Reinforcement Learning Project

Freud Fan
qf265@bath.ac.uk

Nikhil John
ntj25@bath.ac.uk

Iason Mylonas
im625@bath.ac.uk

Pavlina Paschalidou
pp866@bath.ac.uk

Shashank Sharma
ss3966@bath.ac.uk

1 Problem Definition

The aim of this project was to implement a Machine Learning model for the video game Sonic the Hedgehog. This problem was chosen since it offers a level of difficulty that other games may not. In contrast to single-screen games, the levels are long horizontal obstacle courses, requiring the model to adjust quickly. Moreover, several areas of a level require the agent to acquire substantial speed to complete, implying that the player must approach the obstacle in a quick manner. Because of the game's branching level design, which offers different routes for the agent to take, early actions can have a significant influence on subsequent performance. In addition, the agent's decisions may be influenced by the intricate geometrical patterns of the level, in comparable events. The scope of the project was restricted to the game's initial zone, Green Hill Zone (Act 1) (Figure 1).

Figure 1: Green Hill Zone Act 1 Map
(Zone: 0, 2010)

1.1 Sonic the Hedgehog - Genesis Video Game

Sonic The Hedgehog is a platform game with a 2D side-scrolling gameplay as shown in Figures 2& 3. An agent has three lives. Every time the agent hits an enemy or faces a danger and dies, he is re-positioned at the beginning of the game, and can continue playing. Once he runs out of lives it's game-over. The basic goal of the game is for Sonic to gather rings while traveling through vertical loops, across bottomless pits, and off of springs. Rings are arranged in groups of three or more and they award points that serve as a shield. Sonic loses a life in case he is attacked by an enemy or faces a danger while having no rings. However, if Sonic has at least one ring, he gets knocked backwards and loses up to 20 rings without losing a life by hitting an enemy or hazard.

Figure 2: Snapshot from the game
(Zone: 0, 2010)

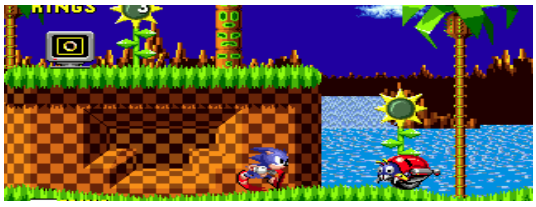

Figure 3: Snapshot from the game
(Zone: 0, 2010)

1.2 Gym Retro

Gym retro is a project that focused at constructing RL environments from several video games. An important part of Gym Retro is the gym-retro Python library, that includes Gym (Brockman et al., 2016) environments for several emulated games (including Sonic the Hedgehog). Similarly to RLE (Bhonker, Rozenberg and Hubara, 2016), gym-retro exploits the *libretro* API to interface with game emulators, allowing for simple additions of new emulators to gym-retro.

Games are stored in a dataset that is included in the gym-retro package. Every game is comprised of a ROM, one or more save states, one or more scenarios, and a data file.

- ROM: essential data and code for a game purchased from *Steam*.
- Save state: a capture of the status of the console at some time throughout the game.
- Data file: contains a description of locations (in console memory) of several parts.
- Scenario: a summary of the reward functions and terminal state conditions.

1.3 Sonic the Hedgehog environment

Similarly to other reinforcement learning problems, an agent follows a policy π given parameters θ and returns actions given an observation $\pi_\theta : o \rightarrow A$. The agent performs an action $a \epsilon A$ (where A is the set of possible actions) whilst in state $s \epsilon S$ (where S is the set of possible states). Each action corresponds to a combination of buttons on the game console. Actions are represented by binary vectors (a shape [12] array of boolean values), with 1 indicating "pressed" and 0 indicating "not pressed". Since only a few actions and button combinations are relevant, the action space is reduced and the following used in the project:

A = {{}, {LEFT}, {RIGHT}, {LEFT, DOWN},
{RIGHT, DOWN}, {DOWN}, {DOWN, B}, {B}}}

The agent performs an action and is rewarded based on the state that results from the action. The reward of an agent is proportionate to its movement toward the predetermined horizontal offset in each level, with positive rewards for getting closer and negative rewards for getting further away. Reaching the offset results in a total award of 9000. Additionally, a time bonus is used, with initial value of 1000 that drops to 0 when the time is up.

