CM50270 Reinforcement Learning
Reinforcement Learning With The Super Mario Bros

Group 20
Department of Computer Science
University of Bath
Bath, BA2 7AY

1 Problem Definition

In this project, we aim to train reinforcement learning (RL) agents to play the game Super Mario Bros (SMB). In every stage of SMB, the player controls the actions of Mario with the aim to avoid enemies and obstacles, and finally reach the flag pole that marks the finish of a stage. To allow systematic analysis on the interaction between Mario and the OpenAI Gym SMB environment, we define the game as a Markov Decision Process (MDP), as illustrated in Figure 1. At each time-step, Mario chooses an action and observes the reward of the action as well as the resulted new state.

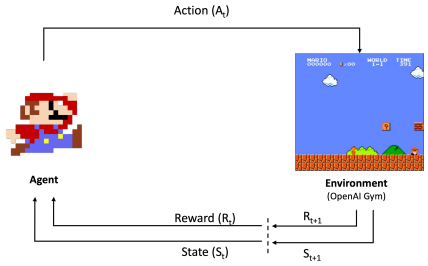

Figure 1: Interaction between Mario and the environment.

1.1 States

States in SMB are represented as frame pixels of game scenes. Combinations of the locations of the stage map, movements of Mario and monsters, as well as changes in numbers in the score bar would create new state representations. Resulting in an enormous state space that is prohibitive to compute using tabular methods. So, the feature selection method is necessary to allow effective learning of the optimal policy. In this project, we make use of the feature extraction capability of Convolutional Neural Network (CNN) to achieve such purpose.

1.2 Actions

In the game SMB, there are 12 possible actions. Due to limited training time, the project will only focus on the action set 'RIGHT_ONLY' and the custom action set of right and right + jump, as shown in appendix A. If more actions are given to Mario, it requires a larger amount of time to train. Mario has to use these actions to pass through the obstacles and reach the destination point.

Variables	Description
Δx	Change in agent's x position between steps; positive if agent has moved further right
Δt	Change in clock between steps
d	Death penalty: -15 if agent is dead; 0 otherwise

Table 1: Description of variables in the reward function.

1.3 Transition dynamics

Transition dynamics of the environment refers to the probability of transitioning from one state to another, these probabilities are unknown at the beginning of the game. As there are many possible states in SMB, using model-based algorithms that learn all of these transitions would imply extensive memory requirements. Therefore, we decide to use model-free Q-learning methods to train our agents.

1.4 Reward function

As the objective of the game is for Mario to finish the stage, reward is given when he moves as far right as possible, in the quickest time, without dying. The reward function comprises of three variables 1, which are listed in Table 1.

$$R = \Delta x + \Delta t + d \tag{1}$$

2 Background

Due to limited time and computing resources, we identified the following model-free algorithms that may be effective in solving SMB.

2.1 Deep Q-Network (DQN)

DQN leverages the structure of the off-policy Q-Learning. However, instead of storing and retrieving Q values in a tabular manner, DQN extracts features of states and estimates their corresponding Q values using a deep neural network (DNN) (Figure 2), which can handle the problem of high dimensionality state space in SMB. Unlike Value Function Approximation methods, the use of DNN in DQN reduces the manual effort and knowledge required for handcrafting discriminative feature vectors. One important feature of DQN is **Experience Replay**, which stores past state transitions in a buffer and samples a batch of experiences from the buffer instead of running Q-Learning algorithm with the latest experience to avoid Q values from diverging. When the weights have to be updated, an experience is randomly selected from the buffer to learn the off-policy. This prevents unstable training due to auto-correlation. We implemented three variants of DQN, which will be discussed in detail in section 3. A tutorial on PyTorch clears the first stage of SMB using one of the variants, with 40,000 as the suggested number of episodes [2].

2.2 Actor-Critic (AC) Methods

Another method suitable for our problem would be AC methods. It combines both value-based and policy gradient learning using two components, namely actor and critic. The critic estimates the value function of an action, while the actor updates the policy according to the feedback received from the critic. Advantage AC (A2C) improves on the vanilla AC by judging an action chosen by the actor in the critic's feedback [6]. Asynchronous Advantage AC (A3C) further extends on A2C by using multiple actors to explore the environment in parallel [6], so that the learning process can be made more efficient. We found a GitHub repository which cleared 19 stages of SMB using A3C agents [8]. However, due to the more complex structure and limited computational power, we decided against using AC methods.

