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[**?**]
In this project we set out to create two competitive agents that will be able to play Super Mario Kart (SMK) against one another and the game's agents. An extra goal is to be able to play against these agents in a match. A full description of Super Mario Kart can be found in the official documentation.
During the project we had failed to import SMK into OpenAI's retro gym but we were still able to successfully implement a Q-Learner that is able to play games that correctly import into retro. We have tested the learner on Airstriker Genesis, the default game, and Donkey Kong Country.

2 Short 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.
For this project we will be using a Deep Q Learner. A simple Q learner and explanation can be found in this gist that I wrote. A link is also provided to explain how a deep Q learner works.
For this problem we will be using OpenAI's retro gym. Similar to gym, retro gym creates and environment that allows us to play "retro" video games. It handles all the emulation and provides us with bindings to the buttons that would be pressed on an actual controller. Retro also provides some states and actions for common games, though not Super Mario Kart (more on this later). Using this module a user can focus on just creating a reinforcement learner to learn to play the game and they do not have to think much about the environment or action states, that is: they do not need to formulate the quality matrix to create a basic learner.

2.1 Integration into Retro

Currently we have completed the task of integrating the Super Mario Kart package into retro's engine, but this is not enough to play the game. 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 are starting 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 1: 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

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.

To summarize the integration: we need to fully define how our agent will receive rewards, when the agent ends, and our action set.

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

2.2 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. For this game there are several criteria that will help us create different reward functions to test out.

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.

2.3 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.

2.4 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

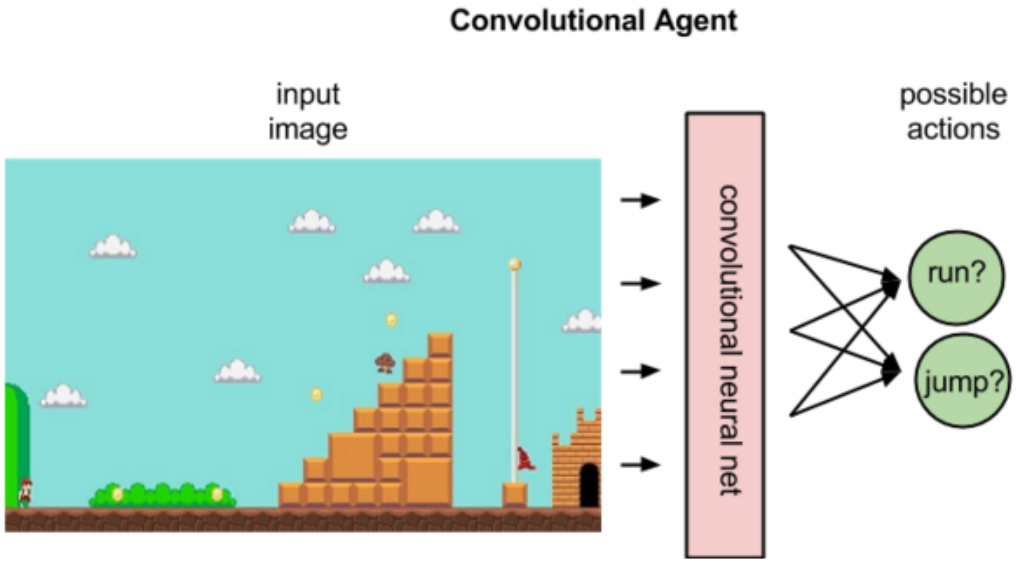

Figure 3: Example flow of network with Mario Brothers

other objects on the screen, the agent needs to determine if it will run or jump. Our action set will be slightly more complicated.

2.5 Policy Gradient

Since we do not have labeled solutions to perform an error with back propagation, our reinforcement learner uses a policy gradient. Having defined our reward structure we can teach the network that certain actions are better than others. Essentially we are teaching the learner a set of actions that

will beat the game. In a solved game we would expect this set of actions to converge on the solved action sequence, but in a game like this we do not have such a solution so can only base it upon how well it ranks.

3 Implementation and Experimentation

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 discusss 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. This may be because not every system has a rambase value.
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 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. For this we used a tool called BizHawk. BizHawk is a tool used by supped runners (people who try to finish a game as fast as possible) to investigate RAM values. This is a Windows only tool. Some values may be easy to find, such as the second in SMK. To find the seconds on the clock we look for RAM addresses that are constantly changing. We then run around the race, performing different movements and we narrow down the list of addresses from hundreds of thousands to a handful. From here it is generally easy to test each address, by modifying or freezing the values. When performing this we found the milisecond address and then opened up RAM watch and looked at the adjacent values, which matched the clock. It is common for values to be grouped like this.
Other values may be much more difficult to find. For example in SMK the address to determine your rank is 0x001040. Unfortunately to get the desired value you have to divide the RAM's value by two and add one. Addresses like this can be extremely difficult to find and commonly speed runners script this. I was unable to find these on my own and was able to get help from a prominant speed runner named Seth Bling. He provided me a Lua script to find these values.

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. 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 provies 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. 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 indicies and the value of those actions in that state as entries. Using this we can iteratively find the best policy to accomplish a task.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r + \gamma \max_{a_t+1}(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)))$$

α : Learning Rate

r : Reward at current state

γ : discount rate

This is a recursive equation where we update the reward by looking ahead to see the rewards we previously received by doing certain actions. There are a few things to notice about this equation. The first is that each $Q(s_{t+1}, a_{t+1})$ follows the same equation, by recursion. The next part is that the discount rate, γ, will be multiplied by itself each time we recurs. The discount helps us find solutions that are not just locally optimal, but pick actions that help us gain future rewards. Our discount rate sets how heavily we weight those future rewards. The reason we do this is so that our learner cares more about immediate future rewards than rewards that are distant. If this were not used our learner could spend all eternity doing nonsense actions and then just before the eternity is over it could reach the goal. We want the goal to be reached sooner than later, so we implement this value.

There is another subtle aspect to making a Q-Learner work that isn't apparent from this equation. A Q-Learner is composed of two competing entities, which we will call the gambler and the book keeper. These two entities are similar to a random agent and a greedy agent. The gambler wants to explore and try actions it has has never tried before. On the other hand, the book keeper has recorded the best actions and strictly goes by the book, following what it says the best actions are.

While the book keeper is extremely important, the gambler is needed to help escape from local optimal solutions.
Another subtle aspect is that we pick the future action with the maximal reward. 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

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

u : utility from action

n : number of times action has been taken

k : a constant

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.

3.4 Experimentation

By using the above concepts we can put together a machine learning algorithm that can play videogames.