CIS 510: Q-Learning Classic Videogames

Steven Walton

June 6, 2019

1 Project Description

In this project we create a Q-Learning algorithm that is able to play classic, or "retro", videogames. These games are widely considered to be NP-Hard or many times PSPACE-Hard [2]. PSPACE-Hard problems are not only hard to compute, but also hard to verify. This should match intuition of many videogames as it is almost always unclear what an optimal strategy is. This is largely due to the number of states in a single game. While actions may be few in number, generally limited to the buttons one can press, each frame in the game can generally be considered a new state. This makes even calculating the number of states extremely difficult.

At the beginning of this project we set out with the goal to create a Q-learner that would be able to play Super Mario Kart (SMK) [4] through use of OpenAI's Retro environment [5]. Retro creates an environment that abstracts out emulation and helps developers focus on creating algorithms, specifying actions, and rewards. Stretch goals were to get a deep Q network (DQN) working and have several agents play against one another.

While we were not able to reach the goal of getting a reinforcement algorithm playing SMK, we were able to demonstrate feasibility in retro's environment on several other games, which had already be correctly integrated.

2 Background

In this section we will discuss the various tools and break down how we will formulate the project. In the next section we will discuss our proposition of how to achieve these goals.

2.1 Q-Learning

For this project we will be using a Q-Learning algorithm to play these videogames. The Q-Learning algorithm is a simple reinforcement learner that follows Bellman's Equation. A Q matrix can be defined where each row represents a different state and each column represents an action for that state. These can be used to solve different Markov Decision Processes (MDPs). An example MDP is shown in Figure 1.

Each node, labeled s_i, represents a state and the connections between nodes represent actions. Each action has an associated probability with it. There are certain "rewards" that an actor gets for taking certain actions, a, which leaves them in a new state, s'. We then want to find a set of actions, also known as a policy (π) that gives us our maximal value (V). Knowing this we want to create a learner that will learn the best policy, set of actions, that will maximize its reward. The

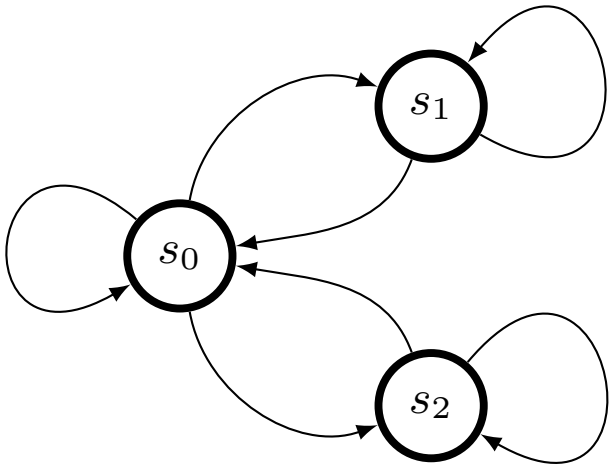

Figure 1: Sample Markov Decision Process

Bellman Equation, Equation 1, does just this. In this equation α represents the learning rate, γ is the discount rate, and $Q(s', a')$ is the value in our Q matrix given the next state and action.

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{1}$$

Essentially what we are doing is looking at the reward we get for taking an action and all future rewards given our best policy. A discount rate is applied so that rewards that are achieved sooner are weighted more heavily than rewards further away. A simple example of why we might want to do this is because if we give a mouse a reward for completing a maze we want the mouse to finish the maze as fast as possible and not take an infinite time to finish. The learning rate applies a weight to to our new reward and future rewards. Additionally all actions are probabilistic. If we have the actions forward, left, and right, if we select forward there is still a probability that the agent goes left or right. This is commonly known as a Monte-Carlo walk (or Drunken Walk).
For our videogames we discussed that there are potentially an unknown number of states. There are also limitations in computational hardware. To account for this we can simply limit the depth of our lookahead values. This will create an approximate Q-Learner and allow us to solve problems with unknown or infinite number of states.

2.2 Deep Q-Learning

An extension of Q-Learning is deep Q-Learning (DQN) is by using a neural network to compute the above solution. We can accomplish this because a neural network is a universal function approximater, meaning it can approximate any function. Following DeepMind's work [3], if we rewrite the Bellman Equation as

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \right] \tag{2}$$

We can generate a loss function $L_i(\theta_i)$

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3}$$

where $y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})\right]$. Recognizing that this is the same format as the Mean Square Error (MSE) we can generate the following gradient (Equation 4).