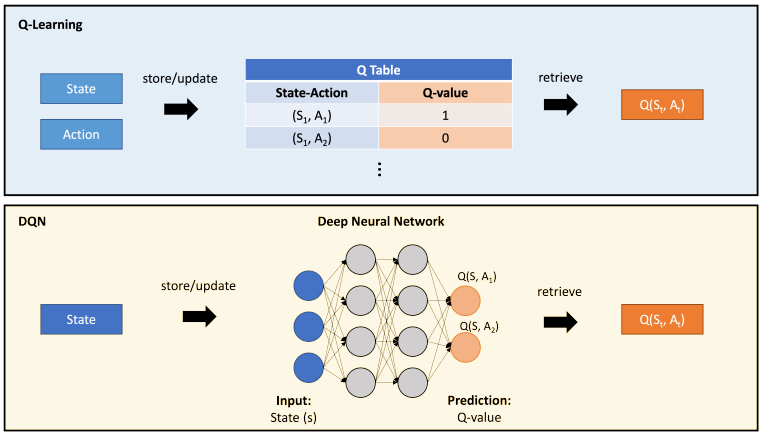

Figure 2: Comparison between Q-Learning and DQN. (Adapted from [1])

3 Method

3.1 Frames Preprocessing

Raw frames of the game are of size 240 x 256 x 3, which are computationally expensive to process. We referenced some of the frame preprocessing steps in DeepMind's Atari agents for downsizing the feature vectors, representing motions and optimising computational efficiency [7]. Our final preprocessing steps are illustrated in Figure 3. All observations are converted into grayscale images with a resolution of 100 x 100. The frames are stacked in piles of 4 before being passed into the DNN model, so that different motions can be represented. As SMB has a framerate of 60 fps, consecutive frames are likely to give similar information about the state, hence, only one in every 4 frames are used to avoid overlapping frames in the model.

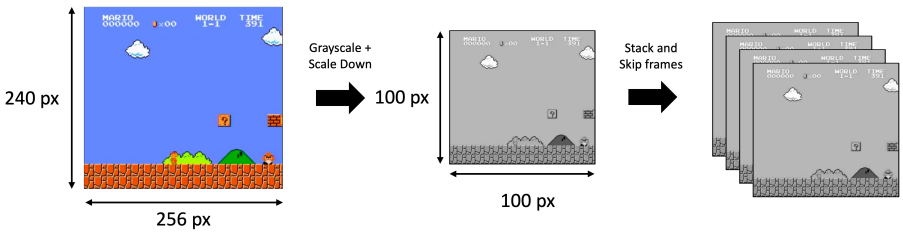

Figure 3: Frames preprocessing steps.

3.2 Deep Q-Network (DQN)

The first agent we developed uses the vanilla DQN algorithm. In every episode, our agent chooses an action based on the ϵ-greedy policy, which means that with a probability ϵ, the agent chooses a random action, otherwise, it chooses the action with the highest Q-value. We decay the ϵ value after every time step, as exploration becomes less important towards the end of training.

At every time step t, experience replay will take place to update the Q-values if the number of experiences in buffer is greater than our specified batch size. The states in the mini-batch of experiences, represented as stacks of frames, are passed into the CNN network for feature extraction and Q-value prediction (Figure 2). The convolutional layer in CNN extracts features vectors from the frames, while the dense layers selects the discriminative features for prediction of Q-values. With the predicted values obtained from the CNN, new target values for the state-actions are computed using equation 2. These target values are then back-propagated through the network to perform a gradient descent update step on the model weights using equation 3. Huber loss is used to stablise the weight updates. The high-level architecture of the CNN model is shown in 4, which is based on the CNN model used in DeepMind's Atari agents [7]. The detailed architecture can be found in 7.

$$y_i = \begin{cases} r_i & \text{for terminal } S_{t+1} \\ r_i + \gamma max_{a'} Q(S_{t+1}, a', \theta) & \text{for non-terminal } S_{t+1} \end{cases} \tag{2}$$

$$\nabla L_\delta (y_i - \hat{Q}(S_i, a_i, \theta)) \tag{3}$$

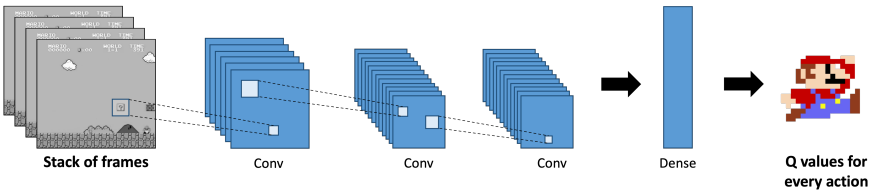

Figure 4: Structure of the CNN model used for the DQN agent.

