Reinforcement Learning in Mario Kart Wii

Harry Stevenson

Student ID: 2314315

BSc Computer Science

Supervised by Leandro Minku

Word Count: 8173

40 Credits

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2023-24

Abstract

In recent years, Reinforcement Learning (RL) has been studied extensively, with an apparent exponential growth of publications. Deep Reinforcement Learning (D-RL) accounts for approximately 30 % of RL publications in the last 5 years. In this paper I formulate a kart racing game as both and RL and Deep-RL problem and compare both approaches to human players, considering trained performance, resource requirements and adaptability.

Acknowledgements

I would first like to express my gratitude to and acknowledge my project supervisor, Leandro Minku. His sustained support and guidance in meetings always brought a wave of relief when I was feeling unsure of my project direction. Secondly to the many members of the Mariokart Wii TAS community, whose existing efforts in developing cheat codes and a python-embedded emulator made this project possible. Finally to Benjamin Middleton, who's Deep-RL implementation was an essential fallback due to an error with mine.

Abbreviations

RL	Reinforcement Learning
D-RL	Deep Reinforcement Learning
CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
MDP	Markov Decision Process
HER	Hindsight Experience Replay
DQN	Deep Q-Network
ALE	Arcade Learning Environment
MOBA	Multiplayer Online Battle Arena
H-RL	Hierarchical Reinforcement Learning
FPS	First Person Shooter
fps	frames per second
API	Application Programming Interface
DME	Dolphin Memory Engine
MKW	Mario Kart Wii
OCR	Optical Character Recognition
WR	World Record
TAS	Tool Assisted Speedrun
MT	Miniturbo

Contents

List of Figures

List of Tables

Introduction

Games, despite primarily being recreational hobbies, can be used as useful environments to test and compare the effectiveness of new developments in AI. Mnih *(et al.)* (2015) [21] were able to achieve a performance level comparable to that of a professional human games tester in the Atari Learning Environment (ALE) [5] through an algorithm called DQN [20]. These developments in AI, specifically RL, for games can lead to insights into other disciplines, informing decision making in a wide range of situations. An example of an interesting and potentially life-changing discipline is self-driving cars, where this research has helped lead to successful demonstrations of end-to-end driving [11] and high-level decision making [13]. However, these complex developments come at a cost. Image processing in particular is heavily GPU reliant and therefore not always accessible, particularly to those without this hardware. My project considers the effectiveness and relevance of 'traditional' RL approaches compared to D-RL when applied to the same scenario. I aim to achieve this by giving two formulations of my chosen environment, one for RL and one for D-RL, along with comparing the corresponding trained agents against each other and human performance. Through this, I investigate whether human level performance is also achievable through 'traditional' RL approaches. Specifically, I compare Q-Learning (RL) [36], a model-free RL algorithm, and Rainbow (D-RL) [18],a variation of DQN. The scenario I have chosen for this is Mariokart Wii. The game is hugely popular, with over 38 million copies sold. This means that this is a familiar environment for many, unlocking the potential for an application in education. This, paired with my great interest and passion for the game, made it a clear choice for my project.

CHAPTER 1

Background

1.1 Mario Kart Wii

Mariokart Wii is a popular local and online multiplayer racing game for the Nintendo Wii, where players select from a wide range of characters and vehicles to compete in a variety of game modes; including 3 lap races, time trials and battle modes. Despite official online servers shutting down in 2014, community-run servers continue to support a dedicated fanbase. Within this fanbase many members spend their time to creating Tool-Assisted Speedruns (TASs), where creators step through a race frame by frame and manually set the controller inputs for each frame. This is then replayed by a computer player at full speed. While this has proved very effective in producing extremely quick lap times, sometimes even exploiting quirks in the physics engine, it is a long and complicated process, requiring many hours to make a TAS. Reinforcement Learning, potentially in combination with other AI approaches, will be be useful to this community in discovering new strategies or exploring existing ones.

Figure 1.1: Screenshot from Mariokart Wii[31]

1.2 Reinforcement Learning

Reinforcement Learning describes an interaction between an agent and an environment, in which an agent takes actions in the environment, changing its state, associated with a reward[29]. In contrast to traditional AI approaches, where a learner is *taught* which actions to take, RL *learns* by itself from its own actions. One field of RL is episodic RL, in which the training process consists of many discrete episodes, starting from an initial state and ending in a terminal state.

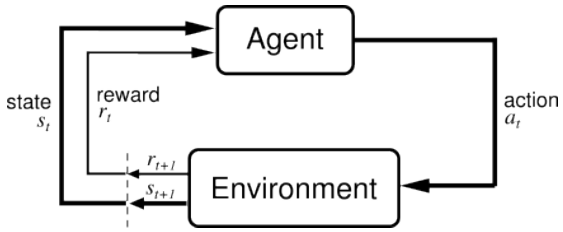


Figure 1.2: Generic reinforcement learning architecture[29]

1.2.1 Markov Decision Process

Many problems in RL can be modelled by a 'Markov Decision Process', first introduced by Bellman in 1957 [6]. This is a structure which models sequential decision making problems. This is well-suited to RL, due to the repeated decisions the agent takes.
A standard MDP for optimisation is a tuple of

$$< S, A, P, R, \gamma >$$

where

- S is the set of states the agent can be in (the *state space*),
- A is the set of actions available to the agent (the *action space*),
- P is the transition probability of moving from one given state to another, taking a given action,
- R is the reward associated with taking an action in a given state, moving to another state,
- γ is the discount factor, which balances short and long-term rewards

States and The Markov Property

For an MDP to be effective, it must satisfy the Markov Property [24]. This specifies that the next state depends solely on the current state, and not on any states that preceded it. Formally, at each time-step t,

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_t, S_{t-1}, \ldots, S_0)$$

Essentially, this means that all necessary information about the state must be included at all times.

1.2.2 Q-Learning

Q-Learning [36] is a model-free RL algorithm, which maintains and updates a Q-table containing values of taking given actions in given states. This table serves as a direct approximation of the optimal action-value function. As an agent transitions from one state to the next, the table is updated according to the following update rule

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \Big(r + \gamma \cdot \max_{a'} Q(s',a')\Big)$$

where:

- s is the previous state
- a is the previous action
- $Q(s,a)$ is the Q-value of state-action pair (s,a)[1],
- α is the learning rate, affecting how much new updates affect the stored value,
- r is the reward earned by taking action a in s, moving to s',
- γ is the discount factor, weighting potential future rewards,
- s' is the current state, and
- $\max_{a'} Q(s',a')$ represents the maximum Q-value for the current state.

It is proven that Q-Learning, with a finite state-action space, will always converge to the optimal action selection policy.

1.2.3 Deep Reinforcement Learning

Mnih *et al.* (2015) [21] were successful in achieving human-level control with a Deep Q-Network (DQN), a deep neural network that acts as an estimator for the Q-function. RL transitions [2] are stored in a replay buffer which is randomly sampled from. These samples are then used to optimise the parameters of the neural network, in this case done by gradient descent on the mean squared error. Additionally, DQN makes use of 2 concurrent neural networks, the *online* network and the *target* network. The *online* network is directly optimised through backpropogation and used for action selection, and the *target* network is a periodic copy of the *online* network. The *target* network is used during the maximisation step of the mean squared error calculation and has the effect of improving the algorithm's stability. The loss function is as follows:

$$(R_{t+1} + \gamma_{t+1} + \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2$$

where

[1] A Q-table is usually initialised to 0, assuming no knowledge about the environment, however initialising randomly can help the agent to explore the search space.

[2] A record of an interaction in RL, containing the current state s, the action taken a, the reward earned by taking that action r and the next state (result of taking action a in state s)

- t is a time step picked randomly from the replay buffer,
- R_{t+1} is the immediate reward of taking action A_t in state S_t,
- γ_{t+1} is the discount factor at step t,
- $\bar{\theta}$ is the parameters of the *target* network,
- θ is the parameters of the *online* network.

