Playing Mario Kart 64 with Deep Reinforcement Learning

Young James Yang
The Hong Kong University of Science and Technology
jyyoungaa@connect.ust.hk

Abstract

In recent years, there has been a significant surge in the capabilities of artificial intelligence systems designed for playing games. With the advent of deep learning algorithms and various classes of reinforcement learning algorithms, many tasks such as beating Atari games have become achievable. These AI algorithms can learn to play games effectively through experience, even with minimal prior knowledge about the specific domain they are operating in. This project attempts to develop a deep reinforcement learning agent that can finish a track in the popular racing game Mario Kart 64. The project uses Deep Q-Networks, which combines convolutional neural networks and Q-learning by using a convolutional neural network as a function approximator to predict the optimal action to take in a given state [9]. The convolutional neural network takes the game screen raw pixels and outputs estimates of the Q-values for different actions to predict the best action to take. The reward function uses the speed and progress of the kart extracted from the game's memory. The findings of the project result finds that the agent is incapable of completing the track Luigi Raceway.

1. Introduction

The application of artificial intelligence and machine learning techniques to play video games has emerged as a captivating area of research, showcasing their potential in mastering complex game environments. One widely used machine learning method to play games is through deep reinforcement learning, which combines reinforcement learning and deep learning to train agents to play games. Deep reinforcement learning in video games has made significant progress in recent years, where it has been successfully applied to play games such as Doom [10], various Atari games [9], Starcraft [13] and so on.

This project focuses on using deep reinforcement learning to play Mario Kart 64. Mario Kart 64 is a popular go-kart racing game developed and published by Nintendo for the Nintendo 64 (N64). The game introduces complex environments with various obstacles and terrains to navigate through, presenting a unique and interesting opportunity to develop and test a real-time autopilot agent.

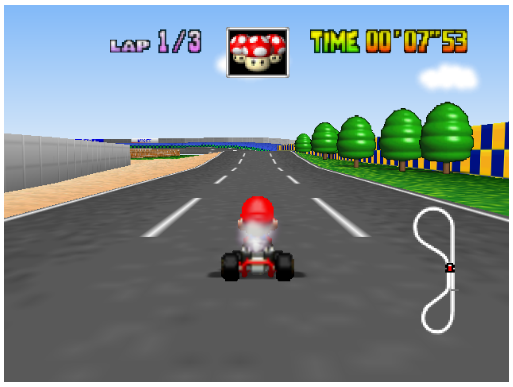


Figure 1. Mario Kart Agent on Luigi Raceway

Specifically, the problem the project aims to tackle is to design and train an autonomous game playing agent that is able to navigate and complete a course as fast as possible. To simplify the environment, the game is played in time-trial mode, where the player's objective is to finish a set number of laps of a track as fast as possible. Further simplification includes removing the option to drift and use items. The agent is trained and evaluated on the the course Luigi Raceway.

The project uses Deep Q-Networks (DQN) to approach this problem. The project uses real-time screenshot images of the game and data from the game's memory to extract the kart's speed, laps completed, and progress. Then, the images are fed to a convolutional neural network (CNN) to predict the best action to take. The reward function is designed based on the kart's speed, laps completed, and progress.

However, the approach presented in this paper was unsuccessful in tackling this problem. The agent was unsuccessful in completing a single lap and would fail after about a quarter way through the track.

2. Related Works

Using machine learning to play video games has been studied extensively in the past few years. Two popular methods include imitation learning and deep reinforcement learning.

2.1. Imitation Learning

In imitation learning, an agent learns to imitate the behavior of an expert by observing their actions and generalizing from the provided demonstrations. By imitating the demonstrated behavior, the agent can quickly acquire competent policies without the need for trial-and-error exploration. However, one of the key limitation of imitation learning is the distribution shift problem. Distributional shift occurs when there is dissimilarity between the data distribution during training and the distribution encountered during testing or deployment. In practice, this means that the model may struggle to recover when multiple errors occur if the expert data is too good [16]. Futhermore, imitation learning relies heavily on the availability of expert demonstrations, which may be costly or difficult to obtain for complex tasks.