3.3 Double Deep Q-Network (DDQN)

The main challenge in training a Deep Q-Network is that there is no supervision. In the simple DQN implementation, both the target and current state-action Q-values are estimated using the same model. This leads to the correlation causing the shift, which inhibits the TD error from decreasing; which ultimately makes convergence of the model difficult.

One of the solutions is to use fixed targets, which are temporarily "frozen" in place - allowing for boosted convergence. A commonly used method introduces a second network (target net), which copies the weights of the DQN network (online net) after a number of steps in the environment. In our implementation, the variable controlling that is the *update_steps*.

Additionally, decoupling target Q-values from the online network helps preventing the overestimation of Q-values, which further improves training.

$$y_i = \begin{cases} r_i & \text{for terminal } S_{t+1} \\ r_i + \gamma max_{a'} \hat{q}_2(S_{t+1}, a', \theta_2) & \text{for non-terminal } S_{t+1} \end{cases} \tag{4}$$

$$\nabla L_\delta (y_i - \hat{q_1}(S_i, a_i, \theta_1)) \tag{5}$$

3.4 Dueling Deep Q-Network

This idea is that the state-action Q-values can be decomposed into two terms: state value $V(s)$, which describes the "goodness" of being in a given state, and state-dependent action advantages defined as: $A(S, a) = Q(S, a) - V(s)$, which compare available actions relative to the state.

The standard CNN architecture of three convolutional blocks is used to extract features. After the flattening layer, $V(S)$ and $A(s, a)$ estimators are divided into their own branches, each preceded by its own fully-connected layer.

Finally, the two branches are merged together in the aggregation layer to produce the Q-value output. Note a simple addition is not desired due to the issue of identifiability, i.e. given Q(s,a), V(s) and A(s,a) cannot be recovered, which raises problems in back-propagation. We use a common solution, where state value is added to the advantages tensor reduced by their mean value:

$$Q(s, a) = V(s) + [A(s, a) - avg(A(s, a))] \tag{6}$$

The motivation behind implementing such dual structure is that the network could learn the more valuable states via the $V(S)$ stream and focusing on performing updates towards the actions available in them. It is also useful for finding redundancies in the action space.

4 Results

4.1 Performance Comparison

The DQN, DDQN and Dueling-DQN agents were trained for 1000 episodes to juxtapose the rolling average reward performance for right only and custom actions spaces.

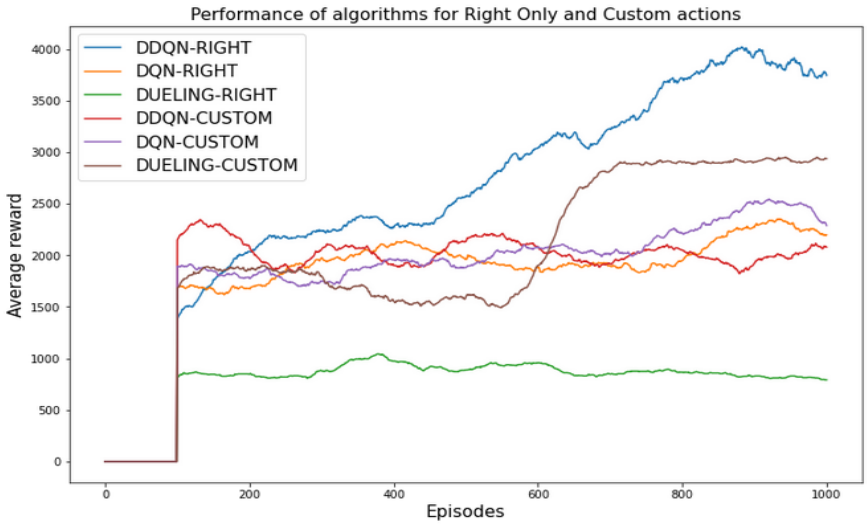

Figure 5: Rolling average performance of different agents based on right only and custom actions

Notably, there is a significant difference between the dueling agents. The custom dueling agent learns optimal movements by 700th episode, whereas right only dueling agent does not improve and likely requires a greater number of episodes to learn optimal actions due to a larger action space.

Both DQN agents for different action spaces achieve similar sub-par performance, where neither displays any significant learning throughout the training.

Lastly, the DDQN-right only agent has performed the best among tested agents. The agent achieves a steep learning curve and vastly outperforms other agents.

4.2 Best Performing Agent

Considering the trade off between performance and computation time, we found that our DDQN agent was best, completing 2500 episodes of training in just over 24 hours of runtime. By the end of training, our agent was successfully able to complete the first level of SMB, and made it to the first Piranha Plant enemy in stage 1-2.