This success spearheaded further research into the area, resulting in advancements and extensions to DQN. Including, but not limited to: Dueling DQN [35] and Double DQN [33]. These approaches outperformed traditional RL in all tested cases, but are particularly suited to problems with very wide state spaces, where a Q-table would become intractably large. Additionally, the high-dimensional input gives this model better generalisation capabilities than Q-learning, in which a small change in the state space can greatly decrease the performance of a previously effective agent.

CHAPTER 2

Literature Review

2.1 Existing Work

Due to Mariokart Wii's popularity, it is no surprise that similar projects have been created. Jack Boynton (2022) developed a D-RL Mariokart Wii agent for a presentation he gave on Deep-Reinforcement Learning [12]. This agent was His implementation used a built-in checkpoint system for the reward function and a series of positions, vectors and pixel data for the state representation. One aspect his agent struggled with was the sparsity of rewards due the to nature of the checkpoint system. To address this he used Hindsight Experience Replay Buffer (HER) [2], an extension to a standard replay buffer which generates additional transitions and goals, improving sampling efficiency. An interesting component of the state space he designed was cross-track error, which measures the kart's distance from a pre-defined desired path. This component, while effective in its specific use case, would require a new desired path to be created for each track, which would need knowledge of the track boundaries and quickest route prior to training.
Ben Middleton's AI Environment (2022)[7] implemented a version of DQN called Rainbow [18], which I explain in detail later. This implementation used pixel data combined with key values from Dolphin's emulated memory as the state representation and the current speed of the kart scaled to the range $\{0, 1\}$ as the reward.

2.2 Reinforcement Learning in Games

Games, being a popular hobby, have naturally stood out as an enticing environment to showcase advancements in Reinforcement Learning. They suit the paradigm well, due to the huge range of games available and existing tools available for interaction through code. One such tool is the Arcade Learning Environment (ALE) [5], a learning environment which provides an interface to

hundreds of Atari 2600 games, spanning from simple games such as *Pong* to complex platform games such as *Montezuma's Revenge*. Since publication, this environment has become a go-to for many researchers due to its size and diversity. For example Mnih *et al.* (2013) [20] proposed a D-RL model known as a Deep Q Network (DQN) and compared it to two existing approaches (Contingency Awareness [3] and SARSA [27]). DQN outperformed both existing approaches in all tested cases, and outperformed human players in all but one case.

2.3 Real-World Applications

This research also has various relevant real world applications. For example, E-Sports is a new and rapidly-growing industry [9], with a wide range of game genres supporting immense fanbases, the most popular being Multiplayer Online Battle Arenas (MOBAs). These are highly complex player versus player environments, with an estimated state space in the magnitude of 10^{600}. Ye *et al.* (2020) [37] beat 5 professional MOBA players with D-RL proving the potential for learning strategies from AI models in E-Sports.
Following a similar trajectory, RL's application to motorsport can help with race strategy, where balancing the benefits of fresh tyres with time lost in the pit lane can have a great effect on the race result. Boettinger & Klotz (2023) [10] developed a race simulation model to automate race strategy decisions, exemplifying the potential for a change in real-time decision making processes for race engineers, an integral role in any motorsport team. Comparably, Remonda *et al.* (2022) [26] used telemetry data from a racing car simulator to learn to drive around racetracks, surpassing the baseline simulator bots. Interestingly, the generalisation capabilities of the model differed greatly depending on the training data. For example, when an agent was trained on a simpler track with fewer, wider turns it was not able to complete a more complex track, however the opposite was not true.

CHAPTER 3

Emulator

3.1 Motivations

In order to perform RL in my chosen application I required some Emulation software. Dolphin Emulator [17] is a free, open source Wii and GameCube emulator. My past experience with Dolphin, along with its wide use within the retro gaming community were the initial motivations for choosing Dolphin. However, as I researched deeper into the emulator's capabilities, I found that it has a plethora of useful features, making it an appropriate choice for my project. This section outlines what these features are and why they are useful to me.

3.1.1 Savestates

The savestate system allows for saving to and loading from predefined 'slots' and files. This is extremely useful when performing episodic RL as the environment will need to reset to an initial state s_0 when a terminal state is reached. To implement this I can manually move the character to a desired initial state, in my case the start/finish line), and save it to a specific slot. Once it has been saved, I can load that specific state when required.

3.1.2 Framedumps

Configuring the emulator to perform Framedumping saves the eacg frame's pixel data to a specified directory as a png file as soon as it is rendered. The file is named sequentially based on the order of rendering, i.e. the first frame of an emulation is *framedump_0.png*. Thus allowing access to the pixel data without any interaction with Dolphin itself. This data will be required as input to the D-RL agent's neural network.

3.1.3 Emulation Speed

Dolphin gives the user the option to set a specific emulation speed. This allows the games to run many times quicker than normal, only limited by the user's hardware. This feature helps in decreasing the total amount of training time.

3.1.4 API Access

An unofficial version of Dolphin with integrated API access has been created by the TAS community. Currently known as *Pycore*[30], this version provides an API with Dolphin, allowing for programmatic access and control of a few core features, including:

- Waiting for the next frame to be drawn
- Reading a given location in the game's memory
- Controlling an emulated controller
- Loading a savestate [1]

3.1.5 Multiple Concurrent Controllers

Dolphin also supports local multiplayer, allowing multiple emulated or real controllers connected at one time, enabling an interactive comparison between different amounts of training time and human performance.

3.1.6 Cheat Codes

Due to the age and nature of the game, many cheat codes for Mariokart Wii have been created, known as *Gecko Codes*[34]. Traditionally cheat codes are synonymous with gaining an unfair advantage in online play, however many of these differ from that. The two that I used during this project gave me a fully completed game save (unlocking all characters, vehicles and tracks) and displays many key values from Memory on the screen, known as the *Debug Panel*.

3.2 Emulated Controller Choice

Mariokart Wii permits 4 controller types:

- A standard Wii Remote turned sideways and usually in a 'Wii Wheel' accessory
- A Wii Remote with the 'nunchuck' extension
- A Wii Classic Controller
- A GameCube[2] Controller

[1] As perIssue #123, this is currently not working, however a workaround involving pressing a hotkey assigned to loading a savestate slot is effective

[2] The Nintendo Wii's predecessor

Figure 3.1: Screenshot of Mariokart Wii with the *Debug Panel* cheat enabled

(a) A Wii Remote in a 'Wii Wheel'

(b) The 'nunchuck' extension

(c) A Wii Classic Controller

(d) A GameCube Controller

Figure 3.2: Mariokart Wii's permitted controllers [14] [23]

Out of these available options, I chose to use an emulated GameCube controller. The main reason for this is the clear distinction of its inputs. In contrast, the Wii remote uses motion controls for steering. To avoid a missed detail in my formalisation, I decided that purely button or stick inputs would be the best way forwards. This choice eliminates any uncertainty in regards to motion control actuation zones/values etc. The main actions available to control the kart are:

- 'A' Button = Accelerate
- 'B' Button = Performs 2 different actions:
  - → Brake (when no direction is held)
  - → Drift (when a direction is held)[3]
- 'R' Button = Performs the same as the 'B' Button
- 'Up' Button = Performs a wheelie, increasing top speed, but heavily decreasing handling[4] [5]

[3] A drift's direction cannot be changed during a drift, but can be adjusted. That is, a drift that is started while holding right will always drift to the right, but keeping 'B' held and steering left will make the turn less sharp.