2.2. Deep Reinforcement Learning

Deep reinforcement learning integrates deep learning and reinforcement learning techniques. It involves training deep neural networks to autonomously learn and make decisions in complex environments, such as video games. By leveraging deep neural networks as function approximators, the agent can capture intricate patterns and adapt its behavior through iterative interactions with the environment, aiming to maximize cumulative rewards [3].

Since the AI in deep reinforcement learning will learn through trial and error, this means that compared to imitation learning, it is less prone to the distributional shift issue and requires no human or AI generated dataset. However, because of this, they are also more complicated and take longer to converge [7].

Several deep reinforcement learning algorithms have been proposed in the past few years. One such algorithm is DQN, which was developed to play Atari games. DQN combines deep neural networks with Q-learning, where it learns to estimate action values based on game states and rewards and uses an experience replay buffer to sample and learn from past experiences [9]. This project mainly uses this model as reference to approach to play Mario Kart 64.

2.3. Related Works in Mario Kart 64

Much of the work using CNNs to play Mario Kart 64 uses some form of imitation learning. TensorKart uses screenshots to human inputs to train their autopilot agent and is able to generalize training data to several track scenarios [6]. Their CNN model uses a modified version of NVIDIA's autopilot model from their paper in 2016 [1]. However, TensorKart suffers from the distributional shift limitation of imitation learning, where TensorKart's model does not recover well when the kart is placed in error conditions.

NeuralKart is another project that uses imitation learning and builds on top of TensorKart. Their project uses the DAGGER imitation learning algorithm, which seeks to resolve the distributional shift issue [16]. Instead of using human inputs, NeuralKart uses an omniscient search AI to simulate different possible actions to generate training data. However, one of the limitations they found was that the autopilot would drive near hazards such as walls or sand, and in their paper they suggested that using reinforcement learning could help push the AI towards safer states [5].

3. Data

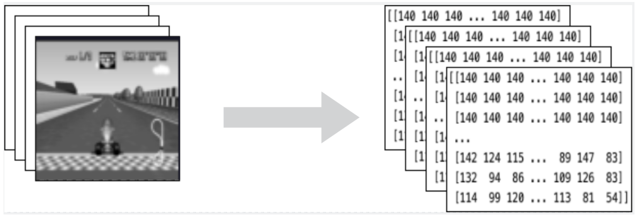
3.1. Emulation and Gameplay

To automate the races in Mario Kart 64, the N64 emulator Mupen64Plus will be used which will allow for saving and loading states as well as checking in-game memory for game information. To simplify the learning environment, the project will use Farama's Gymnasium framework (fork of OpenAI's Gym), which provides a library for developing uniform reinforcement learning interface [11]. Although there is currently a environment wrapper for Mupen64Plus called gym-mupen64plus, it lacks the option to read the game's memory [17]. Therefore this project uses its own custom environment Gymnasium environment for Mupen64Plus instead.

All the recorded games take place in Luigi Raceway, as most tracks are highly different and therefore I wanted to focus on playing well on one track. Luigi Raceway was chosen as the track has well-defined walls which mitigates the issue of the agent driving or falling off track, which will help speed up training.