Once an action is performed, the agent transitions to state $s' \sim T(s, a, s')$ and gets observation $o' \sim O(o', s')$ as well as reward $r = R(s, a) \epsilon R$. For instance, when an enemy is getting close to Sonic, the game state s indicates the magnitude and the direction of the velocity of the enemy. An observation o of this state is a video frame at the current instance of time that may inform that the enemy is in close proximity. Therefore, the player can take an action to attack the enemy. This can result in a transition (s, a, s') to a new state s' in which the enemy is destroyed and consequently, result in a new visual observation o' of the destroyed enemy.

2 Background

Several reinforcement methods could be effective at solving Sonic the Hedgehog. In this section, various potential solution algorithms are presented.

2.1 Deep Q Network (DQN), Double DQN, Dueling DQN

Deep Q Network (DQN) Mnih (2015) was a significant accomplishment that combines Q-learning algorithm in Reinforcement learning with concepts of Deep Learning (convolutional neural networks). Over the years, several improved DQN algorithms have been proposed. For instance, **Double DQN** (van Hasselt, Guez and Silver, 2015) is an extension that solves issues caused by overestimation bias of Q-learning (full algorithm can be found in Appendix B). **Prioritised experience replay** (Tom Schaul and Silver, 2016), another extension algorithm, enhances data efficiency, by replaying more often transitions from which there is more to learn. The **Dueling DQN** (Wang, de Freitas and Lanctot, 2015) allows for separate representation of state values and action advantages, which allows for better generalisation of actions.

2.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimisation(PPO) is an algorithm released by OpenAI in 2017. PPO is a policy optimisation method that constitutes an important improvement on Trust Region Policy Optimization (TRPO) method and is based on Advantage Actor Critic (A2C) (Zakharenkov and Makarov, 2021). The goal of PPO is to achieve a balance amongst critical factors such as implementation simplicity, ease of tuning, complexity of sample and attempting to update an existing policy at every step so that the cost function is minimised. The loss for the PPO method is given by:

$$\mathrm{L}^{CLIP}(\theta) = \bar{E}_t[min(r_t(\theta)\bar{A}_t, clip(r_t(\theta), 1-\epsilon, a+\epsilon)\bar{A}_t]$$

where θ is the policy parameter, $\bar{E}_t$ denotes the empirical expectation over timesteps, r_t denotes the ratio of the probabilities under the new and old policies respectively (i.e. Importance Sampling ratio), $\bar{A}_t$ is the estimated advantage at time t, ϵ is a hyperparameter (usually 0.1 or 0.2).

2.3 Just Enough Retained Knowledge (JERK)

JERK is an algorithm proposed in 2018 (Nichol et al., 2018) that it based only on rewards and absolutely disregards observations (no use of deep learning). The basic concept is to use a simple algorithm to explore, then play again using the optimal actions and action order (*"action sequence"*) more regularly as training continues. A running mean of rewards for each *"action sequence"* is used, to distinguish which is the best *"action sequence"* to play again. The full JERK pseudocode for the algorithm can be found in the Appendix A.

3 Method

3.1 The Q-Learning Network architecture

The model used in order to solve the problem is a **Deep Q-Learning Neural Network** (DQN) Mnih (2015), an algorithm developed by DeepMind in 2013 to play Atari arcade games. It was developed by enhancing the Q-Learning algorithm which was introduced various features in order to ensure stability during training. DQN (full algorithm can be found in Appendix B) is a value-based deep Reinforcement Learning method which combines Q-learning with deep neural network architecture in order to approximate the action-value function $Q(s, a)$. The authors use a Convolutional Neural Network (CNN) that takes the current observation as input image and outputs a vector of size as the action-space, with each value representing the Q-value of the action for the given state. Figure 4 shows an illustration of their approach.

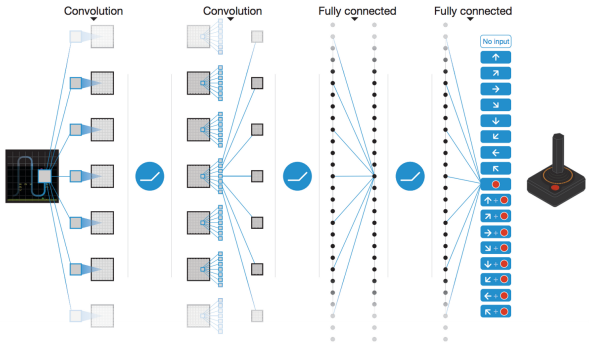

Figure 4: Schematic illustration of the convolutional neural network architecture used by (Mnih, 2015)

The CNN maps state inputs to action-value outputs as seen in Figure 4. The state input is an image constructed by pixels so the CNN takes into account neighbourhoods of nearby pixels instead of treating them all as independent inputs. It is structured with two convolutional layers and two fully-connected dense layers, made up of densely connected single neurons. There is one output layer representing the Q-value of all possible actions in the state (game) and a linear activation on the

output layer (Mnih, 2015).

