

CIS 510: Competitive Super Mario Kart

Deep Q Reinforcement Learning for Multiplayer Super Mario Kart

Steven Walton

May 1, 2019

1 Project Description

In this project we will be creating two competitive agents that will be able to play Super Mario Kart against one another. 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.

2 Problem Formulation

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 The Quality Matrix and Q Learner

Reinforcement learning depends on what is called a quality matrix, or q matrix for short. The q matrix is composed of states and actions, where each row represents a different state of the game and each column represents a different action. In a learning environment a q matrix will be initialized where all elements are 0 or a random number. A learner will then navigate the space and iteratively adjust the elements of this matrix. The learner will do this in two different ways: they will explore, that is making “random” moves testing out the environment and outcome of certain actions; and by relying on historical solutions from the previous iteration of the q matrix. We can define this as the following

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a (Q(s_{t+1}, a) - Q(s_t, a))) \quad (1)$$

Where

$Q(s_t, a_t)$	is the quality matrix at time t
α	is the learning rate
r_{t+1}	is the reward
γ	is the discount
\max_a	is the maximal reward from Q given a specific action in the action set of s_{t+1}
$Q(s_{t+1}, a)$	is the Q matrix at the next state, s_{t+1} , and a is the action set at that state

We can represent problems with a Markovian Decision Process (MDP) and this fills our Q matrix.

A simple example of a q learner and the associated MDP can be found in this gist that I created. It also contains a more detailed explanation of a Q learner and Deep Q Learner.

2.2 Retro

For this problem we will be using OpenAI's retro gym. Similar to gym, retro gym creates an 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.

2.3 Super Mario Kart

This game was chosen for a few reasons. Besides the fact that it is extremely popular, though less popular than its successor (Mario Kart 64), the environment is a nice balance of complication for the allotted time of this course.

Super Mario Kart is a racing game that has different speeds, characters with unique traits, power items, and has a multiplayer aspect. Though the reward function is difficult to define, there is a clear goal to the game: place first. Excluding glitches, it is always optimal to have forward progression. We have also speculated that by having multiple agents compete against one another within the same race that learning will happen faster and not be limited by only being better than the game's AI.

3 Proposed Solution

Completing this project will require several tasks. First it will require importing the game into retro's engine. Next it requires defining the reward function, which means

that the state and action set needs to be well defined. Lastly, the agents need to be trained to complete the tasks.

3.1 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 with the Nintendo Entertainment System’s (NES) Mario Brothers are shown below.

Additionally, scenario.json can contain information about the action set. We can

Listing 1: data.json

```
{
  "info": {
    "level": {
      "address": 65,
      "type": "|d1"
    },
    "lives": {
      "address": 72,
      "type": "|u1"
    },
    "score": {
      "address": 148,
      "type": ">d4"
    }
  }
}
```

Listing 2: scenario.json

```
{
  "done": {
    "variables": {
      "lives": {
        "op": "equal",
        "reference": 0
      }
    }
  },
  "reward": {
    "variables": {
      "score": {
        "reward": 1
      }
    }
  }
}
```

Figure 2: data.json and scenario.json for NES Mario Brothers

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.

3.2 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 way 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.

3.3 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.

3.4 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

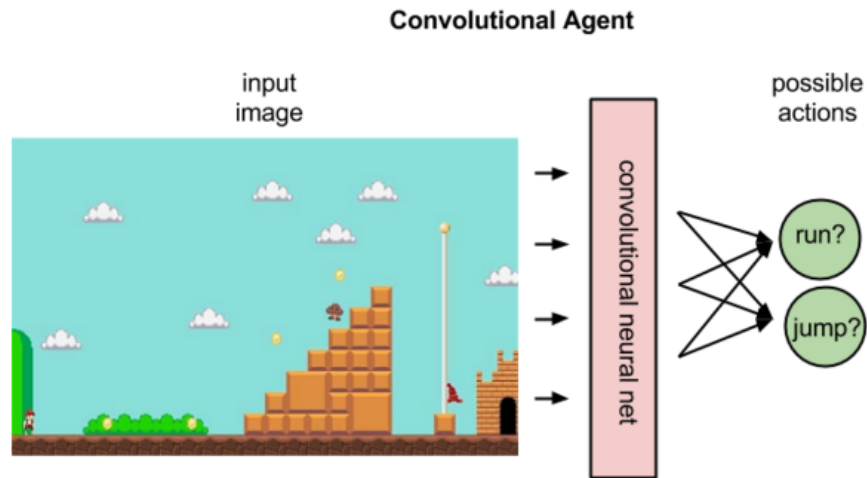


Figure 3: Example flow of network with Mario Brothers

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.