.



[Figure 2.1] Diagram showing the relationship between the Agent, Environment, and Interpreter.

As previously mentioned the agent can only know what moves to make after it has already made them. This presents a problem for reinforcement learning in the form of the Explore/Exploit Trade-off. Put simply, the agent can pick one of explore or exploit at a given time. The agent can learn, or it can act according to what it has learned for a given action, it cannot do both.


The solution used by this project and most other implementations is called epsilon greedy. Epsilon greedy stores a value called the epsilon value that starts at some chosen value between 0 and 1 and decays by a percentage periodically. A random number is generated, if it is less than epsilon the agent acts randomly, otherwise it acts according to the model.[4]

Hopefully, the application of this learning paradigm for video games is apparent. It mimics how we as people learn new concepts by trying and remembering new things, while also providing a reactive model that can approach different situations differently.


## 2.3. Q learning and Deep Q learning

This section will cover Q Learning and Deep Q Learning in more detail, but put simply, Q Learning is a reinforcement learning method that uses a data storage called the Q table to determine what action to take. The Q table maps rewards or Q-values to state action pairs. This Q table is updated using the Bellman equation. As figure 2.2 demonstrates, the Bellman equation assigns Q-values for a given state based off the expected reward of the action. The primary difference between Q Learning and deep Q Learning is how the Bellman equation is used. Q Learning updates the Q table whereas deep Q-Learning uses the bellman equation to update the weights of a neural network that predicts Q-values of actions for a given state.[5]


$$Q^{\pi}(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t, a_t]$$



Q-Values for the state  
given a particular state



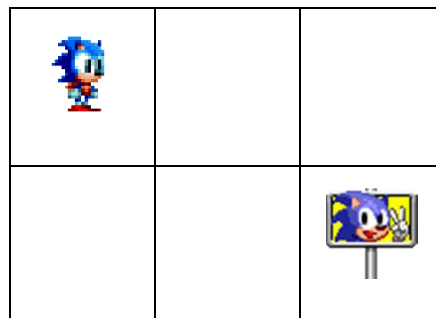
Expected discounted  
cumulative reward



Given the state and action

[Figure 2.2] The Bellman Equation explained[6].

Consider the environment in the figure 2.3. This simple environment has six possible states. The start state shown in the figure, four intermediate states, and the goal state. With the assumption that all the agent can do is move on the cardinal directions, the Q table is initialised by providing a score of 0 for each action at each state, as seen in figure 2.4. The Epsilon greedy algorithm can then be used to fill the Q table using the Bellman equation utilising a chosen reward function. Q Learning is suitable for simple examples like this but what about larger more complicated environments?



[Figure 2.3] An extremely simple Environment The agent (Sonic) wants to reach the goal state (signpost)

	Up	Down	Left	Right
start	0	0	0	0
1,2	0	0	0	0
1,3	0	0	0	0
2,1	0	0	0	0
2,2	0	0	0	0
end	0	0	0	0

[Figure 2.4] the initial Q table for the environment in figure 2.3



Consider again the environment from figure 2.3. If an enemy was included that moves randomly throughout the six spaces, our simple six state system increases to a 36 state system. Increasing the complexity of an environment increases the size of a Q table multiplicatively. Because of this for more complicated environments, we need a more space efficient solution to determine the best action from a given state.

This is where Deep Q Learning can be applied. In Deep Q Learning we create two neural networks, the main network, and the target network. These networks are used to assign Q-values to state action pairs. Two networks are used to improve learning stability, limiting the AI's ability to overestimate. Much Like how the Bellman equation updates the Q-values in the table, the Bellman equation is used to update the Q-values in nodes in the neural networks. By using neural networks, we should train an artificial intelligence that can react more generally to new scenarios than a standard Q Learning environment could. While also dealing with the potential memory problems mentioned earlier[7].

## 2.4. The Gym-Retro Emulator Environment

You may be familiar with the Open AI gym environment[8]. It is a Python module for use with artificial intelligence. Open AI gym exists mainly to compare different artificial intelligence models on simple environments, and as a teaching tool for machine learning.

To use the module, you initialise an environment and go through the states in order using the step function, providing an action to do in that state. After using the step function the environment provides a reward value, a boolean value letting you know if the environment should be reset, Miscellaneous information about the environment, and most importantly an observation of the environment.

This observation is typically used as a state for machine learning algorithms. The observation can be anything from a set of variables providing information about the environment, to a graphical representation of said environment. Using this observation and the reward value we can update our Q-values on a Deep Q Learning model as mentioned earlier. This makes Open AI gym incredibly easy to set up for our use case, something that would greatly benefit the project if discovered earlier<sup>3</sup>.

Gym retro is very much the same as gym in how it is used. You set up the environment and step through it action by action in the same way, gym retro provides reward functions, information and observations exactly like gym. The key difference is that gym retro comes with several emulators for use.

An emulator is a reconstruction of physical hardware using software. They are primarily used to simulate old video game consoles to play backups created from old video game cartridges. Emulators can be computationally expensive, However older devices such as the Mega Drive tend to be easier to emulate due to advances in hardware and optimisation of code.

---

<sup>3</sup> See section 3.4.

Gym retro's emulator differs from most other emulators however. Instead of playing back in real time Gym's step system is implemented, for video playback the emulator can create something called a *bk2 file*, which is essentially just a series of inputs for the emulator to take. A function can then be run to create an mp4 file using these inputs. In addition, Gym retro allows us to create custom scenarios for gym environment. Meaning we can change the reward function and done<sup>4</sup> functions. We can write these functions using simple Lua scripting, to calculate the reward using variables obtain from a JSON storage of RAM values we can select[9].

## 2.5. Technologies and Constraints

There are some other technologies that are worth mentioning as a precursors to the project. Firstly TensorFlow, a machine learning package for Python will be used to implement the neural networks. TensorFlow offers easy to implement flexible neural network and provides debugging tools and guides to make implementing a neural network more pleasant[10].

The project encountered problems due to the lack of processing power available to successfully run a machine learning project with a sizeable memory in a reasonable amount of time. Because of this another computing solution had to be found. Google Colab was chosen due to its ease of use and access, and its variety of features that would be appealing for the project. Google Colab is a cloud-based Python environment that provides access to High RAM environments with powerful GPUs. The environment is structured like a jupyter notebook and has easy installation of modules for the project. As a bonus the platform easily connects to Google drive for convenient storage options[11].

The main constraints of the project are in decreasing order of priority:

- Time
- My personal understanding of machine learning implementation
- Computational power provided by Colab

Time is a big limiting factor; time management was a big weakness for this project.. With the project being so limited in time, proper time management is important, this constraint will be counteracted by regularly meeting with the project supervisor to ensure the project is progressing smoothly.

The next limiting factor is understanding of machine learning. Machine learning is a new, deep incredibly complicated field. Because of this what code is doing may be harder to parse than what is normally expected. To counteract this constraint, I will have to be diligent in my implementation and my reading.

Finally, the final constraint on the project is computational power provided by Google Colab. While Colab provides significantly more resources than the project would otherwise have access to, there are limits. This is counteracted by upgrading to Colab pro and timing running the project to maximise resources.

---

<sup>4</sup> The done function determines when a session is done and when a new one should begin.

## 2.6. Project Aims.

As discussed in the initial plan, the project has three aims: The second aim has been slightly amended as ethics approval was not obtained due to timing issues, so the experiment was not run. Therefore the aims of the project are as follows:

1. Demonstrate that Q learning can successfully be implemented for Sonic the hedgehog 2.
2. Demonstrate speed improvement.
3. Demonstrate some degree of generality.

For this project, a successful implementation is one which demonstrates an improvement over purely random gameplay and can successfully beat the first level of the game. To demonstrate speed improvement the model will need to improve upon its time and achieve a low time compared to other implementations found online. To demonstrate generality, a model trained on act 1 will be run on act 2 alongside another untrained model. The model will demonstrate generality if performs better than the untrained model.

## 3. Approach

### 3.1. The Approach in Theory

Referring back to figure 2.1, the approach in theory becomes quite obvious. There is an environment, some system that runs Sonic the hedgehog 2 and allows for data output and input via a computer interface. An agent, that given an input will produce an action for the player to make in the environment. And finally, and intermediary between the two. Some way for the two systems to interact meaningfully. All three of these components are necessary for the project to be implemented successfully.

The first two implementations fail on the implementation of one of the three necessary components and had to be iterated on to arrive at the final approach. Before covering the implementation of the final version of the project we will cover what was worked for the approaches that came before and what about them was unsuitable for the project. Every approach uses the same implementation for the neural network and will be discussed more in section 4. The key differences between these implementations lie in the code that surrounds the agent, and the power afforded to the network.

### 3.2. The First Approach: Bizhawk

The initial search for an emulator lead to the Bizhawk emulator[12]. Bizhawk is a multi-console emulator written in C# for casual gaming and the creation of tool assisted gameplay. Particularly useful is Bizhawk's Lua scripting interface that comes with several powerful tools that would be useful for the project. The Lua interface provides tools to access RAM values<sup>5</sup>, the ability to advance frame by frame, and importantly the ability to toggle which buttons are held.