4.3 Baseline Comparison

Our agent significantly outperforms a random agent, whose actions at each time step are uniformly sampled. The vast majority of random agents tested failed to clear the first three pipes. When comparing to human performance, however, the model does not fare well. Stage 1-1 of SMB is a fairly simple level, with most human players, even inexperienced ones, requiring only a few attempts before figuring out the mechanics of the game and completing stage 1-1. The current world record for the fastest human completion of SMB finished stage 1-1 in less than 19 seconds [9] (using a pipe to an underworld, something which our agent was extremely unlikely to find) - in our test run, our agent took 836 seconds to reach the flag at the end of 1-1, losing two lives along the way.

4.4 Prioritised Experience Replay (PER)

PER aims to improve the efficiency of learning by assigning priorities to each of the experiences stored in memory. When sampling, experiences with higher priority are more likely to be chosen for

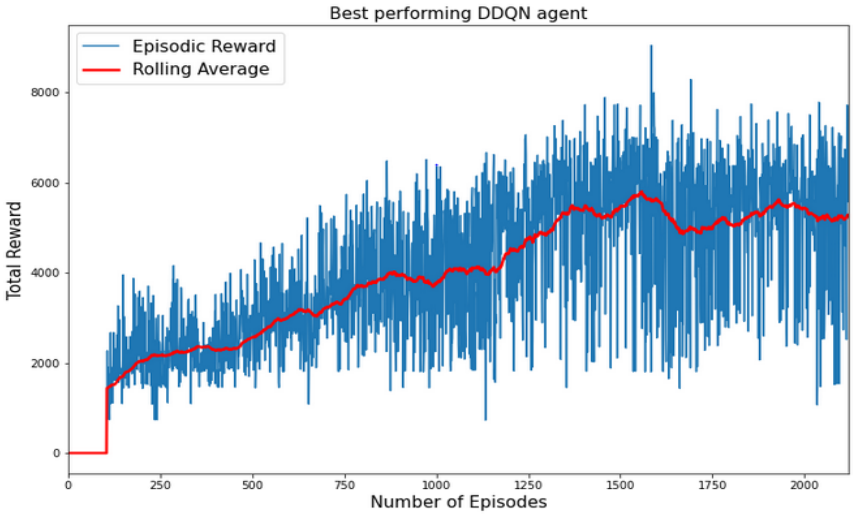

Figure 6: Learning curve of our best performing agent.

model fitting. This is in contrast to Uniform Experience Replay, where each experience is equally likely to be sampled.

Schaul et. al [10] identify two potential equations for calculating the priority p_i given to an experience i:

$$p_i = |\delta_i| + e, \quad \text{where } e \text{ is a small positive constant.} \tag{7}$$

$$p_i = \frac{1}{rank(i)}, \quad \begin{array}{l}\text{where } rank(i) \text{ is the rank of the} \\ \text{experience } i \text{ when the replay buffer is} \\ \text{sorted according to } |\delta_i|.\end{array} \tag{8}$$

The probability of sampling experience i is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \quad \text{where } \alpha \text{ is a constant.} \tag{9}$$

In our implementation, Equation 7 is used. We found Uniform Experience Replay outperformed PER for 500 episodes of training in the right only SMB environment (Figure 14).

5 Discussion

5.1 Successes

After training, Mario was consistently able to identify and avoid the common Goomba and Koopa enemies. Mario was even able to avoid a Goomba falling from a higher platform in stage 1-2. Mario was successfully able to traverse common obstacles such as pipes, and utilised running in order to progress at the fastest possible rate.

5.2 Issues

An issue seen throughout the training process was Mario's tendency to get stuck behind obstacles, repeatedly running into them and failing to jump in order to clear them in the level's time limit. By the end of training, Mario was able to deal with pipes, successfully identifying and jumping over them, but obstacles such as the stairs leading to the flag at the end of the level in 1-1 continued to be an issue. Mario also struggled with the first Piranha Plant enemy of the game - an enemy who

emerges from pipes, preventing the player from jumping over (see Figure 7). By episode 2000, our agent was beating stage 1-1 in training, but very few runs saw Mario getting past this enemy. This is likely caused by the agent's previous experience of pipes which can be safely jumped over - the Piranha Plant represents the an instance in the game where this is not the case.

Figure 7: A safe pipe and a Piranha Plant.

5.3 Evaluation

Overall, given the complicated nature of the problem and large size of the state space, all agents struggled to learn in the environment. The sparsity of rewards and variety of environmental obstacles significantly slowed down the agent's ability to complete the course.