3.2. Data Collection and Image Preprocessing

For training the reinforcement learning agent, the project records the game's screen raw pixels for each frame, which is then stored in an experience replay buffer. The game is run at 640x480 pixels with RGB, which the model then resize to 84x84 pixels and is grayscaled. To provide the agent with better temporal information and capture motion cues, 4 consecutive frames are stacked, meaning that the input to the CNN is a 84x84x4 tensor as shown in Figure 2. These processes and parameters chosen were based off of Playing Atari with Deep Reinforcement Learning paper [9].



```
[[140 140 140 ... 140 140 140]
 [140 140 140 ... 140 140 140]
 [140 140 140 ... 140 140 140]
 ...
 [142 124 115 ...  89 147  83]
 [132  94  86 ... 109 126  83]
 [114  99 120 ... 113  81  54]]
```

Figure 2. Example of 4 frames downsampled to 84x84 and stacked which are used as input to CNN

Alongside the game's screen frames, the project also extracts certain game information from the game's memory. Mainly, the project extracts the kart's current progress, speed, and current lap which are stored as 32-bit integers. The current lap ranges from 0 to 4, where 0 indicates the car behind the start-line and 4 indicates finishing of the track. The kart's speed without using boosts can range anywhere from 0 to 1.9×10^{10} but as long as the kart is moving the speed will stay between 1×10^{10} and 1.9×10^{10}. The progress from the start to end of the track ranges from 0 to 1.08×10^{10}. These information are used in the reward function of the DQN.

4. Methodology

4.1. DQN Algorithm

Q-learning is a foundational reinforcement learning algorithm extensively employed for solving Markov Decision Processes (MDPs). The fundamental objective of Q-learning is to acquire an optimal action-value function, known as the Q-function, which encapsulates the expected cumulative rewards associated with taking specific actions in particular states [14].

The Q-learning algorithm updates the Q-values, denoted by $(Q^*(s, a))$, which represent the expected cumulative rewards for taking action (a) in state (s). The Q-values are iteratively updated using the following equation:

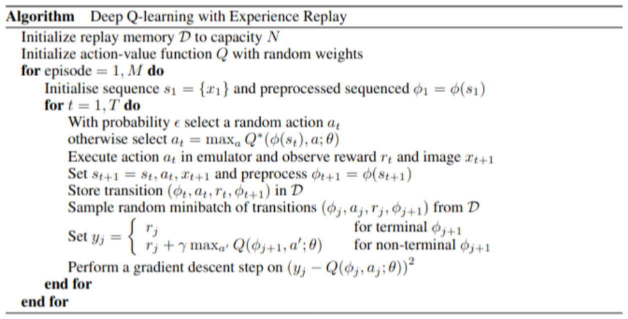
$$Q^*(s, a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s', a')] \tag{1}$$

where (α) is the learning rate, (r) is the immediate reward obtained by the agent, (γ) is the discount factor that determines the importance of future rewards, (s') is the next state, and (a') is the optimal action in the next state. This update equation allows the agent to learn from its experiences and gradually converge to an optimal policy that maximizes long-term rewards [14].

DQN is an extension of Q-learning that leverages deep neural networks to approximate the Q-values, enabling learning in high-dimensional state spaces. The DQN algorithm employs a deep neural network with parameters (θ) to approximate the Q-values. The network is trained by minimizing the mean squared error loss between the predicted Q-values and the target Q-values, given by the Bellman equation:

$$L(\theta) = \mathbb{E}s, a, r, s' \left[\left(r + \gamma \max a' Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right] \tag{2}$$

where (r) is the immediate reward obtained by the agent, (γ) is the discount factor, (s') is the next state, (a') is the optimal action in the next state, and (θ^-) represents the target network parameters. The network is updated using gradient descent to minimize the loss, allowing it to approximate the optimal Q-values and guide the agent's decision-making. Through this process, DQN enables agents to learn optimal policies in complex and high-dimensional environments [9]. Figure 3 shows an example overview of applying this algorithm.



```
Algorithm  Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
      With probability $\epsilon$ select a random action $a_t$
      otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
      Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
      Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
      Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
      Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
  end for
```

Figure 3. DQN Algorithm

Therefore, due to DQN's ability to bridge the gap between high-dimensional visual inputs and actions alongside its utilization of a replay buffer for efficient learning from past experiences, I decided to use DQN to play Mario Kart 64.