Initial testing of a Bizhawk implementation went well. Methods to extract and display RAM values were found quickly, and a system for inputting moves was implemented shortly after. Initial trouble for Bizhawk came in implementing a method to communicate between the Lua script in Bizhawk and Python. Data transfer using temporary files was deemed to slow a solution, so another approach had to be taken.

Looking at other Bizhawk projects that needed to input and output data, a networking solution looked to be the most appropriate. Web sockets were to be used to easily transfer data between the two systems. Having used the Python sockets module before, implementing a socket server in Python was not too difficult. The server was implemented in Python because I believed it would be easier to connect to a network in Lua than to establish a server in Lua. This proved to be true as the Lua sockets implementation was quite barebones and left a lot of the work up to the user. the systems for connecting to an existing server however were quite easy to use.

---

<sup>5</sup> This is useful for getting data for our *reward function*.

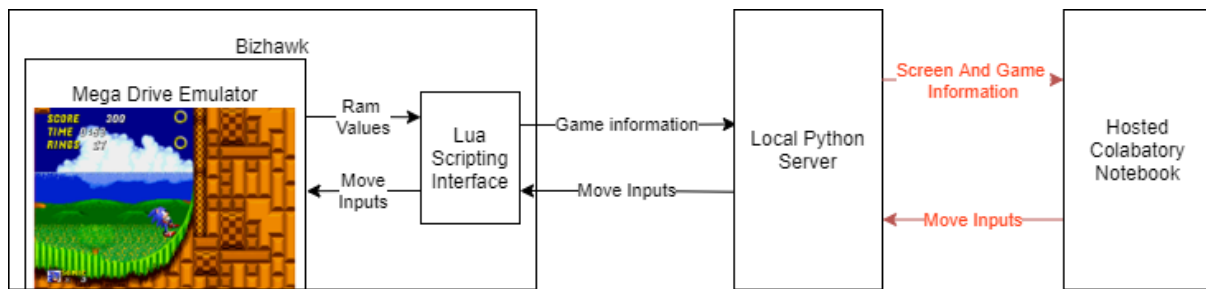
Getting Bizhawk to work with Lua sockets proved to be a problem despite the ease of use. Bizhawk uses its own installation Lua specifically for its own use cases. Bizhawk's documentation does provide information on its networking capabilities, however the system seems rather archaic. The Lua installation would need direct access to the Lua sockets module found online. This was initially troublesome, but installation came down to putting the resources in the right file locations.

At this point the approach was progressing well. Bizhawk's Lua script extracted RAM values and sent them to a Python server where calculations were done and then sent back to Bizhawk to display in real time as a proof of concept. All that needed doing was creating a neural network and modifying the Lua code to accept a move input and advance a certain number of frames, before accepting another input. The Lua code had issue consistently accepting more than one byte of input. It was decided to work inside this constraint, as it would not be an issue with a careful implantation. Using a random move trial, the system was functional. The Python server would send a move to Lua client to execute in Bizhawk, then send commands to advance to the next action. Functionality to reset the environment was also implemented. Everything was looking promising for this approach to the problem. However, after implementing the deep Q learning module, it became apparent that my hardware could not support this approach.

### **3.3. The Second Approach: Bizhawk with Colab**

After it became obvious that hardware being used would not be able to run the project, several options had to be considered. One option would be making use of university computers, while accessing them physically was initially impossible, later becoming impractical. Using VPN software to access university hardware was always an option worth considering; however, another technology considered and eventually used was Google Colab. Google Colab was recommended by peers who had previously implemented a machine learning project. Initial observations of Colab looked promising, Colab offered powerful processing options and implementing what had already been written seemed like it would be quite simple.

After moving the code on to Google Colab, it was discovered what would ultimately doom this approach to the project. Google Colab makes use of hosted runtimes on Google's own servers. A side effect of this is that Google has made it somewhat impractical to send data between a local pc and the Colab session in real time. The machine a Colab session is run on is accessed through Google's servers not directly, this makes setting up the previously implemented socket system impossible. Another issue that arose from this is that the Python server in addition to connecting the deep Q learning model to the environment was also gathering data for the model by capturing the screen of the emulator using a screenshotting module called MSS. This meant that a large portion of the data going to Colab was graphical, meaning the data transfer method needed to be efficient if the program was going to run effectively.



[Figure 3.1] Diagram showing how the systems interacted at this point in the approach, interactions in red had yet to be implemented.

Initial research to deal with this issue found the option to connect a Colab session to a local jupyter notebook runtime. Initially this sounded very promising for implementing the system described in figure 3.1. Setting up this connection revealed that connecting to a local runtime like this connects to a local runtime instead of a hosted runtime. Effectively loosing access to the powerful processing required for this project, ruling out this option.

Following on, the next option appeared to be creating an SSH tunnel<sup>6</sup>. This seemed to be the most common solution suggested online. Through external secure SSH providers it was possible to import modules into Colab that would allow creation of an SSH tunnel. I did not have much experience with using SSH, but I was initially able to create an SSH connection, issues instead arose from using the SSH tunnel. Implementing a successful SSH connection in Python was not possible within the time plan, and thus a new solution had to be found.

More unconventional options were considered after these initial failures. Google Colab has an incredibly robust system for interacting with Google Drive which the project would later take full advantage of. But this gave me the idea to use Google sheets to send the data. While this would be slower than ideal, at this point in the project it was worth considering to at least prototype the solution. After setting up a spreadsheet in Google sheets and implementing a way to obtain that data on Colab, it was time to implement a way for the Python server to communicate with Colab.

Google has a Python module for this use case called Gspread. By setting up a Google Cloud Platform[13] project and giving it the correct permissions, Gspread was able to communicate with Google Sheets and as a result, Colab. After creating functions to upload and download data to and from the sheet, I created a timing system so that each system knew when to upload and when to download. This worked by changing a control value in the spreadsheet, sadly this didn't work. The Python server would set the control value, then immediately begin checking the sheet to see if it had changed. In practice this prevented Colab from changing the value due to sheet being in constant use. Things were not looking good for the project at this point. Despite first having a working implementation, the processing power was not there. Then when a method to obtain that processing power was found, it was not able to properly be utilised. Other machine learning implementations were investigated for inspiration and Gym Retro was discovered.

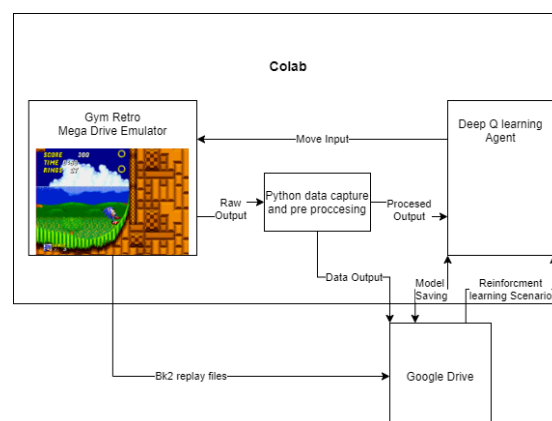
<sup>6</sup> Put extremely simply an SSH tunnel is an encrypted connection between two sources.

### 3.4. The Final Approach: Colab

Initially for the project it was assumed that the Open AI gym module, used for many machine learning examples, was limited to the preinstalled environments that come with the module. However, examples sourced online cited using Open AI gym for their machine learning project in video games. This sparked deep research into the system, leading to the discovery of Open AI gym retro, the extension of gym that emulates retro video game consoles.

This discovery meant that the entire project could be hosted on Colab. This comes with several advantages. Having one system instead of three meant there were less fault points, this also would decrease the latency of the project which is desirable considering the possibility of long runtimes of the project. Using Colab exclusively also meant that there was no hardware burden on personal computing devices which would have a positive effect on performance.

Installing and implementing gym retro was a simple process, the module could easily be installed through PIP and only some minor changes to my code was needed to make it function correctly. The primary issue in implementing gym retro on Colab came in the form of playback. Hosted runtimes on Colab have no displays or display drivers. Meaning gym retro had to be run in a no display mode. This was fine for running the project, but some visually output would greatly increase the appeal of the end product. Gym retro has a simple movie recording system that allows for recording of input in the form of an archive called a bk2 file. These bk2 files can then be taken and turned into an mp4 using a command that comes packaged with retro. The documentation for gym retro mentions this as an argument for the environment object, what the documentation fails to mention is that this can be toggled on and off with a simple function, allowing recording only after a certain point. This is useful for the project, as there are moments of gameplay that do not need to be recorded.



[Figure 3.2] Diagram showing the final state of the project and how various systems are interacting.

Gym retro's implementation led to an overall improvement in the project, allowing it to be entirely independent of local computing power and being centralised on mostly one service barring saving and loading of data. Implementing the project in this way vastly increased the speed and efficiency of the project.

## 4. Implementation

### 4.1. Creating Scenario Files

#### 4.1.1. Data Files

To know what rewards to give the agent and when to reset the environment, gym retro makes use of json files it calls scenarios. As we can see in *q.json* the basic use case is relatively simple. While it is possible to calculate rewards and done flags entirely within this json file by assigning changes in certain RAM addresses a reward value or a true or false flag to reset the environment. This was not a particularly robust system so instead we point the environment towards a Lua function for the reward and done flags.