While, the agent performed rather well, improvements could be made by tuning reward signal to encourage the agent to jump over obstacles while being stuck in the same x-position and also force additional incremental penalties for wasting time.

6 Future Work

An area for improvement could be the agent's learning rate. Due to the natural sparsity of rewards within the Super Mario Bros environment, it can be difficult for the agent to learn the optimal action to take because after a series of actions with no reward, it's hard to evaluate these actions. This in turn means that the agent has to do a lot of exploring before learning.

If we had more time and computational power on our project, another way to combat the issue discussed above could be by making use of a concept called Imitation Learning, discussed here [3]. This uses demonstration data, which can be obtained from a domain expert or from simulated experience, to provide trajectories of optimal actions to take in specific states. This means that the agent is almost "pre-trained" on its environment so can perform better from the beginning and then improve upon the data it was provided- leading to an accelerated rate of learning.

7 Personal Experience

We have learned the concept of DQN, DDQN, Dueling-DQN and prioritised replay which help the agent to train how to play Super Mario Bros. The basic prerequisite is Q-learning which we learned at the beginning of the course.

We encountered some difficulties such as the time limit and computational power. We used the university GPU to train the agent, however it has still proven to be insufficient to train the agent extensively for a longer period of time. Due to limited computational power, we were only able to complete hyper-parameter tuning on four hyper-parameters.

We were surprised to see that our DDQN - RIGHT_ONLY agent completed the 1-1 stage in a relatively short number of episodes, while also acknowledging the difficulties it may face in the following runs due to previously unseen environment obstacles.

Overall, the project has taught us the intricacies behind the implementation of RL models, as well as the differences when it comes to their architecture and performance.

References

[1] Ankit Choudhary. Deep q-learning: An introduction to deep reinforcement learning, 2020.

[2] Yuansong Feng, Suraj Subramanian, Howard Wang, and Steven Guo. Train a mario-playing rl agent — pytorch tutorials 1.11.0+cu102 documentation, 2020.

[3] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations, 2017.

[4] Christian Kauten. Github - kautenja/gym-super-mario-bros: An openai gym interface to super mario bros. & super mario bros. 2 (lost levels) on the nes, 2018.

[5] 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. *CoRR*, abs/1602.01783, 2016.

[6] 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.

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

[8] Viet Nguyen. Github - uvipen/super-mario-bros-a3c-pytorch: Asynchronous advantage actor-critic (a3c) algorithm for super mario bros, 2021.

[9] Niftski. Super mario bros. any% speedrun in 4:54.881 *wr*, Dec 2021. http://www.youtube.com/watch?v=aukKeS8LdDI.

[10] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, Feb 2016.

Appendices

Appendix A: Actions available in each action set

	Default action sets in the "gym-super-mario-bros" library			Custom action set
Actions	**RIGHT_ONLY**	**SIMPLE_MOVEMENT**	**COMPLEX_MOVEMENT**	
Do nothing	Y	Y	Y	
Right	Y	Y	Y	Y
Right + A	Y	Y	Y	Y
Right + B	Y	Y	Y	
Right + A + B	Y	Y	Y	
Left		Y	Y	
Left + A			Y	
Left + B			Y	
Left + A + B			Y	
Down			Y	
Up			Y	
A		Y	Y	
Total number of actions	**5**	**7**	**12**	**2**

Table 2: Actions available in each action set.

'A' refers to jump, while 'B' refers to run or shoot flames when Mario has the fireball power-up.

Appendix B: Detailed Architecture of the CNN Models

Figure 8 shows the detailed architectural structure of the CNN model for the DQN agents (non-dueling) with the RIGHT_ONLY action set. Processed image frames are feed into the CNN network as input, and the predicted Q-values for every action in the action set are outputted by the model. Figure 8 shows a similar architecture for dueling models.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 100, 100, 4)]     0