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{4}$$

Here we have everything we need to create a DQN. The property of universal approximation handles the lookahead depth for us. This is because a neural network iterates to create an approximate function. Being that our games are NP- or PSPACE-Hard this is reasonable.

2.3 Integration into Retro

Unfortunately SMK was not able to be integrated into Retro's environment, but much about the environment and integration was learned along the way. Retro relies on different "states", which can be defined as different starting states. This is placed in quotes because the state will continuously be changing as the agent progresses through the level, more on this later. For this project we started with the scenario where there is a single agent playing the 50cc Mushroom Cup. Essentially we set a place in the game where we want our agent to start. Retro provides a UI where we can interactively play the game and save specific states. For this example our initial state looks like this:

Figure 2: Initial state for 1P_GrandPrix_50cc_MushroomCup.state

From this we can see that in the upper half we see our player and the world around us. In the bottom half we see the entire map and can actually identify certain features like coins and items. We will be focusing on the top half.

Now that we have where we want to start, we need to define everything else. Retro uses json files to define many different parts to the game. There are two important files for each game: data.json and scenario.json. Data defines information about the level, lives, and score. Scenario defines the ending condition and the reward structure. An example of values are shown below

The data.json file contains hex addresses that represent the RAM address that the game uses

Listing 1: data.json

```
{
"info": {
    "lap": {
        "address": 8261825,
        "type": "|u1"
    },
    "lapsize": {
        "address": 8257864,
        "type": "|u1"
    },
    "coins": {
        "address": 8261120,
        "type": "|u1"
    },
    "minute": {
        "address": 8257796,
        "type": "|u1"
    },
    "second": {
        "address": 8257794,
        "type": |u1"
```

Listing 2: scenario.json

```
{
"done": {
    "variables": {
        "finish": {
            "reference": "coins",
            "op": "eq"
        }
    }
},
"reward": {
    "variables": {
        "rank": {
            "reward": 1.0
        }
    }
},
"actions": [
    [[], ["UP"], ["DOWN"]],
    [[], ["LEFT"], ["RIGHT"]],
    [[], ["A"], ["B"], ...
}
```

Figure 3: Snippets from data.json and scenario.json for SNES Super Mario Kart

to determine certain aspects of the game. We also need to add these to a rambase value that is given by retro (warning: this is not noted within the documentation but is required). Additionally, scenario.json can contain information about the action set. We can either restrict our action set to a specific set of buttons, and button combinations, or we can expose all possible set of actions to the agent. Initially we will be limiting our agent, since certain actions like ["DOWN","UP"] are not likely to be useful, excluding glitches. Super Mario Kart has the following set of actions: Steering with the D-Pad, B for Accelerate, A to use an item, Y for Brakes, X for the rear-view, L or R for Hop/Power-side (same action), Start to pause the game, and select for Rear-View. We will limit the agent so that it may not look behind itself, cannot pause the game, and we will select R for hop/power-slide (because that is what I use when playing). So our new full action set for the steering will be {{UP},{DOWN},{LEFT},{RIGHT}, {UP,LEFT}, {UP, RIGHT}, {DOWN,RIGHT}, {DOWN,LEFT}}. This represents the left side of the controller. For the right side of the controller we will assume that the agent can play like a human does (this will also limit glitches that the agent might be able to exploit). The controls action set will be defined as {{B},{A},{Y}, {R},{A,B},{B,R},{A,R}}. The full action set is the combination of actions from the steering set

and the control set.
RAM addresses can be found by using emulator tools, such as BizHawk [1], that allow for hex searching and manipulation. Some values are easy to find since their values in RAM identically match the values displayed on the screen. Other values may be more difficult to find given that they may be manipulated by the program. An example of finding a memory address in BizHawk is shown in Figure 8 in the Appendix. We were unable to find all the required address on our own, but a prominant Speed-Runner[1] named SethBling [6] kindly provided me with Lua scripts to help me find the missing addresses. Section 3.1.2 provides a more detailed discussion on finding RAM values.
To summarize the integration: we need to fully define how our agent will receive rewards, when the agent ends, and our action set.