This function has access to certain RAM values from the emulator as mentioned earlier. The environment access these RAM values through a file known as *data.json*. The *data.json* file is a list of RAM addresses, a name and how the data should be taken<sup>7</sup>. Most games already supported in gym retro come with a *data.json* file, however the project needed access to a few more RAM addresses, specifically the in-game timer and the current act. A full list of Sonic the Hedgehog 2 RAM values was easy to find on Sonic Retro[14], but the RAM addresses hosted here are stored as hexadecimal bites. These worked perfectly for Bizhawk but gym retro's RAM addresses were regular decimal numbers. A comparison of the hexadecimal values to the decimal values revealed they were not the same numbers. Comparing two address shows that the hex and decimal values were offset by the same amount. So, to get the decimal values simply compare the hex value of the wanted address to another hex with a known decimal value.

With the RAM addresses set up correctly the *reward* and *done functions*<sup>8</sup> could now be created. The done function depends on actions taken in the reward function to work so we will cover the reward function first. The reward function shown is my final test iteration. Other reward functions were used to gather data for comparison, the same is true of the done function. We will discuss these changes in more detail in section 5 where we will compare how changing parameters effected the results.

#### 4.1.2. Reward Function

The reward function first initialises certain values, At first this was done because the example reward function did this, however after implementing the checkpoint system this became a necessity due to a peculiar issue with how lua handles 0 and nil values. Assigning values at the start ensured the function worked correctly so it was kept. There are three ways the agent can influence their reward:

- Making forward progress in the level.
- Collecting or losing rings.
- Advancing past a checkpoint.

---

<sup>7</sup> Upper/lower byte, big/small endian, etc.

<sup>8</sup> Found in *positive loose.lua*.



The agent is rewarded for forward progress in a simple manner. The system tracks Sonic's movement through the level through his *X position* in pixels from the beginning of the level. When Sonic achieves a new *X position* that is greater than the previous maximum i.e., closer to the goal. The agent begins gaining reward equal to the distance between Sonics current position and their previous position. This implementation encourages forward exploration and speed. In order for the agent to gain reward it has to make progress in the level, and since it gains reward based of distance travelled, traveling a further distance in between steps increases the reward gained and thus traveling forward at a high speed provides more reward than traveling forward at a low speed.

The second way reward can be influenced is via rings. Rings in Sonic the Hedgehog act as a sort of health, if Sonic has at least one ring you do not lose a life when you get hit, you instead lose all your rings. To ensure Sonic can progress through the level, when sonic has zero rings the agent is rewarded for collecting one ring. The agent is not rewarded for any additional rings collected as it is conducive to fast gameplay. The agent loses reward when Sonic gets hit, as this generally slows Sonic down. To ensure a cycle of losing and gaining rings does not happen, the reward for getting Sonic's first ring is less than the reward lost for getting hit.

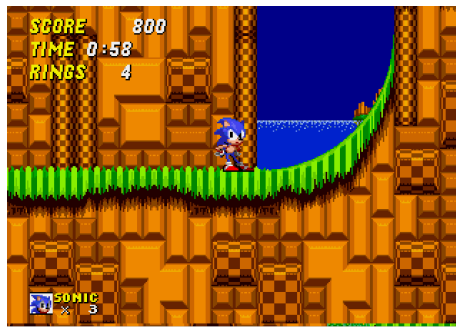
Finally, there is the checkpoint system. The checkpoint system acts as a way to encourage forward progress and one of the ways the done function can reset the environment. The way it works is quite simple. When ever Sonic reaches an *X position* that is a multiple of 100, the agent gets a reward value of 100. The function then checks for the next multiple of 100 until a multiple has not been passed. Then, to start gaining more reward sonic must pass a new multiple of 100. This system gives the agent the biggest reward so it should really encourage the agent to make forward progress.

#### 4.1.3. Done Function

As previously mentioned, the *done function* exists to tell gym retro to reset the environment. It is similar to the *reward function* in that it returns a value determined by values gathered in the data file. Values created in the *reward function* can also be used in the *done function*, this allows us to track non RAM values like Sonic's *maximum X position*. Using these values, it can be determined if the environment should be reset by returning true or false. The *done function* makes the following considerations to determine if the environment should be reset:

- Has sonic gone too far backwards?
- Has sonic lost a life?
- Has sonic recently passed a checkpoint?
- Has sonic entered a special stage?
- Has sonic cleared the level?

Knowing how far back to let sonic go is important. Take this hill for example.



[Figure 4.1] A Steep hill that Sonic would struggle to get up from his current position.

Due to Sonic the Hedgehog's physics system a large amount of speed is needed to pass obstacles like the hill shown in figure 4.1. Holding right to run up the hill simply makes sonic fall down the hill. In order to get this speed sonic needs to either have time to build up to a higher speed or accelerate quickly via methods provided by the environment.



[Figure 4.2] Slightly back from the hill in figure 4.1 is a spring that can propel sonic up the hill.

As we can see in figure 4.2, the key to making forward progress sometimes lies backwards. Determining the right amount of backward distance allowed is key to a functional agent. Later levels experiment with this concept to a much larger degree than emerald hill, and this approach would not work for many levels later in the game but should make creating an agent for acts 1 and 2 more efficient. The main implementation checks if Sonic has gone too far backwards by checking to see if Sonic's *x position* is 400 pixels less than the *maximum obtained x position*. This allows the agent to take advantage of situations like the one displayed in figure 4.1 and 4.2, while also optimising the speed of the training, ending attempts that go too far backwards.

The function also makes use of the checkpoint system. If during the *reward function* a new checkpoint is passed, a variable called *check\_flag* is set to 1. This flag is checked in the *done function*. If the flag is not set to 1 the function increments a counter by 1. If this counter reaches a certain value, the environment is reset. The check flag is reset to 0 at the end of each *done function*, and the counter is reset when ever sonic passes a checkpoint. This part of the *done function* ensures that the agent does not get stuck at one point it cannot pass and waste valuable processing time.

The rest of the *done function* is mostly checking important RAM values. While we could let the agent play until it gets a game over, the project assumes that avoiding losing lives will improve overall play. The other cases the function is checking for is if Sonic enters a special stage and if Sonic clears the level. It is highly unlikely the model would enter a special stage<sup>9</sup> but this was implemented as a failsafe.

Creating the Scenario files had to consider both optimization and giving the agent enough room to experiment as we will discuss in section 5. The methods described above are my original implementations that were iterated on to their current state then compared against other methods.

## 4.2. Creating the Deep Q Network Agent

### 4.2.1. Implementation

The implementation of deep q learning for this project was made following the example demonstrated in[15]. The implementation uses tensorflow 1 as opposed to the more recent tensorflow 2. This was not discovered until the project was ported to Google Colab, where the agent had to be run in compatibility mode for tensorflow 1. Tensorflow 2 comes with this compatibility mode for legacy reasons but in the future it would be better practice to upgrade to the newest supported version.

The implementation consists of two classes, the *deep q network* class and the *agent* class. The *deep q network class* contains the code to create and store a recursive neural network. While the *agent* uses *the deep q network* class to generate it selected move. The *agent* creates two networks as mentioned earlier as this stabilises the model and prevents the agent from focusing on one solution. A good way to think about the system is to imagine the networks as tools that the *agent* class controls.

### 4.2.2. The Network Class

The *network* class contains two main function and then another two functions for saving and loading. The first function initialises the object when it is made, the second builds the neural network contained in the object. The two *save* and *load* functions conveniently make use of tensorflow's built in model saving. It was initially planned to save the agent object to save the current replay memory, however tensorflow cannot save objects, and all object saving modules attempted had trouble saving tensorflow variables, so the implementation of object saving was scrapped.

The class initialises with several variables provided to it via the *agent* class, mainly learning and control parameters. The function assigns these variables then begins a *tensorflow session*. Sessions are tensorflow 1's class for running operations on neural nets. It then constructs the neural network using the *network class' build network* function, hands the variables to the *tensorflow session* and defines the *checkpoint files*. By using tensorflow sessions in this way, most of the interactions with the network done in the *agent* class go through the *tensorflow session* class, greatly simplifying its use once set up correctly.

---

<sup>9</sup> To enter a special stage sonic must collect 50 rings then jump into a special portal that open above in game checkpoints(not the checkpoints used in the reward function)

The main function of the *network* class other than creating the *tensorflow session* is building the neural network. The neural network accepts an *observation* as an input, typically a 224 by 320 by 3 array which represents the RGB channels of the screen image. This image is then passed through three *2d convolution layers* with *relu activation*. We parse the input in this way to put into a form that the neural network can interoperate more information from. The convolved input is then flattened and passed through two *dense layers* to obtain the *Q values*. To optimise the network, we calculate the loss between our *Q* and our *Target networks*, and we attempt to minimize that using *tensorflow's adam optimiser*.