[4] A wheelie can only be performed on 'bike' vehicles

[5] The button does not need to be held to perform the wheelie. Once pressed, the wheelie

- Main Control Stick = Steers the kart/dictates direction of drift
- 'L' Button = Uses an item

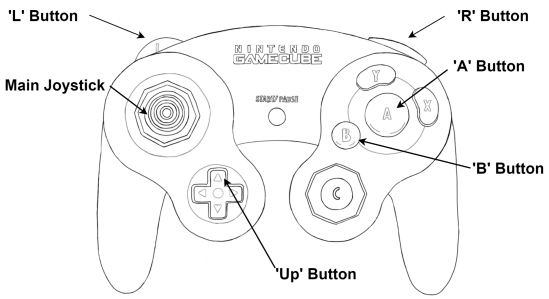


Figure 3.3: Diagram of GameCube controller with labelled inputs[32]

3.3 Memory Access

The values shown on-screen in the *Debug Panel* provide a lot of information about the current state of the kart, giving me lots of options to consider for my state space. In order to implement the state space, I would need a way of extracting these values from the screen. My initial idea was to use Google's Optical Character Recognition (OCR) software Tesseract [16] to read these values from the screen, but due to issues with external packages in Dolphin's embedded Python I had to reconsider this choice. Instead, I chose the read the values directly from memory, meaning I needed a method for finding a memory address, given its current value. For this I used Dolphin Memory Engine [1], a tool that allows for filtering and searching through Dolphin's emulated memory, achieved by hooking to the current Dolphin process.

3.3.1 Memory Search Method

I used an iterative search process to find the required memory addresses. Just searching one value at one time gives many possible addresses, so I needed to filter out addresses that just matched my required value by chance. Let's say I am searching for the memory address that contains the current speed of the kart. We know that it will most likely be a float, as speed is real-valued. To get a baseline, I search the memory for any float with value 0, discarding all other locations. I can then accelerate up to a certain speed and search the remaining memory for that value, which I know from the on-screen readout. Repeating this elimination style search proved effective in finding the locations of all the values I required, those being:

- **Speed** = The current speed of the vehicle
- **Race Completion %** = A number from 0 to 4 representing the vehicle's progress through the course. Where the whole number portion represents

will continue for a short period, unless it is cancelled by pressing 'B'

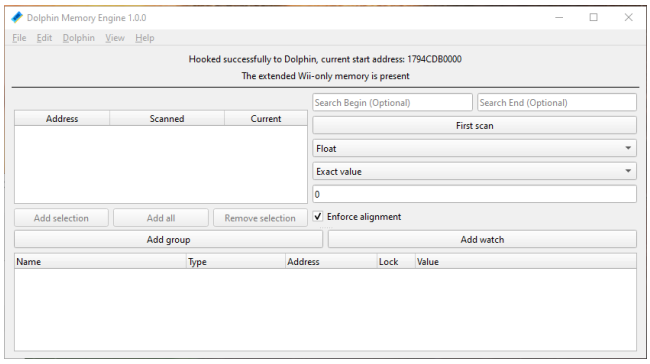


Figure 3.4: Screenshot of Dolphin Memory Engine

the current lap and the decimal portion represents the progress through that lap.

- **X Position** = The current X co-ordinate of the kart
- **Z Position** = The current Z co-ordinate of the kart[6]
- **Miniturbo (MT)** charge = (when drifting) How long a drift has been held for[7]
- **Wheelie** = Whether the vehicle is currently performing a wheelie
- **Road Type** = What type of road surface the vehicle is on

Once found, I stored the addresses to read during runtime, which would utilise the memory module of the API, giving live access to the emulated memory.

[6]Mariokart Wii uses the X-Z plane for horizontal positions
[7]In the range of [0,270] with 0 representing no charge and 270 representing fully charged

CHAPTER 4

Design - Q-Learning

4.1 Problem Formulation

To perform the Q-learning algorithm, I represent the problem as an MDP(1.2.1). However Q-learning, being a model-free RL algorithm learns purely through interaction with its environment, meaning it does not require the transition probabilities P. This gives my formalisation the structure of

$$< S, R, A, \gamma >$$

4.1.1 State Space

When designing the state space, it was important to balance both accuracy and simplicity. A state must maintain the Markov Property, in that it contains all necessary information, and must not be too detailed, doing so can lead to an extremely large state space, and possibly an intractable problem. To represent the state space, I used the following components:

Speed

The speed of the vehicle is an important factor to consider when racing. A higher speed means a quicker lap time, which is what we are optimising.

Position

The position of the vehicle is also integral to the RL process, as moving its position around the track to the finish line is the ultimate goal of the agent. To represent the agent's position I used X and Z components of the bike's co-ordinates. Choosing a suitable degree of rounding accuracy is an impoertant consideration for my state space representatino. I chose to train 3 agents, each with varying degrees of precision in its position, for 5000 episodes.

- **Agent A** = Accurate to 1 d.p.

- **Agent B** = Accurate to the nearest Integer
- **Agent C** = Accurate to the nearest 5th Integer[1]

The Y component was not required in my environment, as the track I have chosen is completely flat, however on other tracks this would need to be included to account for places where the track overlaps itself.
When choosing my representation for the position of the kart I had two options, those being the XZ coordinates and the completion % .In earlier versions of my representation, I instead used the completion value as it seemed to be a precise representation of the karts position. However after some failed trial runs I investigated this value further, and from these I found that the position of the kart across the track has no effect on the completion % making it unsuitable for my implementation. This can be seen in the below diagram, the locations of the karts are different but the completion values are the same.

(a) Kart positioned on the start line in the middle of the track

(b) Kart positined on the start line on the side of the track

Figure 4.1: Demonstration of the inaccuracy in Race%

Drift

To progress around a corner, a kart must either turn or drift in that direction. If the kart is drifting, then its physics change slightly, depending on the direction that the drift was started in. Additionally, after holding a drift for approximately a second a 'miniturbo' (MT) is charged, resulting in a great speed boost when the drift is released. This dynamic gives me 2 elements to track: The start direction of the drift, and how charged the miniturbo is.

Wheelie

When driving in a straight line, performing a wheelie increases the top speed by 12 mph [2]. This does however greatly decrease the turning capabilities of the bike. Because of this change to the steering, it is important that the bike's wheelie state is included in the state information. Performing a wheelie only requires the button to be pressed once, after which the bike enters a wheelie for approximately 3 seconds, unless interrupted by the player.

This design gives me a state space in the magnitude of 10^6.

[1] To achieve this I perform integer division by 5, essentially mapping every 5 coordinates to one in each axis

[2] From 84mph to 96mph

Terminal States

For episodic RL, I need to define a set of terminal states, which indicate that the episode has ended and needs to be reset. I want the episode to terminate if the vehicle's speed is below a certain value, if the vehicle is off the track, or if a lap has been completed.
In Mariokart Wii there are many different types of track surface, such as normal road, off-road and boost panels. On the track I have selected, an area of off-road surrounds the entire track, allowing me to check the type of surface the vehicle is on to find whether it is off the track.
I have also set a low speed bound, terminating the episode if the vehicle is below a certain speed. Additionally, hitting the obstacles on track doesn't change the type of road we are on, but will greatly decrease the speed, further justifying this condition.
Finally, I want to check whether an episode has been successful in completing a lap, given by the completion % value. If this is > 2 ,meaning we are currently on lap 2), then a lap has been completed.[3]
The set of terminal states can be formalised as the following:

$$S_t = \{s \in S | s_{roadType} = \textit{offroad} \text{ or } s_{speed} < minSpeed \text{ or } s_{completion} > 2\}$$

where

- $s_{roadType}$ is the type of road in state s
- *offroad* is Mariokart Wii's encoding of off-road road type
- s_{speed} is the speed of the vehicle in state s
- $minSpeed$ is a constant representing the minimum acceptable speed[4]
- $s_{completion}$ is the completion value in state s

4.1.2 Reward Function

The reward function is a key element of the learning process, serving as an evaluation of the agent's performance. To design the reward function I investigated the fastest route around the track by looking at the current top speedruns. From these I saw that the fastest lap around the track consisted mostly of drifting, to charge miniturbos around the corners, and performing wheelies on the straights. This behaviour is what I want to encourage through the design of my reward function, as well as making progress through the track.