A similar architecture is used making changes to account for a higher state space. The DQN used is shown in Table 1. The network contains a CNN with three convolutional layers. A 128x128x3 image is used as the neural network's input as observation. The first hidden layer consists of 32 8x8 filters with stride 2 , followed by a LeakyReLU activation. The second hidden layer consists of 64 4x4 stride 2 filters, which are followed by a LeakyReLU unit. The third convolutional layer consists of 64 3x3 stride 1 filters, followed by a LeakyReLU. The last hidden layer has 512 rectifier units and is fully linked. The output layer is a fully-connected linear layer with a single output for each possible actions given by Eq. 1.3.

Table 1: CNN-based Q-Learning Network that takes state size of 128x128x3 and outputs a vector with size as action-space(=8).

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	128x128x3	8x8	4	32	LeakyReLU	32x32x32
conv2	32x32x32	4x4	2	64	LeakyReLU	16x16x64
conv3	16x16x64	3x3	2	64	LeakyReLU	16x16x64
flatten4	16x16x64					16,384
fc5	16,384			512	LeakyReLU	512
fc6	512			8	Linear	8

3.2 Experience Replay

In order to make network updates more reliable, ***Experience Replay*** was introduced. The agent's experiences are collected whilst the agent generates them. These are stored to a memory buffer D of the form (s_t, a_t, r_t, s_{t+1}), called the replay buffer, at each time step of data gathering. Instead of computing the loss and gradient using only the most recent transition, the updates are then performed using the mini-batch of experiences taken from the replay buffer during training. This results in improved data efficiency by reusing each transition in several updates, and improved batch stability by employing uncorrelated transitions.

DQN was chosen to solve the Sonic the Hedgehog problem for a few reasons. DQN has the ability to train utilising delayed and sparse rewards in highly dimensional input fields. Apart from this, it was able to learn all 49 Atari games with no changes to the model's fundamental architecture and hyperparameters, which demonstrates the generality of the approach. Moreover, DQN tends to work best with a small, discrete action space and, as mentioned above, Sonic the Hedgehog has an action space containing eight actions.

3.3 Freezing Network for Updates

When training the Q-Network while simultaneously using it for inference for generating the targets causes instability Evans (2021). This is caused due to trying to chase a moving target. To address this issue, we train the network in short frequent cycles, and the target network is fixed for the duration of each training cycle.

3.4 Prioritised Experience Replay (PER)

The DQN model is further enhanced by the implementation of **Prioritised Experience Replay** (Tom Schaul and Silver, 2016), which is a modification of the Experience Replay technique included in the DQN method mentioned earlier. Prioritised Experience Replay aims to apply different types of priority in the experience replay process in order to improve the sampling efficiency among the replay memory. The prioritizing is achieved by prioritizing which input experience instances from the source system are saved in the experience trail, from which they can be randomly selected for "replay." The following conditions were checked for storing an experience in the buffer:

- **Terminal state:** If the current state is the terminal state.

- **Error in Q-value:** If the difference between the predicted and target Q-values is large.
- **No best action:** If the difference between the Q-values of the best and worst action is too small.

3.5 Double DQN

Double DQN (van Hasselt, Guez and Silver, 2015) has also been implemented. Standard Q-learning suffers from an overestimation bias under certain conditions, as mentioned in section 2.1. Double DQN suggests using a second network as the target model with corresponding Q-value for each state-action pair. The value of the greedy policy in Q-learning is still estimated according to the current values, while the value of this policy is evaluated using the weights from the target network. In particular, the updating equation is now $y_i = r + \gamma Q(s_{i+1}, argmax_{a_{i+1}} Q(s_{i+1}, a_{i+1}; \theta); \theta^-)$ where θ^- is a copy of the initial network parameters. The rate of updating from primary network to target network can be treated as a hyper-parameter hence controls the learning process. The pseudocode of Double DQN algorithm can be found in Appenidx C.