#### 4.2.3. The Agent Class

The agent class is responsible for providing the network class with the appropriate information and collecting and using the outputs received. Much like the *network* class the *agent* class initialises starts by initialising its variables. It then creates two neural networks, *Q\_eval* and *Q\_next*. *Q\_eval* is used to obtain the selected move, while *Q\_next* is used to tweak *Q\_eval* after a set interval to stabilise the learning process.

The *agent* stores previous actions in the form of *transitions*. Several arrays are kept with length up to the maximum size determined by the class, and for each index there is an array that stores, the *action taken*, the *reward* and the *state* of the environment before and after the action was taken. This *transition memory* is used to update the neural networks in the *learn* function. When the *transition memory* fills, the agent starts writing over the oldest *transitions*. This way the *transition memory* is usually full of recent *transitions*.

To learn the agent provides the networks with a selection or batch of *transitions* from the *transition memory*. These batches are fed into the neural networks. A *target network* is then using the reward batch and the *Q\_next* network. The *Q\_eval* network then uses the previously mentioned optimizer. Occasionally the graph for *Q\_eval* is updated using parameters from *Q\_next*. This happens every time the memory reaches a certain threshold.

### 4.3. Bringing Everything Together

All the elements previously discussed are then brought together in a Colab *.ipynb* file. This file also follows the framework established in the tutorial. However, the *.ipynb* file has been more heavily modified for the projects use case. After initial testing there was a structure to the *.ipynb* file that worked. Copies of that *.ipynb* file were made and then slightly altered for the experiment in section 5.

Each experiment has its own folder, where its neural networks, results, *Bk2 files* and *scenario files* as kept. When the experiment starts the *.ipynb file* first must set up by mounting to Google Drive, importing the necessary modules and copying over the scenario and other files. Before the code is run there is a manual check step to ensure that everything is set up correctly. This seems trivial but reminders to check the code and the files before running caught errors that may have otherwise slipped through.

The bulk of the *.ipynb file* can be split into three categories; setup, filling the memory, and training/playing. The setup simply initialises the environment and the *agent* and loads any previously made neural networks if needed. After the setup the script fills the *transition memory* by allowing the *agent* to perform random moves. This done so that when the *agent* starts to select from the memory there will always be *transitions* to select. Once the memory is full the agent is in full control, using epsilon greedy to explore and exploit the solution. The agent gets to select a move every 10 frames or every sixth of a second. That move is then carried out for the next ten frames until it selects a new move. The *rewards* are calculated on every frame and summed and then stored as a *transition*. The script gathers data from each attempt and amends them to a pandas data frame. This data frame is appended to the end of a csv file and cleared every 10 attempts. The csv is then backed up to Google drive.

## 5. Results and Evaluation

### 5.1. Initial Implementations

The primary aim of this project was the implementation of working model. For the purposes of this project a working model is a model that can beat the level at least once. The initial version had a very strict *done function* and heavily rewarded forward movement through an overtuned checkpoint system, whilst penalising backwards movement to a large degree. This resulted in a model that made fast progress through the level but was not able to progress up steep hills that blocked its path.

The first major overhaul was a rescaling of the *reward function* to smaller numbers, and a removal of negative reward for moving in the wrong direction. And while this had a negative effect on the initial learning rate of the model, the player was making it further into the level but was still experiencing trouble with hills. As we can see in the video example in appendix A1, the agent makes good initial progress through level, then approaches the hill shown in figure 1.1. The agent attempts to run up the hill without sufficient speed and begins to fall down the hill. The agent correctly responds to this and enters Sonic's ball mode<sup>10</sup> and begins gaining speed. However, just before colliding with the spring that would get the agent passed the hill, the *done function* resets the environment.

This signalled another change that should be made. The *done flag* needed to be adjusted. The distance sonic could go backwards and length of time without making progress were both increased dramatically whilst the program was still running. This could be a good idea to expand upon in a future implementation as we will explore in section 6. This change ultimately led to the first successful clear of act 1 as can be seen in appendix A2.

### 5.2. Experimentation

#### 5.2.1. The Experiments

In the creation process, parts of the implementation were tweaked until a model that could beat the level was created. This model was used as the control in a series of experiments to determine if the model could be improved by tweaking some of the scenario files. The initial outline for the experiments looks like this. Each model was to be run for 5000 attempts, this allows for data capture of the agent in training and performing as a trained agent.

---

<sup>10</sup> When Sonic is in a ball they gain increased amounts of speed going down slopes allowing for easy clearance of obstacles.

	Movement selection	Reward Function	Done Function	Transition Memory Size
1	Random	Positive only	Loose	6000
2	Model	Positive only	Loose	6000
3	Model	Positive only	Strict	6000
4	Model	Negative	Loose	6000
5	Model	Negative	Strict	6000
6	Model	Positive only	Loose	18000*
7	Model	Positive only	Loose	2000

[Figure 5.1] Initial Experiment plan.

These experiments were designed with certain questions in mind, with experiment 2 serving as the control. Is the implementation provably better than a random agent? The biggest problem found in initial testing was consistency, the model almost seems to forget, only to suddenly remember, can changing the amount of transition memory help this problem. What effects do the reward function and done function have on the agent? These are the questions the first 7 experiments were made to examine.

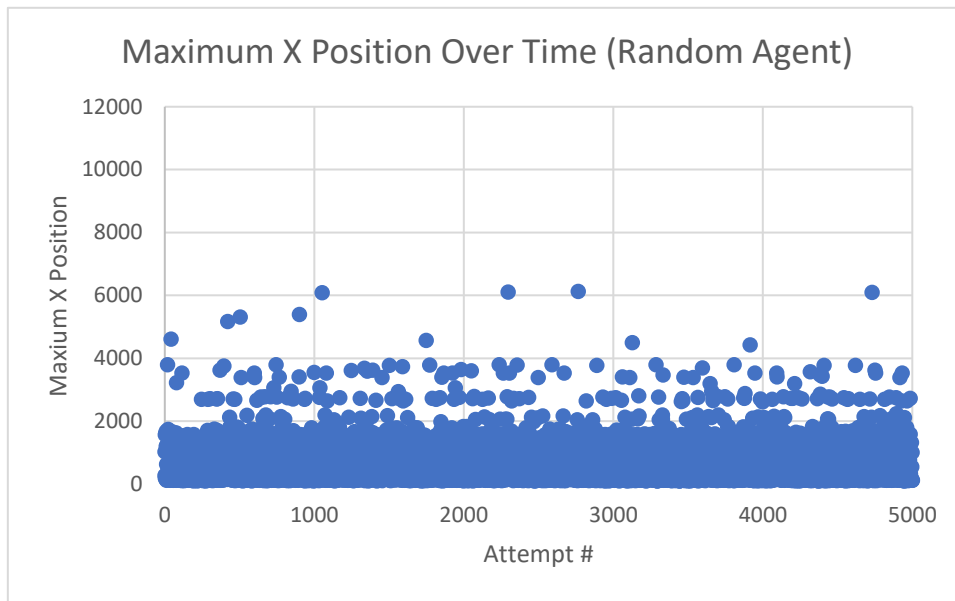
Experiments 8 and 9 however were made to test the generality of our solution. We would take the best model trained on act 1 and compare it to a new model that will train on act 2 with the same parameters and compare how the two perform.

	Movement selection	Reward	Done condition	Memory amount
8	Best Trained Model	???	???	???
9	Untrained Model	???	???	???

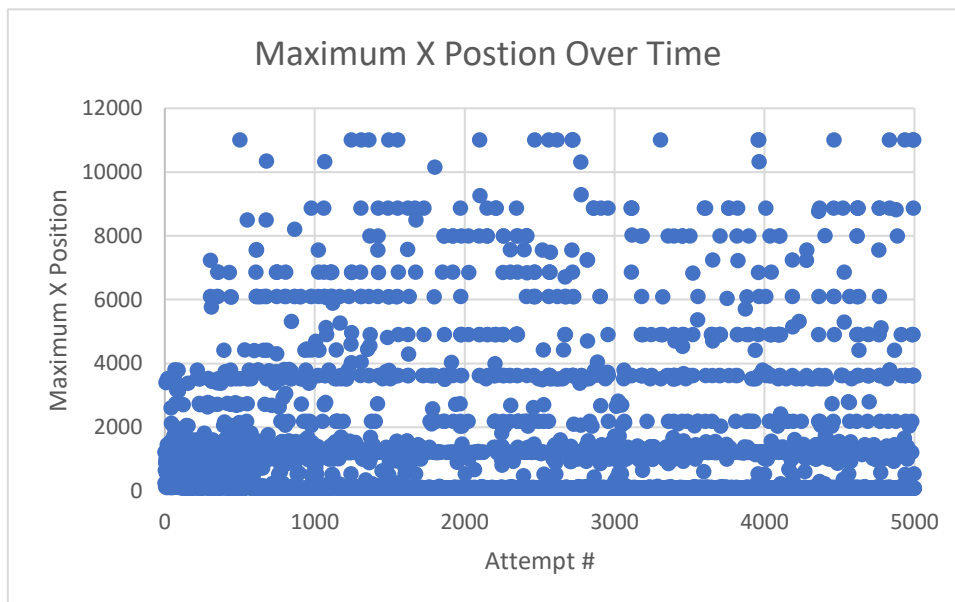
[Figure 5.2] Generality experiment plan

### 5.2.2. Better Than Random

To be sure that the agent can perform better than a random agent we must compare the model to a one. As mentioned Previously consistency was a big issue with the implementation, comparing correlation does not work for the implementation due to the inconsistency. Instead, we compare scatter plots of the maximum X position of the agent over time. We can also perform a T-test to see if the results are statistically significant.