Speed Component

To encourage the agent to perform wheelies, I give a small bonus to the reward if the current speed is greater than the normal top speed of the kart. I also reward the agent for staying above the minimum acceptable speed, and no reward for

[3] This can be changed to 4 to allow the agent to train to complete an entire 3-lap time trial
[4] set to 60mph

terminating.

$$R_{speed}(s') = \begin{cases} 1.5 & \text{if } s'_{speed} > maxNormalSpeed \\ 1 & \text{if } s'_{speed} > minSpeed \\ 0 & \text{otherwise} \end{cases}$$

where

- s' is the current state
- s'_{speed} is the speed in the current state
- $maxNormalSpeed$ is the maximum speed of the vehicle [5]
- $minSpeed$ is the minimum acceptable speed of the vehicle, defined in 4.1.1

Completion Component

In order for the agent to complete a lap, I want it to progress forwards around the track. To encourage this, I use the completion value of the current state and compare it with the completion value of the previous state. I also provide a large reward for completing a lap.

$$R_{completion}(s, s') = \begin{cases} 100 & \text{if } s'_{completion} > 2 \\ 1 & \text{if } s'_{completion} > s_{completion} \\ 0 & \text{otherwise} \end{cases}$$

where

- s is the previous state
- s' is the current state
- $s_{completion}$ is the completion value of the previous state
- $s'_{completion}$ is the completion value of the previous state

MT Component

To encourage the agent to drift around corners instead of simply turning, I give a large reward for fully charging and releasing a miniturbo, and a penalty for releasing a miniturbo before it has fully charged. This will encourage the agent to fully charge a miniturbo before it is released, ensuring it gets the speed boost, as well as discouraging it from starting to drift on straight parts of the track.

$$R_{mt}(s, s') = \begin{cases} 1 & \text{if } s_{mt} = 270 \text{ and } s'_{mt} = 0 \\ -1 & \text{if } s_{mt} < 270 \text{ and } s'_{mt} = 0 \\ 0 & \text{otherwise} \end{cases}$$

where

[5] While not in a wheelie or speed boost. For the chosen character and bike combo, this is 84mph

- s is the previous state
- s' is the current state
- s_{mt} is the MT charge of the previous state
- s'_{mt} is the MT charge of the current state
- 270 is a constant representing a fully charged MT
- 0 is a constant representing no MT being performed

Complete Reward Function

These components combined and weighted give my complete reward function:

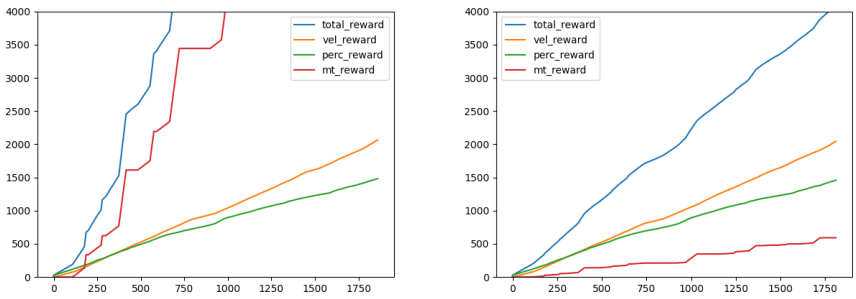
$$R(s, s') = w_1 \cdot R_{speed}(s) + w_2 \cdot R_{completion}(s, s') + w_3 \cdot R_{mt}(s, s')$$

where

- w_1 is the weight associated with the speed component
- w_2 is the weight associated with the completion component
- w_3 is the weight associated with the MT component

Reward Function Evaluation

In order to evaluate the effectiveness of my reward function, I designed and conducted and investigation process. This was to drive a fast lap myself, and record and plot the reward function values. From this data I tuned the weights and rewards so the function performs as I intended.



(a) Reward function of a fast lap before tuning weights

(b) Reward function of a fast lap after tuning weights

Figure 4.2: Reward Function graphs of a 'well driven' lap

4.1.3 Action Space

Defining the action space of my problem is also a very important task. Having too many actions can lead to greatly increased training times, due to the increased number of actions to explore in each state, but having too few can lead to an ineffective agent.

Buttons

As my agent will not be using items, I removed the 'L' Button from the available inputs. Also, as 'R' and 'B' perform the same actions, I chose to only use one of them, that being 'B'. This leaves us with 3 available buttons ('A', 'B' and 'Up'), each having 2 states, either pressed (*True*) or not pressed (*False*). As I always want the agent to progress forwards, I set the accelerate (A) button to always be held down.

Analog Stick

The analog stick allows for a combination of horizontal and vertical inputs, however for my application only the horizontal component is required. This leaves a range between 0 (full left) and 255 (full right). To simplify this, I defined 5 stick angles equally spread apart, representing soft/hard left, neutral and soft/hard right.
From the remaining inputs I defined a set of specific permutations that perform common desirable actions (4.4).

4.2 Implementation

In order to perform RL in my chosen environment, I needed to run code synchronously with the emulation. With this in mind I designed the following architecture, an augmentation of the Agent-Environment interface and Dolphin Emulator:
At each time step t:

- Use an epsilon-greedy policy to choose an action 4.2.3
- Perform the action using the *Controller* endpoint and move to the next state
- Observe the new state information using the *Memory* endpoint
- Calculate the reward earned by performing the action
- Update the Q-table for the previous state

4.2.1 State Space

My implementation of the state space involved reading the required values from memory and performing some slight processing. This is then stored as a python tuple to be used as part of the key for the Q-table.

- **Speed** was rounded to the nearest Integer,

- **Position** was rounded to the nearest Integer,
- **MT** was converted to a tuple where the first element was an Integer in the range [0,2] which indicates the direction of the drift

  → 0 = not drifting

  → 1 = drifting left

  → 2 = drifting right

  , and the second component was a Boolean indicating whether the mini-turbo was fully charged,
- **Wheelie** was converted to a Boolean.

4.2.2 Reward Function

The implementation of my reward function involves checking various conditions on the different components of the previous and current states and returning the appropriate reward. These components are summed and weighted to give the final reward for that step. I also return each component individually in order to track and tune the weights of each.

4.2.3 Action Space

To represent the available actions I used a list of Python dictionaries, where the name of the button was the key, and the state of the button was the value. This is the representation used in the API, so choosing to use this prevented the need for conversion when calling the endpoint. This action is converted to a tuple, containing only the values from this dictionary, when used in the key for the Q-table.

Action Selection Policy

For my implementation of Q-Learning, I decided to use an ϵ-greedy action selection policy, providing a balance between exploitation and exploration when an appropriate ϵ is selected. The policy states that, at any time step t, we pick a

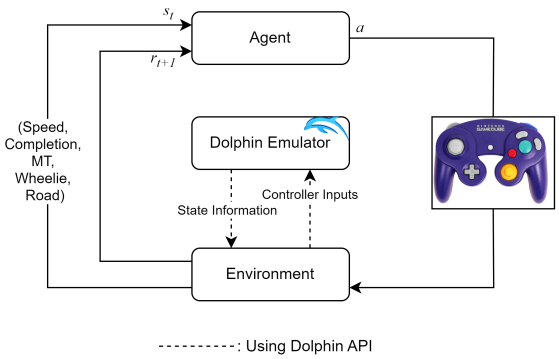


Figure 4.3: Q-learning system architecture

```
actions = [{"B": True, "Up": False, "StickX": 0}, # hard drift left
{"B": True, "Up": False, "StickX": 64}, # soft drift left
{"B": True, "Up": False, "StickX": 192}, # soft drift right
{"B": True, "Up": False, "StickX": 255}, # hard drift right
{"B": False, "Up": True, "StickX": 128}, # wheelie straight
{"B": False, "Up": False, "StickX": 128}, # drive straight
{"B": False, "Up": False, "StickX": 64}, # slight turn left
{"B": False, "Up": False, "StickX": 192}] # slight turn right
```