2.4 Rewards

Defining rewards is always a challenging problem in any machine learning task. There are many fantastic examples where a well thought out reward function leads to outcomes not expected by the machine learning engineer. Rewards can be simple or extremely complex. For example a simple reward function for Super Mario Bros might be to use the x position on the screen and reward the agent for making forward progression through the level.
A more detailed discussion of rewards for Super Mario Kart are given in the appendix 6.1. Additionally sections are provided for determining the end state 6.2 and visualizing the environment 6.3 specifically in the SMK world.

3 Implementation

While we learned a lot from this project, we did not achieve the desired outcome. Along the way we learned much about the many different systems involved and have created a path forward to continue the goal in our own time.

3.1 Integration of Super Mario Kart

Most of the project was spent trying to accomplish just this task. While it was not achieved, we have learned much about the systems involved. Much more time will be required to complete this, but much of the ground work has been laid down. In this subsection we will discuss different parts of the integration and what went wrong.

3.1.1 Retro

While retro was created by a large company, it is not as well documented as OpenAI's gym. Specifically it is lacking much information in the integration of new games. For example, when the Official Guide discusses integrating a new game, it does not mention that addresses need to be added the rambase. This was found by a lot of searching on the internet and then verifying with BizHawk and addresses from other games. This may be because not every system has a rambase value, but is specifically required in Super Nintendo games.
Because of issues like these there was a lot of time spent reading the source code and learning how the functions worked that way. Retro also imports from gym, so source and documentation were

[1] People who compete to finish a game as quickly as possible

read for both modules. While we were able to determine all the desired RAM values, we were not able to get the game running.

3.1.2 Finding RAM values

While the steps illustrated in an earlier section seem easy to implement, there is a lot of work that must be done to determine the values that were used. As mentioned previously, some values were easy to find and others aren't. Using a tool like BizHawk this can be a tedious task. BizHawk looks at different game states and returns the addresses and values associated with them. Since there are hundreds of thousands of addresses, one needs to be clever by searching as many states and executing as many actions as possible until there are a reasonable number of addresses left. One can then verify that an address contains the desired value by freezing or modifying it. This was easy for an address such as the clock, because the miliseconds were always increasing and the seconds and minutes were located near one another in memory, but was overwhelming for addresses like rank and lap number. I would not have found these values without the help of SethBling, as these values do not line up with the values that appear on the screen. Additionally there is a delay in when the values are updated within the RAM and when the values are displayed on the screen. A list of these values and equations can be found in the Appendix 6.4

3.2 Trying without Retro

When we had emailed SethBling he mentioned that he was working on a similar project and offered me his code as a starting point. The intention was to be able to use his setup to handle the environment and then rewrite the learner with basic q-learning and then upgrade to a deep Q network using pytorch. While his scripts helped me find the missing RAM values I could not get his code running on my, or several other, machines. Going through his code helped me understand some of the problems I was having in retro. Due to time constraints we decided to head back to work with retro and concentrate on the main goal, creating a q-learner.

3.3 Back to Retro

Fortunately retro comes with a ROM for testing. Retro's environment should allow for any game to be played with the same "play" code. It was decided that the best option to move forward would be to integrate into this environment testing on the provided ROM and then if there was time left to return back to the integration process. The first part has been successful, while there was no time left to finish the integration process.

3.3.1 A Basic Q-Learner

Retro provides both a random player and a greedy solver as baselines and examples. These were used as the basis for learning how the system operated, and they are included in the main file as playable options. These files allowed us to explore the values and system. Once we knew how the bindings worked we were able to implement a simple Q-Learner based on Equation 1.
A Q-Learner is dependent upon the Bellman Equation. To create this learner we use something called a Q-Matrix. A Q-Matrix is composed of actions and states as indices and the value of those actions in that state as entries. Using this we can iteratively find the best policy to accomplish a task.

While iterating over a large solution space there may be many future actions that have the same value. In this case we could simply just pick a random one, but that might not be optimal. We can do better by creating a bias towards trying actions that we haven't tried before, given the same future reward. We can create the following exploration function, defined in Equation 5.

$$f(u,n) = u + \frac{k}{n} \tag{5}$$

u : utility from action

n : number of times action has been taken

k : a constant

In this equation, if $k = 0$ then we do not care about how many times an action has been taken. However if we use a k value then we can slightly offset our returned rewards to encourage more or less exploration, depending on the value of k.