[Figure 5.3] A graph showing the maximum x position over time for a random agent.



[Figure 5.4] A graph showing the maximum x position over time for our control agent.

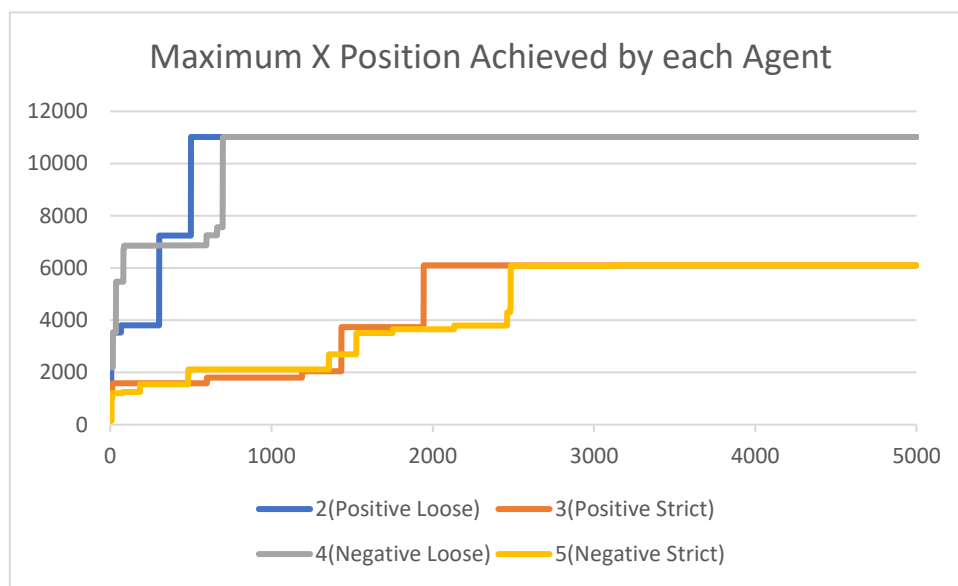
By comparing the two graphs, it is obvious to see that the control agent improves upon a random agent, by overcoming obstacles that a random agent cannot and as a result progressing further. The results of the T-test back up this finding. A one-tailed, Two-sample unequal variance T-test was performed and a p-value of 1.66925E-48 was obtained. This p-value is significant less than 0.05, suggesting that the difference in results is statistically significant.

An interesting observation from the scatter plot is the appearance of lines at obstacles that are hard to progress past. For example, the obstacle at around x position 6000 that the random agent cannot pass is the hill from figure 1.1. Because of the increased progress and the ability to beat the level, we can say that our control agent is better than random.



### 5.2.3. The Effects of the Scenario Files

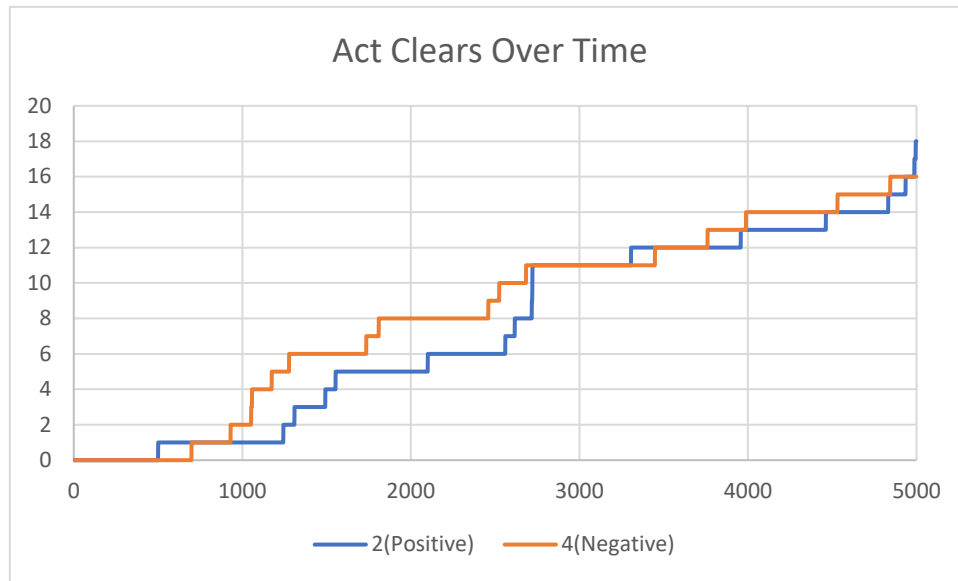
In addition to the control, experiment 2, there are three other Experiments that observe the difference in behaviour with different scenarios. There are two different *reward functions* and two different *done functions*. The positive reward function and Loose done functions are the ones described in section 4.1, whereas the negative reward function and the strict done function approximate my original implementation described in 5.1. The negative function is different in that it always gives a score based off the change in position instead of only when a new maximum is obtained, meaning negative reward can be obtained by going backwards. The strict done function functions the same as the loose done function except the amount of time between checkpoints and distance able to be travelled backwards have both been greatly reduced.



[Figure 5.5] A graph that compares the progression of the maximum X position achieved by the scenario file agents over time.

As can be seen in the above graph changing this part of the *reward function* has little impact on the performance of the agent. Both can overcome hard obstacles and complete the level. Whereas the *Done function* seems to have a dramatic effect on the ability of the agent. Much like the random agent, agents with a strict *done function* have trouble passing obstacles that require considerable speed to pass. This shows the importance of a good *done function*, allowing the agent reasonable space to explore the environment, should produce a better agent.

The two loose agents compare rather similarly to each other. They have a similar average distance; 1163 for positive and 1180 for negative. The positive agent was able to reach the end of the level slightly more than the negative agent. Obtaining 18 act clears compared to 16 for negative. Comparing the act clears over time in a graph reveals that once again they are very similar.

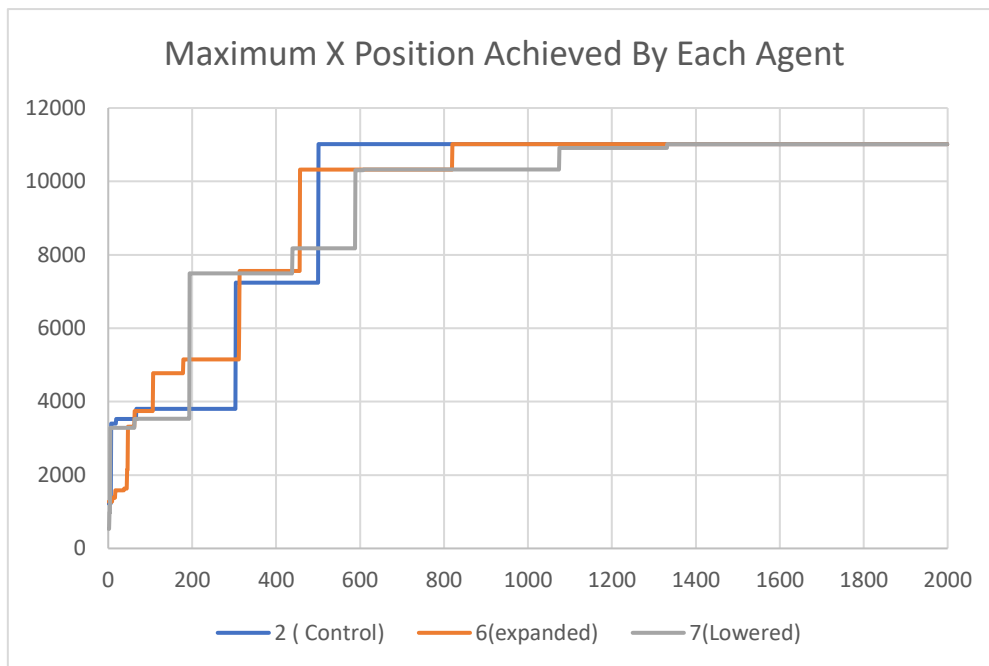


[Figure 5.6] A graph comparing act clears obtained by both agents with loose done functions.

This provides evidence that modifying the reward function in this way does very little for the performance of the agent. This could be for a variety of reasons. Perhaps it is the checkpoint system not the *X position* system that is the main driver of the reward function. Or maybe for the sake of move selection there is not much difference between negative and zero points.