Figure 4.4: Available action permutations

random action with probability ϵ or pick the best action with probability $1 - \epsilon$. To implement this, I uniformly sample a number x in the range $\{0, 1\}$ and take a random action if $x < \epsilon$, or take the best action if $x > \epsilon$

4.2.4 Time Step

Dolphin Emulator runs Mariokart Wii at 60 fps (frames per second). My first implementation used a time step of 1, performing Q-learning at every frame. This experimental training run gave very ineffective results. After observing the agent during training, I found that one of the main reasons for this is that performing an action doesn't have an instantaneous effect in the context of the game. For example, performing a wheelie takes a few frames for the wheelie animation to play, and then a few more to accelerate to the new top speed. It is for this reason that the chosen time step has a substantial impact on the agent. Setting it too high could lead to the agent missing opportunities to take the optimal action at the optimal time, but setting it too low could lead to a greatly increased training time and erratic movement. To choose an appropriate time step for my environment I experimented with a few different values, eventually finding that 20 frames (0.333... seconds) was a good middle ground between accuracy of inputs and episode length.

4.3 Environment

To aid the agent in its training process, I made choices when designing the environment that focused on simplicity, aiming to minimise the search space.

4.3.1 Game Mode

As mentioned earlier, Mariokart Wii supports many game modes, however a full race with 11 other racers is a complex stochastic environment, with many unknown factors. To aid an untrained agent, I chose to use the 'Time Trial' mode. In a time trial, the player tries to set the fastest time possible around the track with no other racers, thus removing this stochastic element from the environment.

4.3.2 Track

The wide range of tracks in Mariokart Wii have varying difficulties. These start off with simple, wide tracks with few turns, but some of the more complex

Figure 4.5: Track layout

ones include 3-dimensional looping and twisting courses. The track that I chose (Called SNES Mario Circuit 3 4.5) is a simple in comparison to these, but as a pure racetrack it is relatively complex. Differing tightnesses and lengths of corners along with a few on-track obstacles make this track surprisingly difficult to navigate, despite its flat layout. These features, in my opinion, provide enough simplicity to aid learning, and enough complexity for an interesting learning process.

4.3.3 Character/Vehicle

Out of all characters and vehicles in Mariokart Wii, one character-vehicle combination is by far the most widely used, that being Funky Kong on the Flame Runner bike. (8.2).This is because of its high values in the 3 most important characteristics (speed, weight and miniturbo). Choosing this allows for the highest degree of comparison between the agent and human players by reducing the external factors on lap time.

4.3.4 Items

In a standard 'Time Trial' race, the player is given 3 mushroom items to use. These provide a large speed boost to the kart. I decided to remove the ability to use these in order to help simplify the state and action spaces.

4.4 Parameter Tuning

The three parameters to Q-Learning ϵ, α and γ have a great effect on the learning process, so to tune them I researched appropriate ranges through literature and then conducted a series of short experimental runs to tune them further. After approximately 250 episodes, I observed both the agent's progress around the track, and Q-values in various states. Tuning ϵ was a rather simple process: Literature such as Sudharsan Ravichandaran's book *Hands-on reinforcement*

learning with Python (2018) [25] suggests a low epsilon (< 0.01) due to the fact that it is used at each time step, repeatedly increasing the probability that a random action has occurred during the episode. In my observations, I found the agent performed better on average with a low epsilon, this was completely as expected as with a lower value, the best action is chosen more often.
For my problem, using a high γ (> 0.9) gave me the best results, due to the ultimate long-term goal of completing a lap and the relative simplicity of my reward function.
When tuning α I found that the instability introduced by a high value was sometimes useful when states had been fully explored but the best action was not being exploited, however it also meant that the agent would not converge to an optimal policy. Ultimately, I found a learning rate of 0.6 to give best results.

4.5 Data Collection

During the training process I collected 2 key pieces of data at the end of each episode. The total reward earned and the sequence of controller inputs. This data allows me to plot the progress of the agent during the training process, and keep a record of how the agent achieved that progress. Keeping the controller inputs also allows for playing against the agent after different amounts of training time, allowing for direct comparison to human performance.

CHAPTER 5

Design - Deep-RL

5.1 Formulation

For the Deep-Learning implementation, I needed to adapt my formulation slightly. The reward function and action space remained the same, but the state space instead consisted of pixel data. At this stage I did consider including the information from my Q-learning state representation as well as the pixel data, but I chose not to as keeping them separate would give a fairer comparison. Accessing the pixel data was the main challenge of this implementation, as there was no direct programmatic access through the API. Ben Middleton's implementation [7] used an older version of Dolphin which did have API access to the pixel data, however this feature has since been removed due to bugs. For my implementation, I made use of the frame-dump functionality built in to Dolphin. This feature causes Dolphin's emulation to slow down significantly, but was the only method I had to access the required pixel data.

5.2 Rainbow Agent

The Rainbow Agent implements a combination of improvements to DQN. Many of these are complicated and require lots of background knowledge outside the scope of this paper. Regardless, here is a short summary of each:

1. **Double Q-learning**[33] addresses the overestimation in the maximisation step ($\max_{a'} Q(s', a')$) of the Q-learning update rule, in some stochastic environments. Double Q-learning maintains 2 Q-tables, each taking the maximising value from the other, giving an unbiased estimate of the best value in the next state. In DQN, the *online* and *target* networks are used for this purpose.

2. **Prioritized Experience Replay**[28] is a variation of Experience Replay[19]

which prioritises transitions[1] from which an agent can learn more efficiently from. The higher priority a transition has, the higher likelihood it will be replayed, allowing the agent to learn efficiently.

3. **Duelling Networks** [35] is a D-RL architecture which splits a Q network into its separate components, giving a value network, which estimates the value of states, and an advantage network, which estimates the value of actions. These are then combined in an aggregating layer, giving a final value. When combined with double Q-learning, this gives a *target* and *online* network for both the value and advantage components.

4. **Multi-step learning** considers the rewards earned n steps into the future during the maximisation step in Q-learning. Allowing the agent to consider the further future effects of the action. This helps to deal with sparse or long term rewards, that would previously take many iterations to have an effect on the update. Sutton and Barto (2018) [29] found that a well-tuned n lead to faster training.

5. **Distributional RL** learns to approximate the distribution of returns, rather than the expected return. It does this by minimising the Kullbeck-Liebler divergence between the current distribution and a newly-constructed target distribution. Bellemere *et al.* (2017)[4] found that it surpassed the performance of other approaches at the time, when applied to the ALE.

6. **Noisy Nets** (Fortunato *et al.* (2017)) [15] introduces additional exploration capabilities by including a noisy layer. Over time, the network learns to ignore the noise at different rates through the search space. Allowing for different levels of exploration depending on the current state.

5.3 Implementation

To effectively implement the rainbow agent, I designed this architecture.

- Using the time step of the emulation as a reference, the corresponding frame is read from the framedumps directory,
- The frame is pre-processed and sent as input to Rainbow, along with the previous frame's reward,
- Rainbow selects an action epsilon-greedily,
- The action is performed using the *Controller* endpoint,
- Calculate the reward earned by performing the action.

Due to the limitations of Dolphin's embedded Python interpreter, the rainbow agent was run in a separate terminal. To allow for the required information to be sent between the two, I used a local socket bound to a specific port. This allowed use of all required external libraries for both rainbow and the image processing. [2]

[1] In this case a state s_{t-1}, an action a_{t-1}, a reward R_t, and a next state s_t

[2] This method would be an appropriate workaround for the issue I encountered with Tesseract OCR, however by this stage in the project I had fully implemented the memory access functionality, which I chose to continue using as to not discard my work.