4 Experiments

Since much of the project was spent on implementation there was little time left for experimentation and testing. We were able to test a few different options and test multiple games. There were many more runs than are included in here but not all were recorded as we were testing that different parameters worked and were were focusing on creating good videos for the presentation.
It is key to note that the default values of hyper parameters are: discount=0.8, frame skips=4, learning rate=0.8, depth=1, explore=0, random action=0.8. The discount is γ in Equation 1, similarly learning rate is α in the same equation. Frame skips are the number of frames that the game skips when emulating, this helps reduce the number of states and can help learning if adjacent states are extremely similar. Depth is the number of lookaheads that we perform (amount of recursion in Bellman's Equation). Explore is the k value in the exploration function, Equation 5. Figure 4 shows the results for Airstriker-Genesis, the default game for Retro. To perform this experiment a timer feature was added to the program to denote the time when the new best reward was obtained. These programs were run for the same amount of time. While this graph shows the brute forcing (greedy) algorithm greatly out performing our Q-Learner this was not always observed in practice. With limited time it is difficult to make accurate conclusions and properly tune hyper-parameters. Experimentation found that performance on specific games were extremely dependent upon the hyper-parameters. The Q-Learner has a tendency to get trapped in local optima and unfortunately without good parameters the learner does not perform as well. Seeing how many of the parameters do not extend along the x-axis it appears that these learners got trapped in local optima. Observing this we can see how important the exploration function is to help the learner escape these.

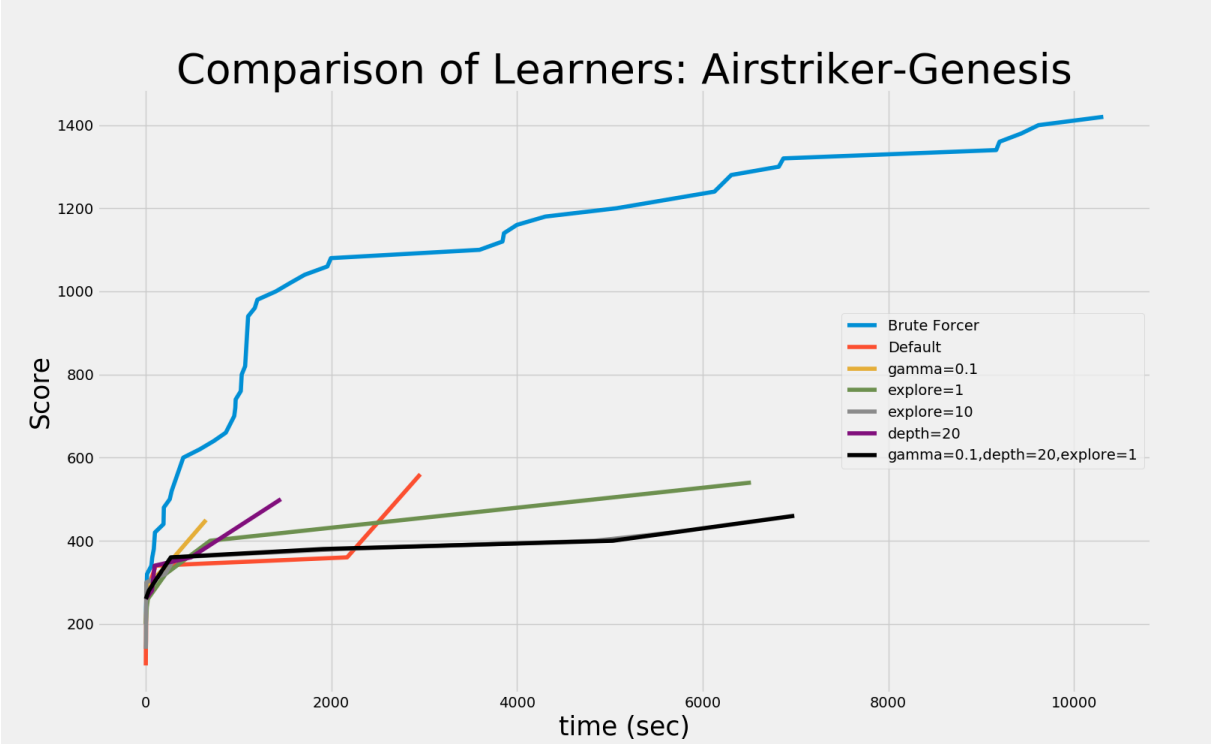

Figure 4: Comparison of different hyper-parameters on Airstriker-Genesis

The next thing to look at is the harder game, Super Mario Bros (SMB). Figure 5 shows different hyper-parameters configured to play SMB. All of these were again run for the same amount of time. I changed the graph slightly because everything got stuck in a local optima except for the parameter with $\gamma = 0.1$ depth= 50 and exploration= 1 (depth= 100 likely would have updated again). These values would likely have continued updating given more time but given the constraints I repeated their final scores with a time of 5000 seconds so that it was easier to visually see what happened. There was no manipulation of data as these were the values at 5000 seconds. Here we can see that every Q-Learner performed much better than the brute forcer algorithm.