4.2. State, Actions, Reward

To utilise DQN, all sequences in Mario Kart 64 are assumed to terminate in a finite number of time-steps. With this, Mario Kart 64 can be modelled to a large but finite Markov decision process (MDP), where each sequence is a distinct state [9].

While Mario Kart 64 uses an analog controller to steer the kart in different directions, DQN only works with dis-

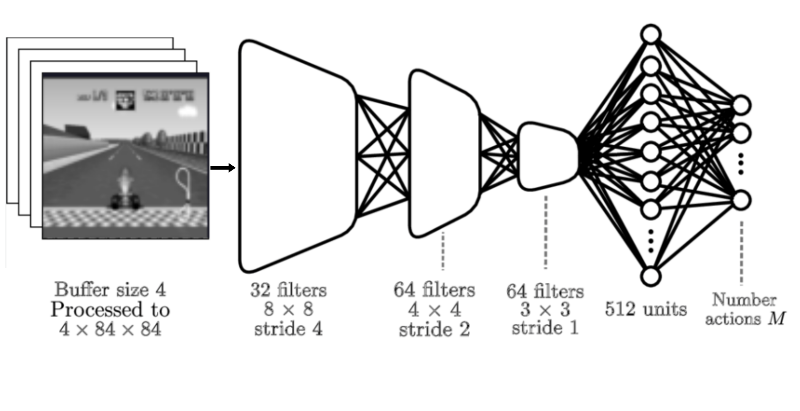


Figure 4. Overview of CNN model

crete actions. Therefore, the action space of the kart are discretized by only allowing the following 3 discrete actions:

- Action 0: Move kart forward-left
- Action 1: Move kart forward-right
- Action 2: Move kart straight forward

These three actions alone are enough to complete a race. Other abilities such as using items, drifting, and moving in more controlled directions are ignored to simplify the problem.

Due to the difficulty of extracting certain game information such as laps completed purely with the game screen, the project uses in-game memory to extract the speed, progress, and current lap of the kart and uses them to design the reward function. The reward system is listed below, where the speed of the kart is denoted as s and the good speed threshold is denoted as t:

- Finishing the race: +100
- Completing a lap: +50
- Making new progress: +($\frac{10*s}{t}$)
- Moving wrong direction (negative progress): -($\frac{5*s}{t}$)
- Moving too slow (ie: $s < t$): -1
- No new progress after 20 frames: -50 and reset game

As written above, we terminate the game early if no new progress is made after 20 frames to avoid situations where the kart gets stuck driving backwards or is stuck on some obstacle for too long.

4.3. CNN Model

The CNN architecture in this project follows the CNN model from the Playing Atari with Deep Reinforcement Learning paper [9]. As mentioned in the previous sections, the input uses 4 consecutive re-scaled and gray-scaled game frames to create a 84x84x4 tensor. Then, the first convolution layer convolves the input with 32 8x8 filters with a stride of 4. The second convolution layer convolves 64 4x4 filters of stride 2 and the third convolution layer has 64 3x3 filters of stride 1. Each convolution layer has a Rectifier Linear Units (ReLu) function between them. The final layer is a fully connected linear layer of 512 units that outputs the Q-values of each valid actions the agent can choose, which in this case is 3. Figure 3 shows a diagram of the CNN architecture described previously. An Adam optimizer is used to train the network and a checkpoint of the model is saved every 10,000 episodes.

5. Experiments

5.1. Hyper-parameters

When training the agent, the model uses a ϵ-greedy approach, where a random action is taken with probability ϵ. I set ϵ to a value of 1. After each episode, epsilon will be decreased by a discount factor of γ set to 0.9999 to limit exploration. The replay buffer size is set to 100,000, learning rate set to 1×10^{-4}, and batch size set to 32. Initially, training was done with a learning rate of 1×10^{-2}, but the kart seems to get stuck taking early sub-optimal actions where it would move forward then quickly turn left before the game would terminate. Therefore the learning rate was decreased to prevent this.