5.3.1 Input Processing

Pixel data is given as input to Rainbow, which is fed into a CNN and then the Rainbow DQN. To improve the efficiency of the CNN this input data is preprocessed through cropping to only the middle part of the screen (containing the kart and the track), down-sampling to 84x84 resolution and grey-scaling. This preprocessing is essential in providing an input of the correct shape and with minimal size, while maintaining required data.

(a) Screenshot of game before preprocessing (1080p, 263kb)

(b) Screenshot of game after preprocessing (84x84, 4kb)

Figure 5.1: Screenshot of game before and after processing

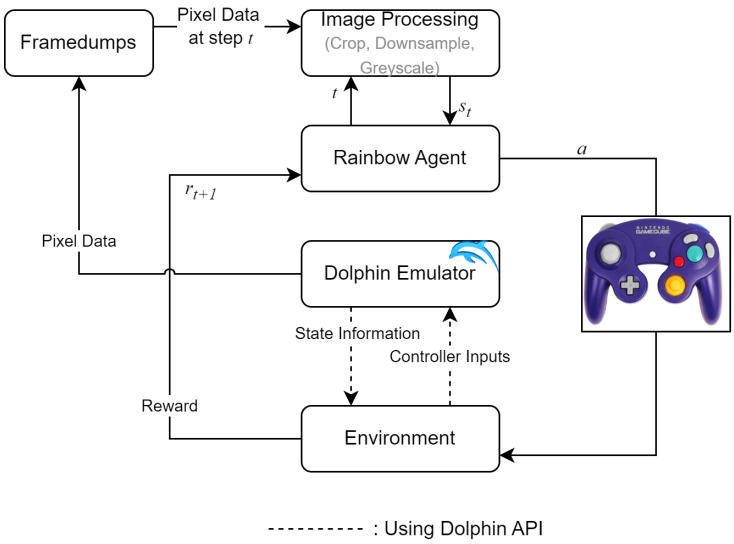


Figure 5.2: D-RL system architecture

CHAPTER 6

Results

6.1 Q-Learning

6.1.1 Problem Instances

In very early iterations, the agent would exploit the MT component of my reward function, leading to the agent getting stuck in a local optima. It did this by drifting to the left, charging a miniturbo and releasing it into the wall, ending the episode. This took advantage of the overly-weighted MT component and my termination conditions, which at the time only considered the agent's speed. This oversight allowed the agent to stay in the offroad area for just long enough to charge the miniturbo, granting it a large reward. To combat this I greatly decreased the weighting of the MT reward component, and introduced the road type check into the termination conditions. These two additions, along with a slight adjustment to γ prevented the agent from exploiting this any further.

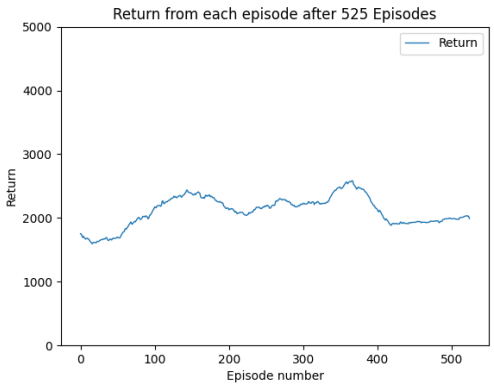


Figure 6.1: Graph showing agent stuck in local optima

6.1.2 Experimental Runs

As mentioned earlier, I conducted a series of experimental runs to find the optimal representation of the karts position in my state space. These runs consisted of adjusting the accuracy of the XZ co-ordinates to three different levels of accuracy. I trained each agent for 5000 episodes, changing only the rounding procedure of the state space. From the three trained agents I found that agent B, which was accurate to nearest Integer, to be the most effective. Analysing the training results, it is interesting to see that agents A and C got stuck in the same local optima, implying similar problems between the two. From monitoring the reward value through a single lap, I found that this occurred at the 'hairpin' turn[1]. This local optima makes sense as it is a difficult part of the track, requiring the agent to come into the corner on the right hand side of the track and hold a drift left the entire way through. Another factor that makes this corner more difficult for the agent is the fact that the track widens around this corner. While this is designed to make it easier, it led to many more states being explored, as going straight on (i.e. not taking the corner) took longer to terminate, leading to more time steps spent exploring actions that all eventually lead to terminal states. This was most likely due to the fact that the agent was unable to detect upcoming track borders. Finding a way to do this could greatly help the agent to navigate difficult corners such as these.
Agent B completed its first lap after 2491 episodes and only improved its time by .2 seconds after the total 5000 episodes. This is most likely due to the absence of a lap-time focused reward function. With the current implementation, completing a lap is rewarded the same whether it is a quick or slow lap, possibly meaning that a fast lap 1 was missed due to earlier termination in lap 2 than a slower lap 1.
Click here to watch agent B drive around the track (YouTube)
Note: Lap timer starts at 47 seconds

6.2 Deep Learning

Unfortunately, when running the D-RL algorithm, I repeatedly encountered an unknown issue. The emulation and the rainbow agent terminal would freeze for seemingly no reason. I investigated the RAM, CPU and GPU usage during this time but noticed no issues, therefore leading me to believe that there was a deadlock within Dolphin or my implementation of Rainbow. Due to the unknown factor of this issue and time restraints, I had to unfortunately abandon this part of my project. However, to still allow for comparison between the approaches, I used existing results from a trained agent [8]. This agent is trained on a different track however, so to compare between tracks I used the agent's lap time as a percentage of the current human world record (WR) time on that track[2]. This allows us to see how close the agent was to 'optimal' performance.

[1] Where the track turns sharply back on itself, almost 180 degrees (4.5)
[2] As of 31st March 2023

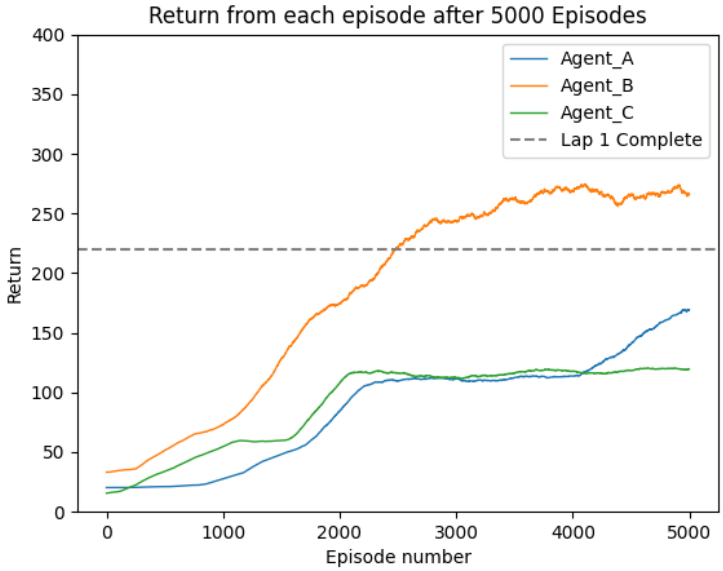


Figure 6.2: Graph of Agents' average reward during training, smoothed using a moving average

6.3 Comparative Analysis

6.3.1 Human Data Collection

In order to compare with human players, I collected race and lap time data from a range of abilities. As the participants were unfamiliar with the controls and the track, I gave each of them 3 attempts at a 3-lap time trial, taking their best result. After separating the results into 3 ability levels (beginner, intermediate and advanced), I took the average lap times and race times to give 3 clear performance bands.