conv2d (Conv2D)              (None, 24, 24, 32)        8224

conv2d_1 (Conv2D)            (None, 11, 11, 64)        32832

conv2d_2 (Conv2D)            (None, 5, 5, 64)          16448

flatten (Flatten)            (None, 1600)              0

dense (Dense)                (None, 512)               819712

actions (Dense)              (None, 5)                 2565

=================================================================
Total params: 879,781
Trainable params: 879,781
Non-trainable params: 0
_________________________________________________________________
```

Figure 8: Detailed Architecture of the CNN Model of the DQN Agents.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 100, 100, 4)]	0
conv2d (Conv2D)	(None, 24, 24, 32)	8224
conv2d_1 (Conv2D)	(None, 11, 11, 64)	32832
conv2d_2 (Conv2D)	(None, 5, 5, 64)	16448
flatten (Flatten)	(None, 1600)	0
dense (Dense)	(None, 128)	204928
dense_1 (Dense)	(None, 128)	204928
value (Dense)	(None, 1)	129
advantage (Dense)	(None, 5)	645

Total params: 468,134
Trainable params: 468,134
Non-trainable params: 0

Figure 9: Detailed Architecture of the CNN Model of the Dueling Agents.

Appendix C: Hyper-parameter Optimisation for DDQN

There were several hyper-parameters tuned to optimise our agent's performance.

We first investigated the best value of epsilon ϵ which controlled the amount of time the agent would spend exploring the environment. We used annealing within our ϵ-greedy strategy, by setting an *epsilon_decay* value, with the idea being that the agent would explore more at the beginning and once it gains more knowledge about its