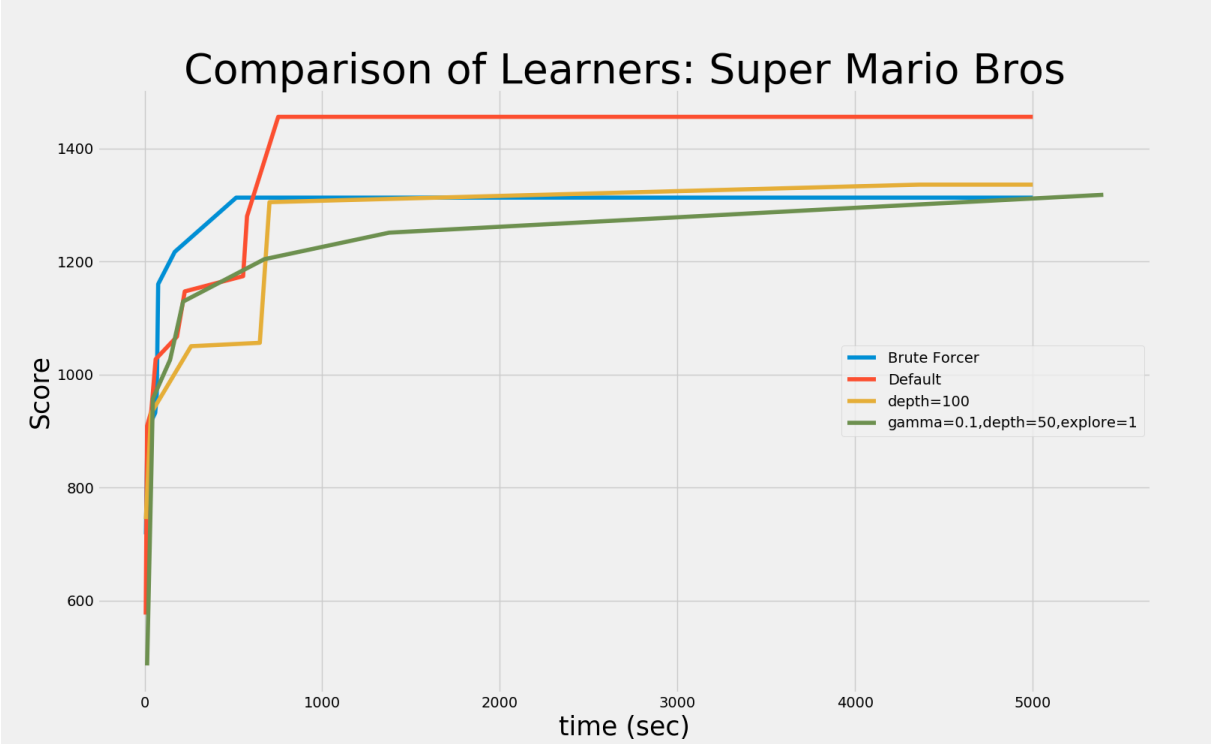

Figure 5: Comparison of different hyper-parameters on Super Mario Bros

We found that when playing a PSPACE-Hard game like DonkeyKong Country that our Q-Learner more easily beat the brute forcing algorithm. This can be seen in Figure 6. DonkeyKong Country is specifically a difficult game, as there are substantially more states and actions than Airstriker-Genesis. There are also two characters that can be played and swapped out, that each have different attributes. One of the sections that the learner consistently got stuck on was when it was using the worse agent at a particular section. The learner that ran with a depth of 100 (depth of lookahead) was not run as long as the other brute and default. There were constraints of how many learners could be run at a single time on our system. Additionally depth 100 explore 1 was run at a different time and showed more steady growth.