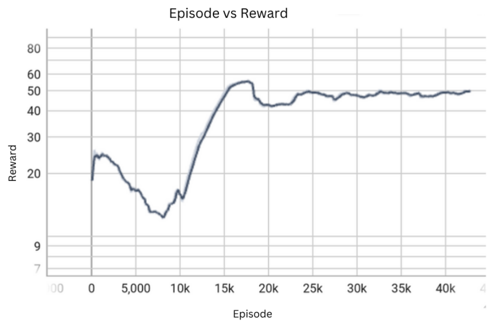


Figure 5. Reward accumulated during training

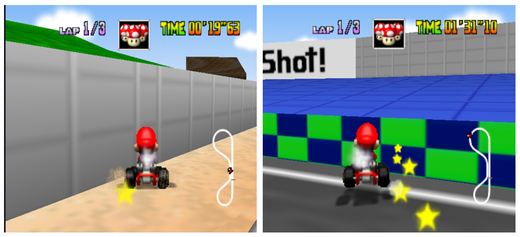
5.2. Results

Compared to supervised learning, evaluating the performance of the agent is a lot more difficult and challenging. Therefore, the evaluation metric used was a reward vs episode metric, where I recorded the total reward gained in each episode, where each episode is a game that ends in a terminal state (either completes the track or ends early due to no progress after 20 frames as described in the methodology section).

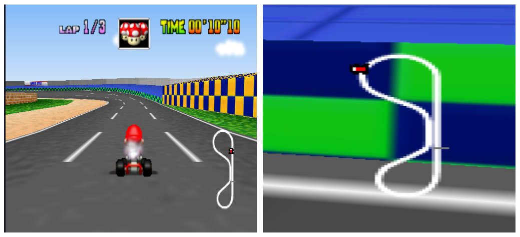
For training the agent, I ran 45,000 episodes totalling for around 12 hours on an Nvidia 2060. More episodes would have been desirable but unfeasible due to time constraints. The end result was that the evaluation of the best model after the training process could only reach a peak reward of 55, where the agent could only reach around a quarter through lap 1 before terminating early due to a lack of progress for a period of time.

Quantitatively, the agent begins learning and increases its rewards until around 15,000 episodes. The agent's reward peaks at around 55 at this episode range before the reward begins to plateau afterwards and the agent is unable to achieve better results. This is shown in the episode vs reward graph of Figure 5.

Qualitatively speaking, when observing the runs, it was found that during the first 5,000 episodes, the kart begins exploring different actions, such as driving forward then immediately turning left into a wall as shown in Figure 6a. After reaching around 15,000 episodes, the agent learns to drive straight forward in the first part of the track where there is a straight road, as shown in Figure 6b. However, during the first left-turn needed to continue progress through the track, the kart doesn't do enough left-turns to navigate through, where it often ends up in a loop of bumping into the wall continuously as shown in Figure 6c. Runs after episode 15,000 mostly repeat the same pattern, where the agent seems to repeat the same action over and over of driving forward, turning slight left before continuously crashing into the wall. Figure 6d shows the furthest progression on the track from the game's mini-map, where the agent finishes around a quarter of the track before terminating from a lack of progress as it continues to crash into the wall.



(a) Early episodes where kart explores (c) Kart starts repeating same action



(b) Kart learns to drive forward (d) Furthest progress on mini-map

5.3. Performance Factors

The failure of this model could be attributed to several factors. It is possible that simply more training was needed for the model to see better performance and perhaps even fully complete the track, as 50,000 episodes may have not been enough in hindsight to fully learn the track. However, as indicated in the results, the model seemed to have plateaued and therefore most likely would have taken extremely long to converge. Therefore, a better approach would be to continue tweaking the hyper-parameters. In particular, increasing the value of the discount factor γ may help with the model. As seen in the results section, the agent's reward seems to plateau after around 15,000 episodes. This may be due to the agent not having a high enough exploration rate to try different actions