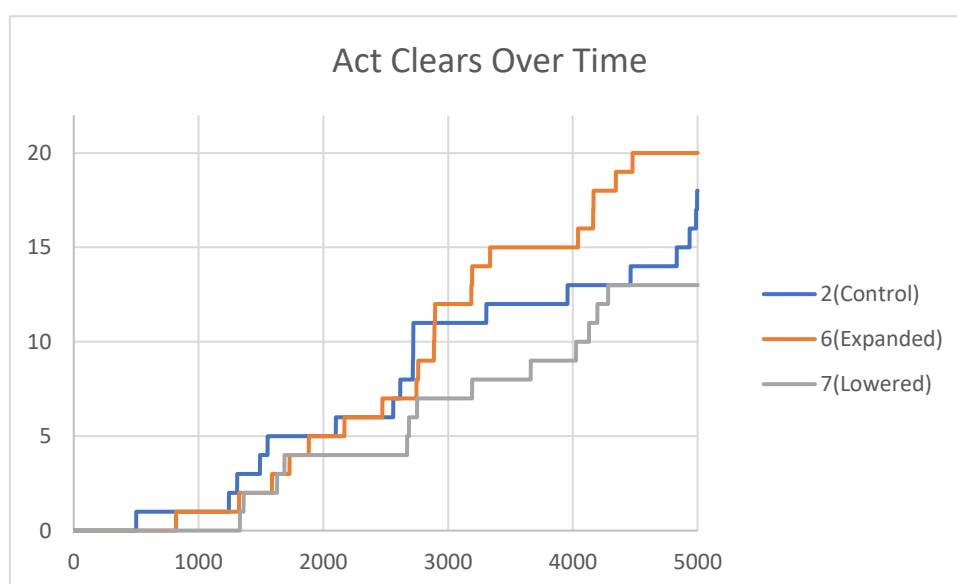
#### 5.2.4 The Effects of Memory Size

To compare the effects on memory size the project runs two experiments. One that converts the RGB image to a greyscale image, affording the system 3 times the amount of storage by making the input less costly in memory. And another that looks at the effects of having a third of the memory has on the system. Unlike the *Done function* changes the memory seems to have a more subtle and nuanced effect on the agent. Both agents in experiment 6 and 7 were able to successfully complete the act, their progress is compared in the graph bellow. Since all agents completed the level, the graph has been shortened to ease comparison.



[Figure 5.7] A graph comparing the maximum X position achieved by each memory experiment agents over time.

As the graph shows they are all rather close and interconnected but the control does beat the level before the other agents, however the other agents do reach milestones before the control implying that more information is needed to understand the data. Changing the memory amount may benefit certain aspects of the AI while hampering others. A good example of this is the average distance covered by the agents, the lowered memory agent has a significant improvement in average distance over the control, 1259 to 1163. Whereas the increased memory has an overall lowered average with 1085. These results are especially interesting as they do not correlate with how many act clears each agent gets.



[figure 5.8] A graph comparing act clears obtained by the control agent and both altered memory agents.

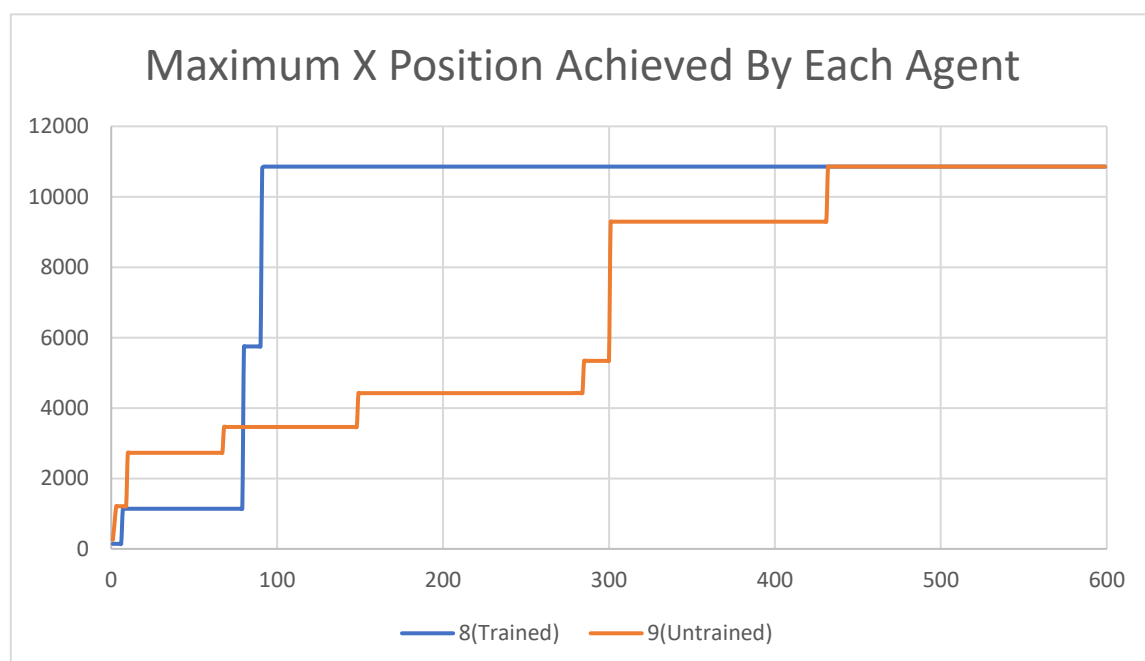
The Expanded memory has more act clears than the control, whereas the lowered memory agent falls behind. While the inconsistency could be interpreted as negative as its unclear what effect memory has on the system, another way to interpret the data, is that there are many ways we can change how the agent performs and each method has its upsides and downsides and future optimisations will have to balance these desirable features.

### 5.2.5 Generality

The final experiment is testing if the project's implementation of deep Q learning can display any generality. Google describes generality as:

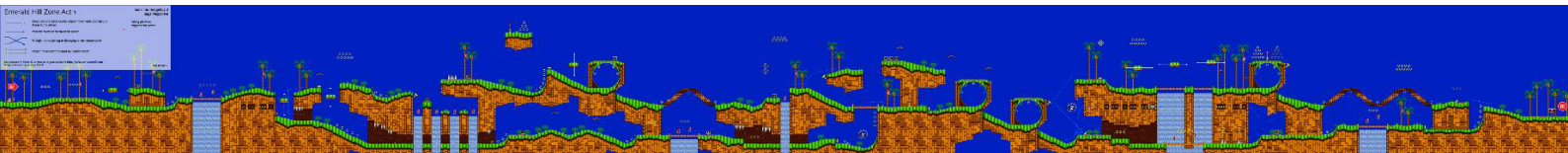
***"Generalization** refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model."* [16]

This description explains generality well for our use case. The agent trained in experiment 6 was taken and compared to a brand new agent with the exact same parameters on how it performs on act 2 of Emerald Hill Zone. The trained agent is at its minimum epsilon value so it will act greedily most of the time, whereas the untrained agent will have to learn the level from scratch. This first graph compares the maximum X position obtained by the trained agent and the untrained agent on act 2.

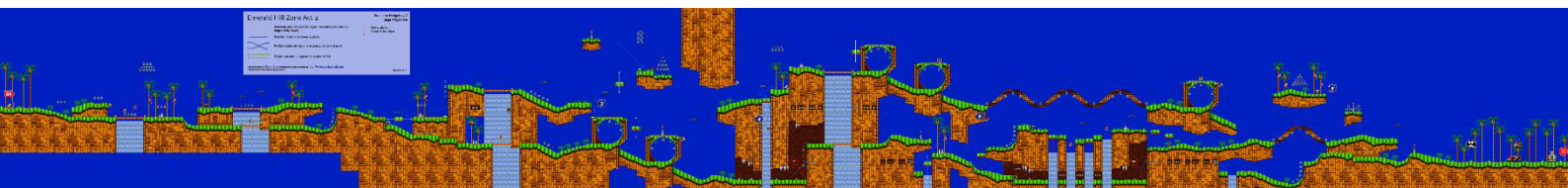


[Figure 5.9] A graph comparing the maximum X position achieved by the trained and the untrained agent in act 2.

This graph is visually distinct to the other graphs shown in this section. The untrained agent acts as expected, while the trained model spends some time at the begin exploring the solution, but almost immediately clears the level soon afterwards. This sounds like quite the success for generality for the project. And it is a success in that the model does demonstrate generality. However, on further inspection, there are some things that make this less significant. Below is a comparison of the maps of act 1 and act 2 of Emerald Hill Zone.



[Figure 5.10] A map of Emerald Hill Zone act 1[17].

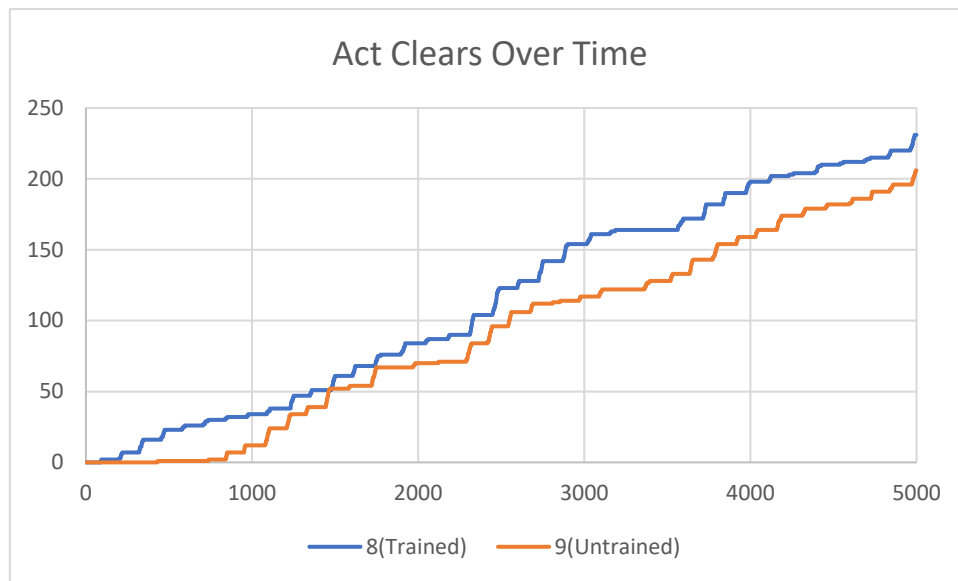


[Figure 5.11] A map of Emerald Hill Zone act 2[17].