From these experiments we can make some conclusions about how Q-Learning works in different games. It seems that when the game is sufficiently complex that it performs much better than a greedy algorithm. It seems that we may have just lucked out with the default values. Additionally I had run some examples over the weekend but did not record the times associated with each step, so they are not included in this analysis. In those examples I had found that a high depth and exploration greatly helped the learners escape local optima. Similarly there were experiments run with frame skipping. It was found that games like DKC and SMB performed better with a slightly higher frame skip, moreso DKC than SMB. We hypothesise that this is because in these games there is more similarity between frames, or states. Specifically DKC performed better with this because it was not reaching enemies, which is where reaction time matters more. It is also believed that given more resources and time that these algorithms would actually be able to do significantly better. For comparison similar experiments by OpenAI, DeepMind, and others use hundreds of GPUs, whereas currently these algorithms are single-threaded and run on CPUs. Those systems also ran for hundreds or thousands of compute hours, where our maximal value here is just under 3 hours. There is clearly a lot of optimization that can be done that would greatly help improve these programs.

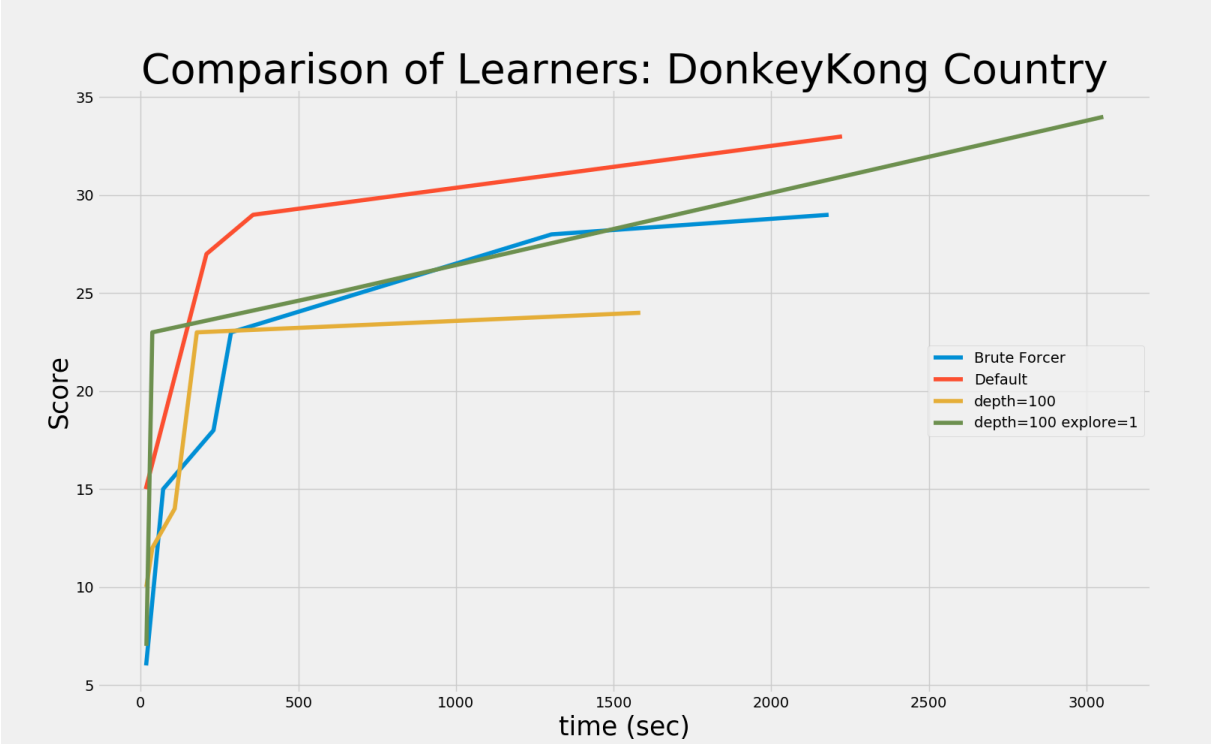

Figure 6: Comparison of different hyper-parameters on DonkeyKong Country

5 Conclusion

While our Q-Learner did not consistently perform better than the brute forcing algorithm on every tested game, there were some clear areas that the learner beat the brute forcer. When the state space was substantially larger the approximating Q-Learner was able to learn faster by using lookaheads and exploring its environment more than the greedy algorithm could. From testing we were able to conclude that dept and exploration had significant impacts on the learning rate of the algorithm. While there was a lack of resources and optimization in the code, these experiments show great promise in Q-Learning being able to solve extremely complex problems and warrants further studying.