Player	**Race Time (m:s)**	**Lap Time (s)**	**% of WR**
Q-Learning (Agent B)	1:52.5	37.5	145%
Rainbow	1:33.91	31.3	136%
Human - Advanced	1:40.58	33.52	129%
Human - Intermediate	1:52.57	37.52	144%
Human - Beginner	2:02.54	40.84	157%

Table 6.1: Human and RL results

6.4 Analysis

6.4.1 Comparison between Approaches

Both the Q-learning and rainbow agents trained for approximately 12-15 hours and achieved relatively similar lap times[3] This suggests that their training speeds for this application are very similar, however that is not completely true. The rainbow agent was able to complete 3 separate laps of the track, however the Q-learning agent was only able to complete 1.5 laps.

6.4.2 Hardware Limitations

While Rainbow performed somewhat comparably to Q-learning after a much shorter training time, it is important to keep in mind the computational resources that each takes. My laptop was able to run the Q-learning algorithm alongside Dolphin at 250% of the normal game speed (approx. 150 fps), allowing for lots of episodes in a short time. When running DQN however, the emulation slowed to around 1.6% game speed (approx. 1fps). These differences outline the hardware limitations of DQN and how Q-learning is a more accessible globally due to the lower processing requirements. An application of this could take the form of an interactive Q-learning demonstration, used in an educational context as an introduction to RL.

6.4.3 Scalability

If I were to increase the scope of this project, to potentially include other tracks, vehicles etc., the size of my state space, and therefore Q-table would increase, potentially becoming intractable. In comparison, the Deep-RL approach would not encounter this same issue, as the size of the neural network doesn't change during execution, only the parameters.

[3] In comparison to the WR

CHAPTER 7

Discussion and Limitations

7.1 Environment

7.1.1 Track

The environment that I chose was designed with simplicity and comparison in mind. However, this heavily limited the scope of my project. While this led to an effective agent on one specific track, its performance on other tracks which require vastly different input sequences would be comparable to random action selection. To counteract this, I could include the current track in the state, therefore allowing for training on multiple tracks, however doing so would increase the state space size by a factor of 32, the total number of tracks in the game.

7.1.2 Character and Vehicle

The same can be said for the character and vehicle selection. It would be interesting to see if the agent trains quicker on an easier vehicle, or how two vehicles compare after a specified training time. This adjustment would, however, require the action space to be broadened slightly, as different karts have marginally different turning characteristics.

7.1.3 Items

The use of items in Mariokart Wii can greatly benefit the user, while also having the potential to greatly disadvantage them. I decided to remove the ability to use these, as designing a reward function that encourages correct usage in the correct situations proved to be very complex. The introduction of items into the state space could lead to interesting situations and learning possibilities as the agent adapts its strategy, but also has the possibility of exploiting the reward function, using all items as soon as possible.

7.1.4 State Representation

The game displays a minimap on the right hand side of the screen, which shows the characters position and rotation on the track. Extracting the current rotation of the kart (in a similar way to Jack Boynton (2022)) could help the agent during the training process by providing it more information about the environment. The minimap, however is unfortunately not very precise, and extracting this information directly from the minimap would require additional image processing. Therefore, if I were to include the rotation of the kart in the state space, it would be more computationally efficient to read the value from memory.

Figure 7.1: On screen minimap

7.2 Q-Learning Implementation

7.2.1 Training Approach

My Q-Learning implementations took approximately 15 hours to complete 5000 episodes, which can be greatly decreased by making a few changes to the training process. Firstly I ran the training process sequentially, with only one instance of Dolphin running at a time. This could be greatly improved by implementing parallel computation, allowing for multiple agents to run simultaneously. Additionally, Dolphin can be configured to run in 'headless' mode, where the game isn't displayed on the screen, but is still run in the background. This would decrease the computational resources required for each instance, increasing the parallel capacity.

7.2.2 Initial States

During training, the agent visits the states towards the start of the track many more times than those further along, meaning they receive more updates and therefore a better optimal policy function approximation. To counteract this, I could define a set of start states, spread equally across the track that are chosen from uniformly upon termination. This would allow the agent to learn different areas of the track at the same rate.

7.2.3 Action Space

In my implementation I reduced the action space greatly, from around 40 total controller permutations down to 8. While this produced good results, the agent used the normal 'turning' inputs a lot more than I expected. Performing these actions reduces the agents speed more than if it used a drift to adjust its direction, suggesting that the agent required inputs that allowed for a small adjustment in direction. In a revised state space, I would remove these turning inputs and instead include more angles of drifting, perhaps 7 or 9 different analog stick angles, allowing the agent to make these slight adjustments.

TAS Inputs

Members of the TAS community have discovered many advanced input combinations that exploit the game's physics engine to achieve higher speeds than normally possible. These techniques are not humanly possible, requiring many sequential frame-perfect inputs, but could easily be performed by a script. One example of these techniques, known as *superhopping*, is executed by sequentially performing a series of drifts. When starting a drift, the kart 'hops' into the air and rotates slightly, if the drift is cancelled midair and another is started the same frame the kart hits the ground, then the kart maintains its current speed, instead of slowing down. A community member who goes by the alias 'Monster' demonstrated the potential of this, creating a short TAS which takes a long turn with lots of speed [22]. To extend and enhance the action space, techniques such as this could be formalised as actions with parameters of *direction* and *duration*, allowing for further exploration of the many existing techniques. One issue with this, however, is that many of these techniques rely on many sub-optimal actions before the technique is executed. This could be tackled by redefining the reward function to give sparse rewards, but this in turn has its own problems. The most likely application for this would be exploration or proof-of-concept, rather than a refined TAS.

7.2.4 Action Selection Policy

The ϵ-greedy policy is a state-independent policy, meaning that the current state doesn't affect the action selection. However, more actions are explored in a given state, it becomes increasingly likely that the optimal action has been found. Therefore we should decrease the likelihood of exploration in this state. This is also true for vice versa. Considering this, an alteration to the policy could be made that adjusts the epsilon value based on the proportion of unexplored actions, encouraging exploration when fewer actions are explored and exploitation when more actions are explored. This could lead to a more efficient balance of exploration and exploitation.

Random Action Weighting

When split into steps of 20 frames, more often than not the best action is the same as or similar to the current action being performed, such as a slight adjustment to the steering during a corner or holding down accelerate through a straight. To encourage these similar actions to be performed more often, I could

weight the available actions and define 'neighbour' actions. The 'neighbour' actions would have higher weights, such that when an action is randomly selected they have a higher probability of being chosen. This would decrease the time taken to explore the optimal action in each state.

7.2.5 Reward Function

The reward function I designed gave a constant reward for completing a lap. However the objective is to complete the lap in the *quickest time possible*. The current implementation actually rewards slower laps more than quicker laps, as there are more episodes to gain a reward from. To mitigate this effect I can give a reward inversely proportional to the number of time steps at the time of completion, giving a higher reward for lower time steps taken to complete the lap. This would however give the same reward to any laps completed within the same 20 frame window. To accommodate this, I would need to redesign my system to check the completion on every frame.

7.3 Rainbow (DQN) Implementation

7.3.1 Input Data Size

The current implementation of Rainbow uses an 84x84 grey-scale image as input to the neural network, changing the size of this could lead to interesting discoveries. For example the minimum amount of pixel data needed, or which areas of the screen lead to the quickest training. For example, the minimap mentioned earlier 7.1 could be compared with the main area of the screen containing the track and character. Furthermore, decreasing the amount of pixel data slightly could lead to similar results whilst using less computational resources, but decreasing too much would undoubtedly lead to a loss of detail and therefore required information.

7.3.2 Ablation Comparisons

Mnih *et al.* (2018) [18] compare the full Rainbow agent with some ablations, removing one extension and performing the same tests. This helps to see the effect of each extension individually. Performing the same study in this application and comparing to the original results could be interesting, as different extensions may have different effects in this scenario, potentially highlighting their specific strengths.