At first the levels look very similar, but there is an important difference that makes act 2 a lot easier for a model to clear the level. The difference here is how slopes are used in the level. Slopes are much less of a problem for the agent in act 2 than act 1. In act 1 the only way to avoid the two incredibly steep slopes is by keeping to the upper path. This is a hard task for our agent, our agent instead tends to stick to the lower easier path. You may notice that act 2 also has two steep slopes that an agent going along the bottom of the level would need to pass, this is a valid concern, lets discuss what makes these slopes different. Both slopes in act 1 come after an enemy and a jump. These cause complications for our model, slowing Sonic down, making them less likely to successfully pass the slope. Both slopes in act 2 however are easy to overcome for different reasons. The first slope has uninterrupted flat land in front of it, easily allowing the agent to reach a high enough speed to pass. The second is a slightly odd case. By the time an agent is getting to the end of the level, it should have learned that to progress it needs to move to the right. Notice that taking the easiest path through act 2 puts Sonic above the second hill near the corkscrew object. If the model is holding right to progress as it has likely learned to do by this point it will enter the corkscrew object and completely bypass the second steep slope.

One may think that while the second act is certainly easier than act 1 it should not have too much of an effect on the performance of the agents. This discussion of how level design affects how successful the agent is important, less as tool to demonstrate how the project demonstrates generality but more on how seemingly subtle things can affect model success. Both agents were able to complete act 2<sup>11</sup> over 200 times. An order of magnitude greater than the best agent for act 1.

<sup>11</sup> Act 2 contains a boss at the end that is not considered for this implementation. The environment is reset at the boss area before the boss fight occurs.



[Figure 5.12] A graph comparing act clears obtained by Trained and Untrained agents on act 2 of Emerald Hill Zone.

Looking at the graph above we can see that the agent trained on act 1 and the agent trained on act 2 are about as good as each other in the ability to beat the level. The trained agent just begins with the advantage of acting greedily earlier, and it keeps a consistent advantage over the untrained agent. Both agents have similar average distances<sup>12</sup>, further signifying that the implementations are equal in quality. From the information gathered in this section we can gather that while the implementation certainly displays generality there are factors that are making this an easier outcome than would be expected.

## 5.3 Other Findings and Evaluation

Before discussing the evaluation of the project, Appendix A3 and A4 should be mentioned. They contain the best clear times for act 1 and act 2 respectively over all models trained. Discussing Speed for a moment, the project's best act 1 attempt is faster than the reinforcement learning example provided by Kaplan, A in [2] but narrowly loses to the NEAT implementation by Thompson, L in [1]. I believe it is fair to say that the project performs comparatively well to other machine learning implementations in regard to speed.

Where the project's implementation does not quite reach a desirable standard is in its consistency. There is a fundamental flaw somewhere in the implementation that was not found, as such the results obtained are wildly inconsistent, with the agent spending dozens of attempts doing nothing. This flaw does weaken a lot of the achievements made by this project.

Because despite the consistency issues the project has reached each of its goals. Several working agents were produced that were able to clear the level where a random agent

<sup>12</sup> 1158 for trained and 1125 for untrained, the slight gap again coming from an earlier solution.

could not. The agents were able to match the speeds of other Machine Learning implementations for Sonic the Hedgehog 2. And finally, the agent displays generality, being able to beat levels it had not previously seen with relative ease. These accomplishments make the inconsistency issues feel worse than they otherwise would as there is clearly a good solution under this overwhelming problem.

## 6. Future Work

### 6.1. Improvements and Further Experimentation

Any future work on this project should first seek to solve the primary issue of inconsistency. While it is clear from the generality tests that the issue is not exclusively on the agents, it should be primary focus of any work following on from this project. There are many things that could be responsible for this issue. From tensorflow 1 compatibility issues, issue with the optimizer, or perhaps deeper issues with the implementation. It would be productive to consider alternative deep Q learning approaches like double or duelling deep q networks[18].

It would conducive to run more experiments in a similar manner to the experiments demonstrated in section 5.2. There is a large variety of systems in this project that have the capability to be tested such as the model optimizer, or Q Learning training parameters for the Bellman equation. These parameters play huge role in how the project works and should be considered with more scrutiny in future work.

To summarise here is what should be considered for anything that follows on from the project:

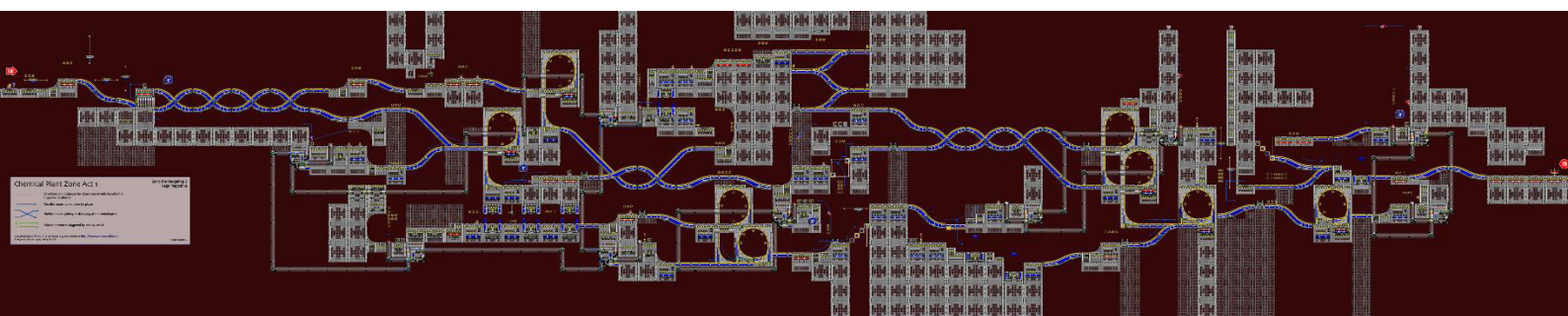
- **Fix Overlying Flaw in Implementation.**
  - Q learning parameters may reveal a major area of improvement.
  - Consider rethinking the current transition memory in the form a full replay Buffer.
  - Tensorflow 2 upgrade may improve performance.
  - Other more advanced methods or upgrades of Deep Q learning could be implemented.
- **Experiment with new Data Possibilities.**
  - Allowing an agent more attempts to see if there is more convergence as time goes on?
  - Implementing time in the reward function, perhaps encouraging faster play?
  - Minimalizing the rewards and done functions, simplification may yield interesting effects?
  - Creating a variable Done Function, where the agent gets more or less time and space around difficult obstacles depending on performance?

By implementing these changes, more information will be discovered about the project, and likely even more questions and optimisations will be found.

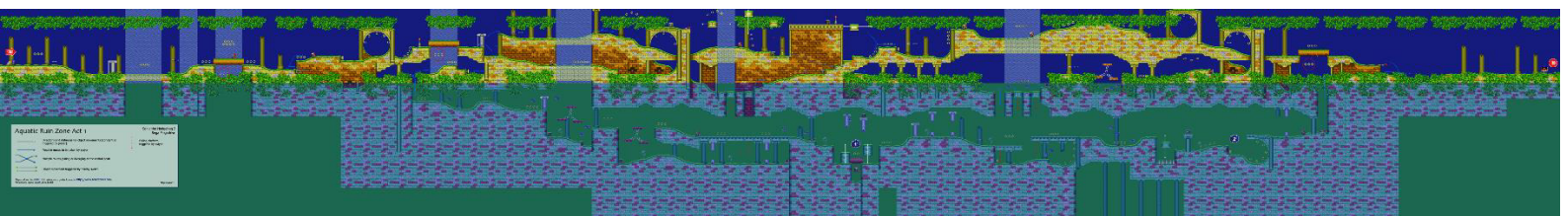
### 6.2. Implementing for Other Levels

The most obvious place for a machine learning implementation for video games to go after successfully completing the first level of the game is to other levels of the game. Other levels of the game range from hard but possible, to completely unfathomable to beat using the current implementation. As a case study the first acts of the second and third levels in the game will be compared.





[Figure 6.1] A map of act 1 of the second level of the game, Chemical Plant Zone[17].



[Figure 6.2] A map of act 1 of the third level of the game, Aquatic Ruin Zone[17].