References

[1] Bizhawk.

[2] Greg Aloupis, Erik D. Demaine, and Alan Guo. Classic nintendo games are (np-)hard. *CoRR*, abs/1203.1895, 2012.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] Nintendo Corporation. *Super Mario Kart Instruction Booklet.*

[5] OpenAI. *Gym Retro*, 2019.

[6] SethBling.

6 Appendix

6.1 SMK Rewards

First off, we need to have a clear goal. Since Super Mario Kart is a competitive racing game, each player is given a ranking when they finish the race. This gives us a well defined goal: be first. Since we will be training over a cup, multiple tracks, we do not need to place first in every race to win the cup, but placing first in every race will always result in placing first in the cup. The position can be seen in the bottom right of the screen, in our picture we start in 8th place (starting position changes depending on finishing position of the previous race).
The second factor that is key is that we have a clock, seen in the upper right of the screen. While we want to place first every time, setting a reward function purely based on position will only train the agent to beat the game's AI. Instead we want the agent to finish the race as fast as possible.
There are other factors to include that we will not consider until combinations of the first two are tested. Such factors include that the number of coins, with maximum of 10, determine how fast one can go. So having more coins helps optimize the first two, but may be harder to incorporate. Additionally, there are pieces on the may that boost the player. We hope that the agent will recognize these features on its own and that we do not need to assign a reward for using these. Including features like these have a higher potential to create an undesirable reward function. So first we will keep the reward system simple and iterate as needed.

6.2 Determining Finish

As mentioned previously, by incorporating a game that is not already defined by retro we need to tell retro when we are done. To accomplish this we will actually need to play this game in an emulator and record the state when finishing a race as well as when we finish a cup. This task is not yet accomplished but is not expected to be exceedingly difficult.

6.3 Seeing the Environment

We have mentioned that we will only be using the top half of the screen, the reason being is that this is where the agent gains the most information about their environment. Additionally, when playing in 2 player mode, the bottom half will be used for that agent's view. Therefore to train properly we need to only rely on the top half.
Retro does not create a way for the agent to "see" its current state. To accomplish this we need to turn this Q learner into a Deep Q learner. To do that we first create a Convolutional Neural Network (CNN) that will determine our state. Using this we will have a simple network which will then determine the action that needs to be taken at any given state, position on the track.
A simple way to think about this flow is with the following image. Using the position of Mario and other objects on the screen, the agent needs to determine if it will run or jump. Our action set will be slightly more complicated.