7.4 Generalisation Capabilities

One of the main known advantages of Deep-RL over traditional RL is its ability to generalise. Testing and demonstrating this could be performed very easily in this application by simply changing the track. If I were to carry out this experiment, I would expect DQN to greatly outperform Q-learning at all stages. In addition to this, finding a situation where Q-learning is superior requires very specific circumstances. One example of this would be a track where the road and offroad are the same colour and texture, with no border between the two. This

would prove very tricky for DQN as the pixel data for very different situations would be near identical. In contrast, the values in memory used by Q-learning would not have the same problem, potentially leading to better performance.

CHAPTER 8

Conclusions

In this paper I have shown implementations of two different approaches to Reinforcement Learning in a simplified Mariokart Wii environment. After tuning and training I found that my Q-learning implementation achieved performance similar to an average human player, demonstrating the effectiveness of my formulation. My deep learning approach was unsuccessful, however I was still able to compare the two using data from an existing implementation. Overall, I found both approaches were equivalent to a human player with intermediate skill level after less than 20 hours of training, with the potential to surpass this level with a few minor adjustments and additional training time. Similarly to Mnih [21], I have shown that human level performance is possible through Reinforcement Learning. I then discussed some limitations with my project, such as its lack of generalisation to other tracks or vehicles and the inability to use items. Additionally, I proposed solutions to these limitations, mainly detailing various additions to the state space of my agent. Finally, I outlined a series of potential future investigations, such as including an enhanced action space with pre-programmed multi-step inputs.

Bibliography

[1] aldelaro5. Dolphin memory engine. `https://github.com/aldelaro5/Dolphin-memory-engine`, 2023. Licensed under MIT License. Accessed 15 Jan 2024.

[2] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

[3] M. Bellemare, J. Veness, and M. Bowling. Investigating contingency awareness using atari 2600 games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 864–871, 2012.

[4] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[5] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[6] R. Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[7] BenJMiddleton. Mkwii ai environment. `https://github.com/benjaminjmiddleton/mkw_ai_env/blob/main/README.md`, 2023. Accessed 7 Feb 2024.

[8] BenJMiddleton. Trained rainbow agent. `https://drive.google.com/file/d/1dJF1GY61CEN9w-zd9BitY7Be1eRwTIOh/view`, 2023. Accessed 7 Feb 2024.

[9] S. Block and F. Haack. esports: a new industry. In *SHS Web of Conferences*, volume 92, page 04002. EDP Sciences, 2021.

[10] M. Boettinger and D. Klotz. Mastering nordschleife–a comprehensive race simulation for ai strategy decision-making in motorsports. *arXiv preprint arXiv:2306.16088*, 2023.

[11] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016.

[12] J. Boynton. Reinforcement learning v. mario kart wii. `https://github.com/JackWBoynton/mariokart-rl`, 2022. Accessed 13 Jan 2024.

[13] J. Duan, S. Eben Li, Y. Guan, Q. Sun, and B. Cheng. Hierarchical reinforcement learning for self-driving decision-making without reliance on labelled driving data. *IET Intelligent Transport Systems*, 14(5):297–305, 2020.

[14] Evan-Amos. Gamecube controller. Digital Image (public domain) `https://en.m.wikipedia.org/wiki/File:Gamecube-controller-breakdown.jpg`, 2010.

[15] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017.

[16] Google. Tesseract OCR. `https://github.com/tesseract-ocr/tesseract`, Year.

[17] Henrik Rydgård and The Dolphin Emulator Project. Dolphin emulator. `https://github.com/dolphin-emu/dolphin`, 2003. Licensed under GPL-2.0-or-later. Accessed 16 Oct 2023.

[18] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[19] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[22] monster (@monster3rum). [mkwii tas] new faster movement with outside drifting bike (superhopping). `https://www.youtube.com/watch?v=CS_1klWMybI`, May 2022. Accessed 26 March 2024.

[23] Nintendo. Wii controllers. Digital Image `https://www.nintendo.co.uk/Hardware/Nintendo-History/Wii/Wii-636022.html`, 2013.

[24] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[25] S. Ravichandiran. *Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and tensorFlow*. Packt Publishing Ltd, 2018.

[26] A. Remonda, S. Krebs, E. Veas, G. Luzhnica, and R. Kern. Formula rl: Deep reinforcement learning for autonomous racing using telemetry data. *arXiv preprint arXiv:2104.11106*, 2021.

[27] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[28] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[29] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] TASLabz. Mkwii ai environment. `https://github.com/TASLabz/dolphin`, 2023. Licensed under GPL-2.0-or-later. Accessed 23 Oct 2023.

[31] unknown (no information provided). Yoshi has a blue spiny shell. Digital Image `https://themushroomkingdom.net/images/ss/mkwii/045.jpg`, 2008.

[32] unknown (reddit account deleted). Dolphin emulator logo. Digital Image `https://www.reddit.com/media?url=https%3A%2F%2Fi.redd.it%2F0g3e70r2ovn21.png`, 2019.

[33] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[34] K. Wade474, CampbellMop and _tZ. Gecko codes for mario kart wii tasing. `https://sites.google.com/view/mkwtastools/resources?authuser=0`, 2023. Accessed 29 November 2023.

[35] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[36] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[37] D. Ye, Z. Liu, M. Sun, B. Shi, P. Zhao, H. Wu, H. Yu, S. Yang, X. Wu, Q. Guo, et al. Mastering complex control in moba games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34:04, pages 6672–6679, 2020.

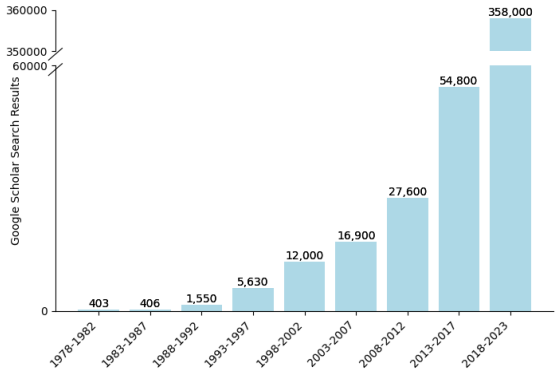


Figure 8.1: Graph of Reinforcement Learning Publications (Google Scholar)

Name	**Location**	**Type (in Memory)**	**Range**
Speed	`0x80E4C678`	32-bit Float	[0, 113.4]
Completion	`0x80E43708`	32-bit Float	[0, 4]
X Position	`0x80E4DAE8`	32-bit Float	[-150,000, 150,000]
Z Position	`0x80E4DAF0`	32-bit Float	[-150,000, 150,000]
MT Charge	`0x80E4C756`	16-bit Integer	{0,270}
Wheelie State	`0x90284F04`	16-bit Integer	{0,1}
Road Type	`0x80E51CE8`	16-bit Integer	{0,7}

Table 8.1: Memory Addresses in Mario Kart Wii's emulated memory

Participant	Lap 1	Lap 2	Lap 3	Best Lap	Average Lap	Total Time	Skill level
1	51.091	36.479	34.97	34.97	40.84666667	122.54	Beginner
2	35.173	37.891	34.5	34.5	35.85466667	107.564	Intermediate
3	39.118	39.02	33.879	33.879	37.339	112.017	Intermediate
4	39.865	36.329	36.897	36.329	37.697	113.091	Intermediate
5	37.399	36.807	40.156	36.807	38.12066667	114.362	Intermediate
6	39.762	37.948	38.109	37.948	38.60633333	115.819	Intermediate
7	34.615	31.155	31.025	31.025	32.265	96.795	Advanced
8	33.498	34.121	31.636	31.636	33.085	99.255	Advanced
9	35.39	33.653	31.655	31.655	33.566	100.698	Advanced
10	35.798	34.922	31.908	31.908	34.20933333	102.628	Advanced
11	38.293	32.747	32.519	32.519	34.51966667	103.559	Advanced

Table 8.2: Lap time data, in seconds, from human participants

Figure 8.2: Screenshot of Funky Kong on the Flame Runner