Another point of failure could be the reward function. In this project, it was very costly and time consuming to score every state-action pair the kart visited. Therefore, using a trajectory-based reward system may be more suitable than purely rewarding speed and progress made by the kart. Trajectory-based reward systems are commonly used for autonomous driving tasks, where instead of providing a reward signal at each time step, a reward is given based on the cumulative effects of actions over a sequence of time steps [15]. For example, TMRL, a reinforcement framework for the racing game TrackMania, mentions that they use a trajectory-based reward instead of a speed-based reward as sharp corners may require the car to slow down and turn, hence a trajectory-based reward is a better solution for finishing a race [2]. Other reward tweaks such as adding checkpoints or punishing collision with obstacles could improve the model as well.

6. Conclusion

In summary, this project tried to use deep reinforcement learning to complete the Luigi Raceway track in the game Mario Kart 64. It used the algorithm DQN to predict the best action for the agent to take at each state by feeding the raw pixels of the screen into a CNN and using various game information to construct the reward function. Using DQN to play Mario Kart 64 ended up being an extremely challenging problem to tackle, and the project failed to achieve the baseline goal of completing the track. The agent manages to complete a quarter of the track before getting stuck and unable to finish turning the corner to make further progress on the race.

As mentioned in the experiment section, further improvements could be made to the model in the future, where adjustments to various tuning hyper-parameters including discount factor γ, learning rate, and other parameters may help improve the model performance. Also, trying different reward methods such as adding a trajectory-based reward similar to those in autonomous vehicles would most likely significantly boost performance of the model.

Outside of the tweaks mentioned above, another improvement that could be made is enhancing the DQN algorithm. For example, a modified version of DQN such as Double Deep Q-Networks (Double DQN) can be used instead of vanilla DQN. Double DQN improves on the original DQN algorithm by addressing overestimation bias. In DQN, the Q-values are estimated using a single neural network, which can lead to overestimation of action values, particularly in the presence of noisy or sparse rewards. Instead of using the same network for both action selection and evaluation, Double DQN uses the online network to select the best action and the target network to estimate its value. The target network is a periodically updated copy of the online network, which helps provide a more accurate estimation of the Q-values [12]. Other variants of DQN include algorithms such as Rainbow DQN [4].

Another consideration is using a completely different deep reinforcement algorithm that may be more suited to playing a more open environment like Mario Kart 64. For example, other algorithms such as asynchronous advantage actor-critic (A3C) may be more suitable to this problem [8].

References

[1] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016. 2

[2] Yann Bouteiller, Edouard Geze, and Andrej Gobe. Tmrl. https://github.com/trackmania-rl/tmrl, 2022. 6

[3] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3–4):219–354, 2018. 2

[4] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017. 6

[5] Harrison Ho, Varun Ramesh, and Eduardo Torres Montano. Neuralkart: A real-time mario kart 64 ai, 2017. 2

[6] Kevin Hughes. Tensorkart. https://github.com/kevinhughes27/TensorKart, 2017. 2

[7] Yuxi Li. Deep reinforcement learning, 2018. 2

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016. 6

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. 1, 2, 3, 4

[10] Georgios Papoudakis, Kyriakos C. Chatzidimitriou, and Pericles A. Mitkas. Deep reinforcement learning for doom using unsupervised auxiliary tasks, 2018. 1

[11] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, Mar. 2023. 2

[12] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. 6

[13] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning, 2017. 1

[14] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992. 3

[15] Tengyu Xu, Yue Wang, Shaofeng Zou, and Yingbin Liang. Provably efficient offline reinforcement learning with trajectory-wise reward, 2023. 5

[16] Boyuan Zheng, Sunny Verma, Jianlong Zhou, Ivor Tsang, and Fang Chen. Imitation learning: Progress, taxonomies and challenges, 2022. 2

[17] Brian Zier. gym-mupen64plus. `https://github.com/bzier/gym-mupen64plus`, 2016. 2