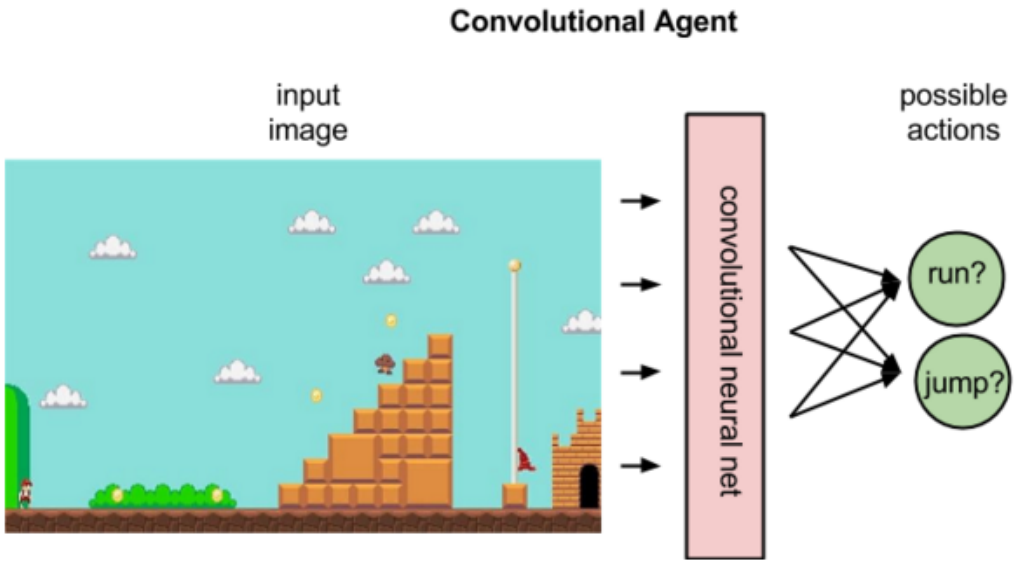

Figure 7: Example flow of network with Mario Brothers

6.4 RAM Addresses

- Lap: a value of 133 represents the end of the race and that one needs to subtract 128 from this number to determine the current lap.
- Rank: Divide by 2 and add 1
- Lap Size: Represents the length of the track
- Checkpoint: Shows distance from the start of the track. Increments up to the lap size - 1. Eg: if Lap Size = 36, then Checkpoint=$N\%36$.

Address	Attribute
0x000101	Milliseconds
0x000102	Seconds
0x000104	Minutes
0x000148	Lap Size
0x000E00	Coins
0x001040	Rank
0x0010C1	Lap
0x0010DC	Checkpoint

Table 1: RAM Addresses for key features in SMK

6.5 Figures

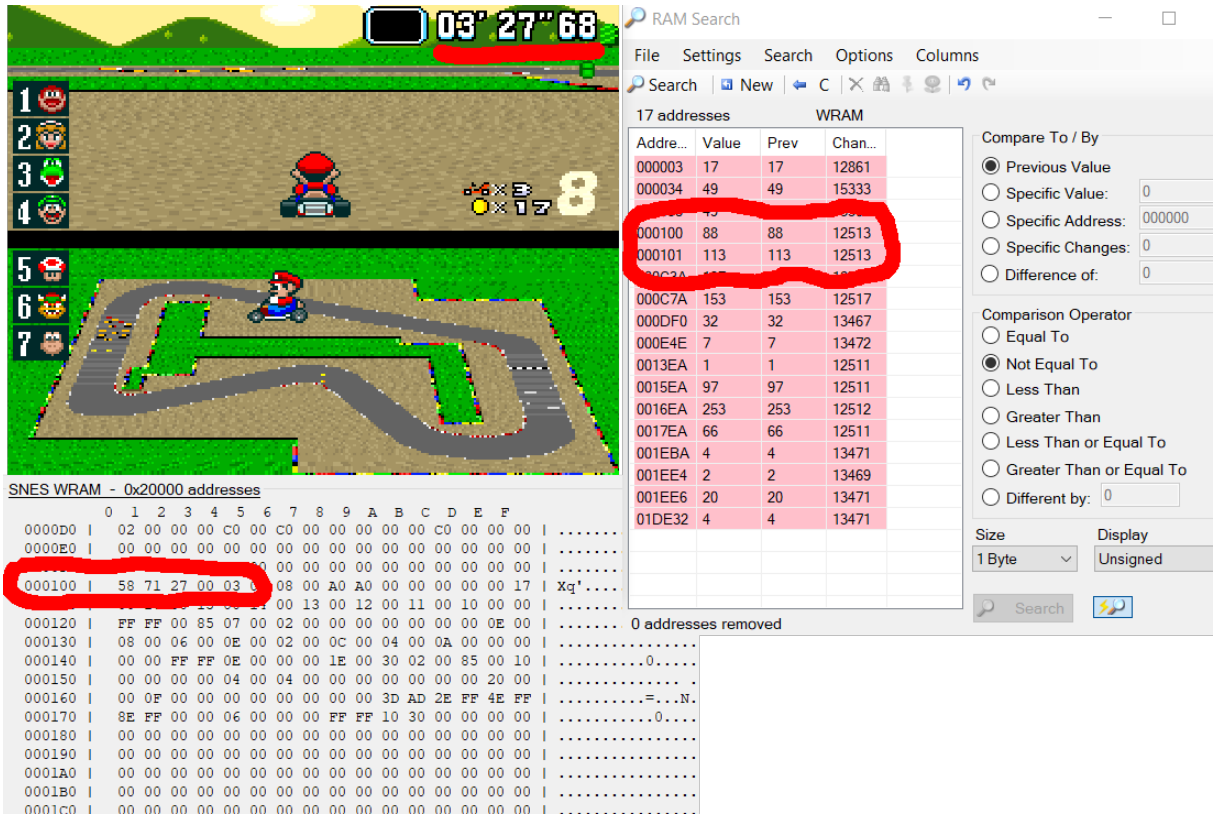

Figure 8: Finding Clock Addresses in BizHawk