Comparing the two maps it can be seen that both are bigger and more complicated than Emerald Hill Zone. Chemical Plant Zone has numerous winding pathways that interconnect. Whereas aquatic ruin has two to three main pathways but has increased difficulty in managing the water mechanic in the level. All other levels in the game follow these two templates to some extent. Multiple interconnected winding paths or long challenging straightaways.

While Aquatic Ruin Zone is certainly a harder level for a human player than Chemical Plant Zone, Aquatic ruin Zone contains a lot less obstacles that may be confusing to the model, which would likely made it easier for the model to learn overall. In order for the agent to even begin playing Chemical Plant Zone the *agents done function* would need to be completely overhauled. Unlike Emerald Hill Zone there are significant portions of Chemical Plant Zone and later levels that are styled similarly that require the agent to go leftwards. In act 1 for example there are several pipes that move you about the level, often necessary to progress in the level that move sonic much further back in the level.

This is where the difficulty of designing this system becomes obvious. Levels like Aquatic Ruin Zone are easy to create reward and failure states for, as the primary goal is consistently “move right towards the goal”. Whereas many closed levels like Chemical Plant Zone become harder to solve. When should the environment be reset? How to encourage the agent to make backwards progress to make future forward progress?

A successful implementation for later levels would have to make those considerations. Making the done function a lot weaker or allowing it to vary through the course of training. For rewarding the agent, three options are proposed. Reward traveling along the Y axis in some way, going backwards typically comes with a change in altitude. Another option would be to design a system that explores and remembers the level, and rewards the agent, not for forward progress but for achieving a new location on the

map, perhaps with additional reward for consistent change to a new position. The final system will be explored in more detail in next subsection, but in short rewarding the agent for taking actions that allow for future reward would improve the implementation considerable.

## 6.3. System for Predicting Future Reward

What does predicting future reward mean for Reinforcement Learning? In this case its not predicting and more about connecting two learned pieces of information.

Sometimes certain actions<sup>13</sup> yield no reward on their own but are necessary for making progress later and gaining a large reward. Coding an intrinsic system for this would be difficult as it would require a large amount of memory to store which actions lead to which states. This mimics the Q-table of a standard Q-learning approach, which we would like to avoid in our implementation.

A possible implementation of this idea would instead apply a percentage of, or the current reward to a few of the previous transitions in the transition memory. This could encourage certain action that would not typically gain reward to taken more often. This system paired with the other changes mentioned in this section should combined create a more robust, well realised agent, that takes more optimised paths through the level and better selects the appropriate move.

---

<sup>13</sup> Moving backwards is the classic example here.

## 7. Conclusions

The project has obtained mixed results. There is a lot of promise in the methods used and results obtained. The project meets each of the aims it set out to accomplish but is ultimately ruined by a few implementation problems that leave the project's agent in a mess of inconsistency.

The report suggest that deep Q learning can be implemented successfully for Sonic the Hedgehog 2. The project has demonstrated agents that are capable of beating the level and acknowledges the roadblocks in the way of beating the level and discusses how the agent may overcome these roadblocks. We can gather from comparison to other machine learning implementations and by comparing the first act completions to later act completions that the agent has improved to a decent speed for a machine learning implementation. And finally, despite the realisation that act 2 is significantly easier than act 1, the implementation was able to demonstrate generality by overcoming unseen obstacles that otherwise take significantly longer to overcome successfully.

Ultimately the success of the project is heavily muddled by the failure, but overall the project can still be considered a success.

## 8. Reflection on Learning

I think if I were to do this project again, I would focus a lot more on the implementation of the actual neural network. Problems with it were the main downfall of the project. Like I mentioned at the beginning of the report, prior to the project I had incredibly little experience with machine learning outside of a passing knowledge of classification systems. Spending that extra time to really develop a deep understanding of the underlying mechanics of the neural network would really help the project. Being the biggest weakness of the project.

Completing this project has taught me a lot about Reinforcement learning. It is an incredibly diverse field that is full of potential. It has a lot of differences from other machine learning paradigms. These differences can easily be interpreted as problems, but should be considered as opportunities, these models can have a lot of problems but show great potential to improve and provides you with the avenues to make these improvements.

Two technologies that I was not expecting to learn so much about when I started this project was Lua and networking. Lua was used in every approach to the project. In hindsight this is because of its ease of use and implementation, but mostly because of Lua's speed in comparison to similar languages like Python. I picked up a surprising amount a networking knowledge over the course of this project, increasing my understanding of network sockets, and briefly dabbling with SSH tunnels.

Completing this project has taught me how to attempt and consider multiple options when implementing code. It has taught me to see multiple ways of processing and consider and weight all these options. It has taught me the values and the dangers of thinking outside of the box. This project has allowed me to personally develop and a practitioner of software development by allowing me to experiment, fail, and learn from these experiences and I've developed considerably in this regard because of it.

## 9. Bibliography

- [1] Thompson, L., 2020. *Sonic-Bot-In-OpenAI-and-NEAT*. [online] GitLab. Available at: <<https://gitlab.com/lucasrthompson/Sonic-Bot-In-OpenAI-and-NEAT>> [Accessed 14 May 2021].
- [2] Kaplan, A., 2018. *rl-sonic*. [online] GitHub. Available at: <<https://github.com/KaplanAlex/rl-sonic>> [Accessed 14 May 2021].
- [3] Sutton, R. and Barto, A., 2018. Reinforcement Learning: An Introduction. 2nd ed. pp.1-2.
- [4] Maroti, A., 2020. *Reward based  $\epsilon$  decay*. [online] Reward Based Epsilon Decay. Available at: <<https://aakash94.github.io/Reward-Based-Epsilon-Decay/>> [Accessed 11 May 2021].
- [5] Shyalika, C., 2019. *A Beginners Guide to Q-Learning*. [online] Medium. Available at: <<https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>> [Accessed 12 May 2021].
- [6] ADL, 2018. *An introduction to Q-Learning: reinforcement learning*. [online] freeCodeCamp.org. Available at: <<https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/>> [Accessed 12 May 2021].
- [7] Wang, M., 2020. *Deep Q-Learning Tutorial: minDQN*. [online] Available at: <<https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc#:~:text=Critically%2C%20Deep%20Q%2DLearning%20replaces,process%20uses%20%20neural%20networks.>> [Accessed 12 May 2021].
- [8] Gym.openai.com. 2016. Gym: A toolkit for developing and comparing reinforcement learning algorithms. [online] Available at: <<https://gym.openai.com/>> [Accessed 12 May 2021].
- [9] OpenAI. 2018. Gym Retro. [online] Available at: <<https://openai.com/blog/gym-retro/>> [Accessed 13 May 2021].
- [10] TensorFlow. n.d. TensorFlow. [online] Available at: <<https://www.tensorflow.org/>> [Accessed 14 May 2021].
- [11] Colab.research.Google.com. n.d. Google Colaboratory. [online] Available at: <<https://Colab.research.Google.com/notebooks/intro.ipynb>> [Accessed 14 May 2021].
- [12] Tasvideos.org. 2012. TASVideos / Bizhawk. [online] Available at: <<http://tasvideos.org/BizHawk.html>> [Accessed 18 May 2021].
- [13] Console.cloud.Google.com. n.d. Google Cloud Platform. [online] Available at: <<https://console.cloud.Google.com/>> [Accessed 19 May 2021].

[14] Info.sonicretro.org. 2020. SCHG:Sonic the Hedgehog 2 (16-bit)/RAM Editing - Sonic Retro. [online] Available at:  
<[https://info.sonicretro.org/SCHG:Sonic\\_the\\_Hedgehog\\_2\\_\(16-bit\)/RAM\\_Editing](https://info.sonicretro.org/SCHG:Sonic_the_Hedgehog_2_(16-bit)/RAM_Editing)>  
[Accessed 20 May 2021].

[15] Tabor, P., 2019. Reinforcement Learning Course - Full Machine Learning Tutorial. [online] Youtube.com. Available at:  
<[https://www.youtube.com/watch?v=ELE2\\_Mftqoc](https://www.youtube.com/watch?v=ELE2_Mftqoc)> [Accessed 22 May 2021].

[16] Google Developers. 2020. Generalization | Machine Learning Crash Course | Google Developers. [online] Available at:  
<<https://developers.Google.com/machine-learning/crash-course/generalization/video-lecture>> [Accessed 23 May 2021].

[17] Soniczone0.com. 2010. Emerald Hill Zone. [online] Available at:  
<<http://www.soniczone0.com/games/sonic2/emeraldhill/>> [Accessed 23 May 2021].

[18] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M. and de Freitas, N., 2016. Dueling Network Architectures for Deep Reinforcement Learning. [online] Arxiv.org. Available at: <<https://arxiv.org/pdf/1511.06581.pdf>> [Accessed 24 May 2021].

## 10. Appendix

### Appendix A – Video Examples

- 1 - <https://www.youtube.com/watch?v=1AeOBB0dj5E>
- 2 - <https://www.youtube.com/watch?v=cZq9L7WVyjo>
- 3 - <https://youtu.be/8JoF4GSDh8o>
- 4 - <https://youtu.be/bGvKi-0lhZU>