environment it reduces the amount of exploring and starts exploiting the best moves to take. From Figure 10, it's clear to see that an *epsilon_decay* of 0.99 performed best in our model. This meant our agent spent a large amount of time initially exploring its environment but then started to exploit its best actions in later episodes and reduced the amount of exploring.

We then tuned the discount factor gamma γ, the value that decides how much importance is giving to future rewards. This value ranges from 0 to 1 - a value of γ closer to 0 would result in more weight being given to immediate rewards, while a value closer to 1 would delay rewards and give more weight to rewards in the future. Figure 11 shows how we tried two different values for γ and found that a value of $\gamma = 0.9$ gave us the best performance.

The *learning_rate* dictates how the weights in the model are updated, a small learning rate means that more training is required due to only small changes to the weights occurring with every update- this can lead to the weights getting stuck in a local minima and not being optimised. A rate too large can cause a model to converge too quickly and overshoot the weights leading to a sub-optimal solution. We found a rate of 0.00025 worked best for our model.

The *max_memory* parameter within our model refers to the size of the replay buffer. This contains the tuples of experience (transitions) that the agent has encountered in training and allows the agent to make use of previous experience to learn. If the buffer is too small then is serves no purpose in allowing the agent to use previous experience to learn. However, if the buffer size is too large the samples taken from the buffer will be uncorrelated and it will be difficult to learn from these samples. Using Figure 12, we found a value of 5000 was better than a smaller buffer of 500, supporting our initial belief that a higher amount of memory used for storing experiences would lead to higher performance. In our highest performing agent, we wished to utilise as much memory as was possible, given the amount of compute we had access to, and this agent used a *max_memory* of 20,000.

We choose how often we sync the model weights from the main DQN network to the target DQN through the use of the *update_steps* parameter. We found that the frequency of updates had a marginal

effect on rewards with an update step size of 5000 performing slightly better than 1000 in later episodes (Figure 13).

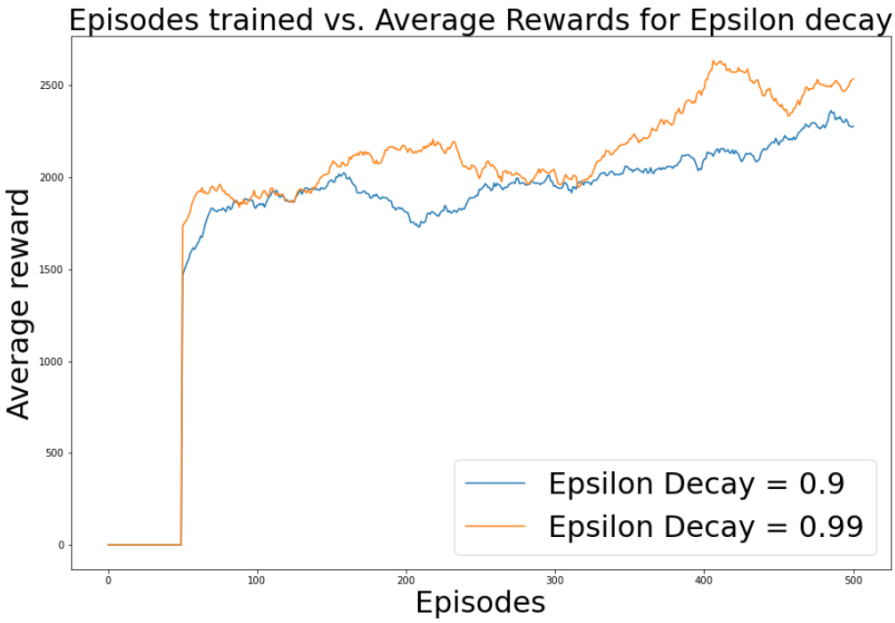

Figure 10: Rolling average rewards comparison for epsilon decay values

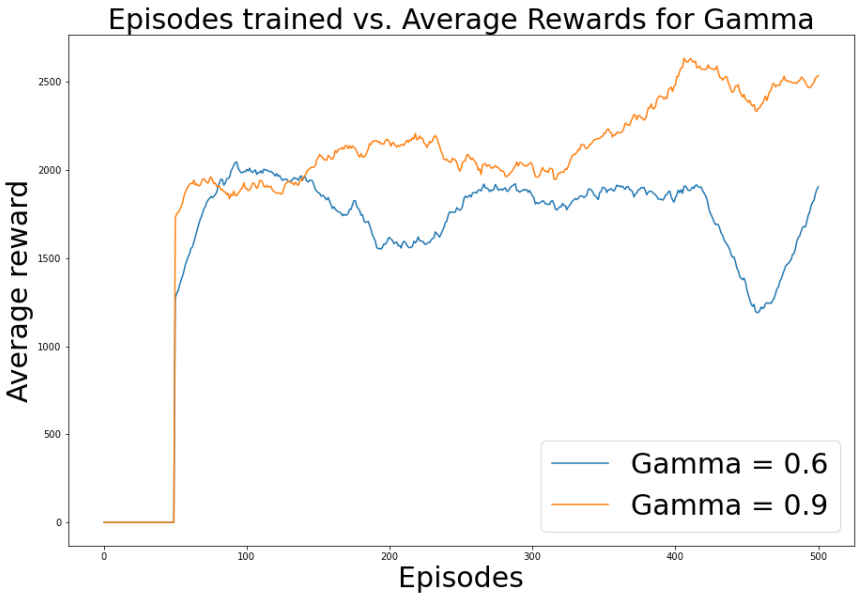

Figure 11: Rolling average rewards comparison for gamma values

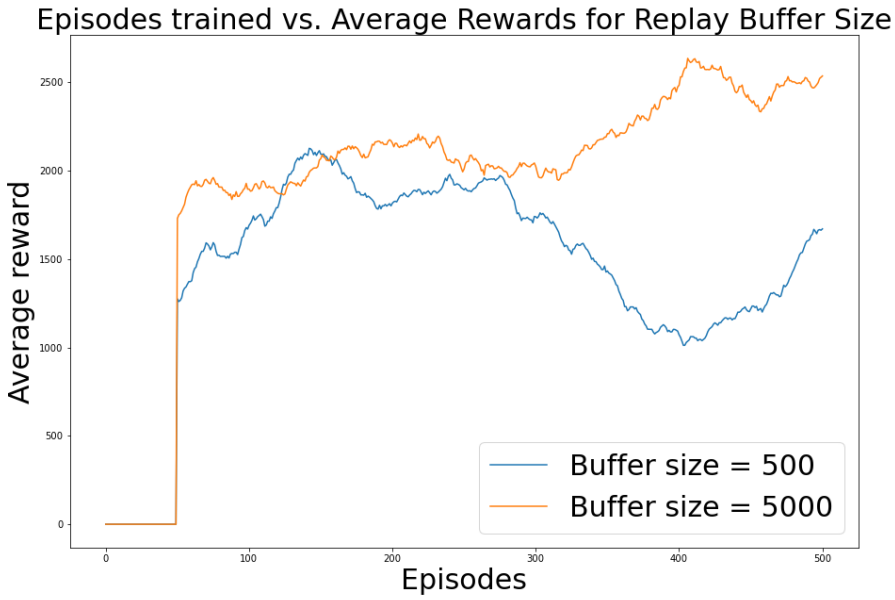

Figure 12: Rolling average rewards comparison for replay buffers

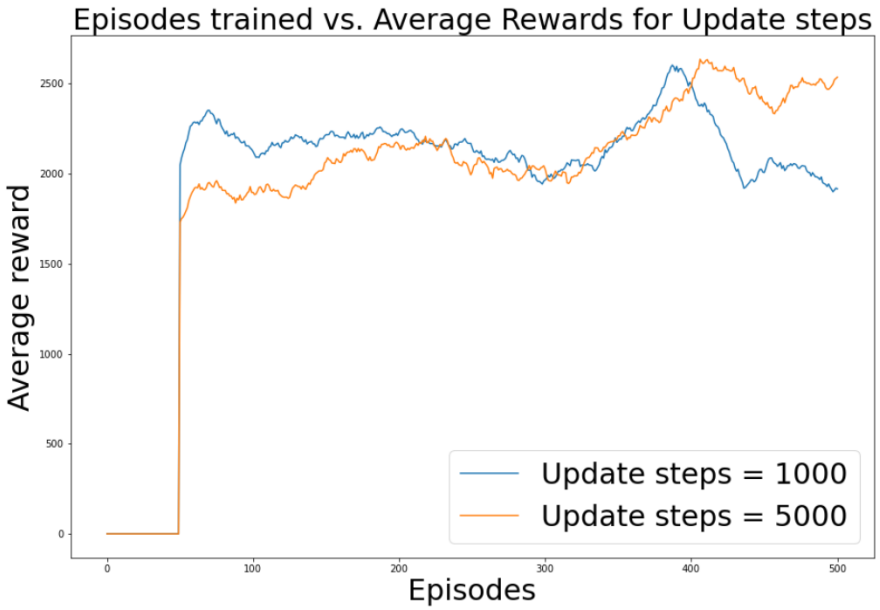

Figure 13: Rolling average rewards comparison for update step values

Appendix D: PER and Uniform experience replay DDQN and DQN

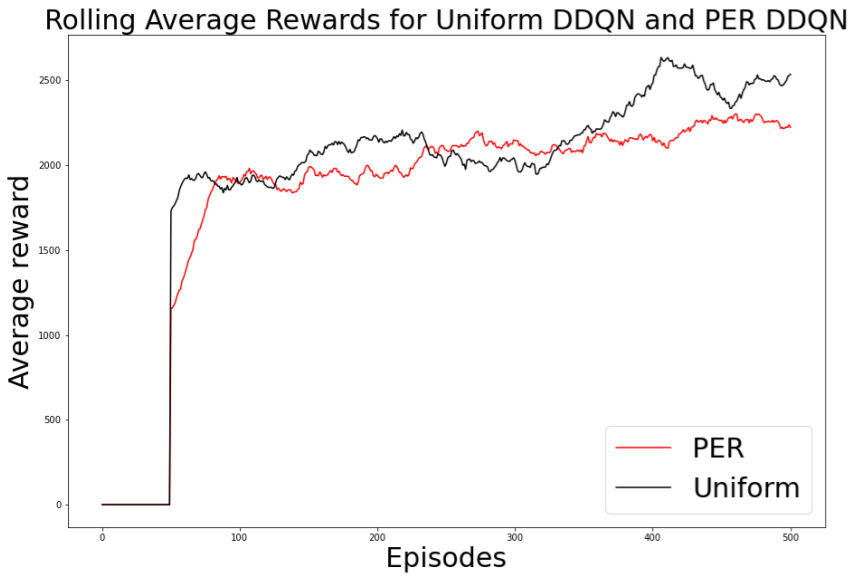

Figure 14: Rolling average rewards comparison for PER and uniform DDQN

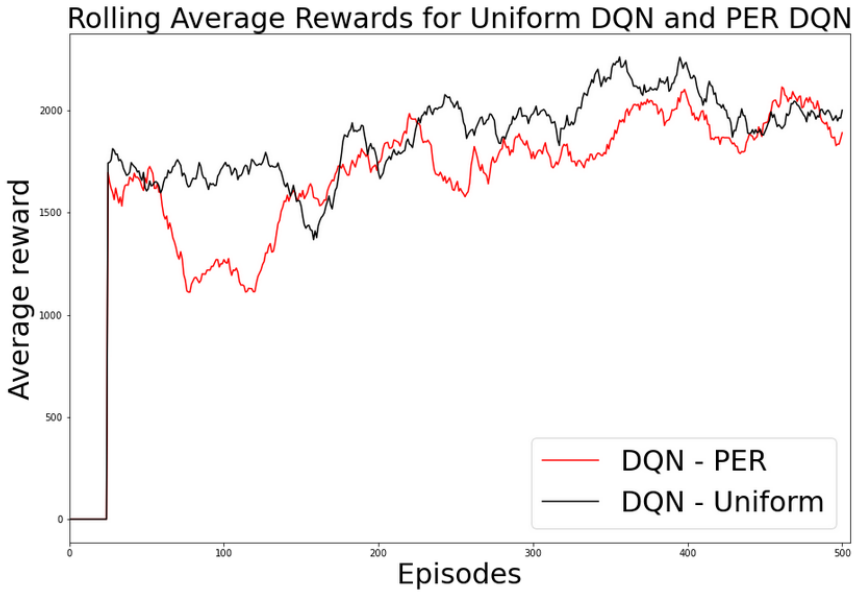

Figure 15: Rolling average rewards comparison for PER and uniform DQN