4 Results

Three variants of the Deep Q-Learning algorithm have been implemented:
(Source code: `https://github.bath.ac.uk/ss3966/RLProject`)

- DQN with experience replay and network freeze during the training cycle.
- DQN model (previous) with Prioritised experience replay (PER).
- Double-DQN algorithm.

The agent performance is measured as the total cumulative reward it receives during the episode. The total rewards are plotted versus the episode number in the Figure 5. It can be seen that while the initial implementation of the DQN algorithm manages to score around 5000 points in best case, the PER version scores more points and performs slightly more consistently. The PER model also manages to reach a score > 9000 at episode 86.

The average human scores over the course of an hour are displayed in Appendix D (Nichol et al., 2018). The authors do not test the human performance for this level but the average human performance is listed as 7438.2 ± 624.2

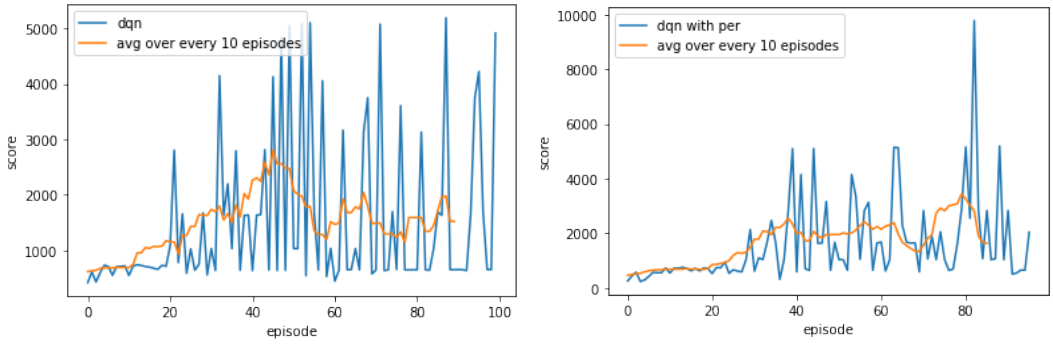

Figure 5: DQN (left) and DQN with Prioritised Experience Replay (right).

In Figure 5 despite of a fluctuating score over 100 episodes for both DQN (left) and DQN with prioritised experience replay (right), it shows an overall increasing trend in their average over every 10 episodes. The drop after 50 episodes for DQN is probably due to model forgetting the past learning. This is also noticable in the work in (Nichol et al., 2018). The performance after implementing prioritised experience replay on DQN is higher than before as expected, with an instant maximum at around 9800 almost doubling that of the original DQN.

The double DQN starts at a higher score between 1000 and 2000 compared to simple DQN and DQN with prioritised experience replay, as shown in Figure 6. It has a rising trend however not trivial till

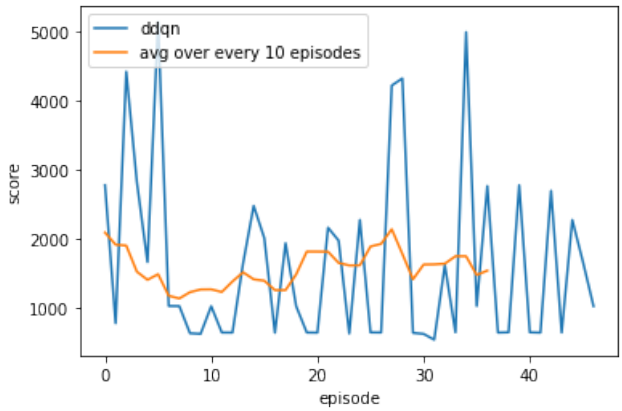

Figure 6: Double DQN.

later episodes, due to time limitation on the model training and the transferring rate between primary network and secondary network incorporated in the model.

5 Discussion

As seen from the figures in Section 4 the model performance is not entirely stable. The model tends to overfit and forget past learning which leads to the performance decreasing for a small period of time and then increasing again. The PER seems to tackle this issue to a certain degree where the learnings don't decrease in such a substantial manner.

Moreover, double DQN does not have enough time to optimise the algorithm which leads to it under-performing. However it gives a better performance at the beginning when the model lacks information on the optimal action, by utilising a target network to avoid the noisy maximum q value hence giving fewer false positives.

Sonic the Hedgehog is a complex game and not as straightforward in several areas. At certain points, where the agent is found in a dead end, the agent is required to trace back its steps and choose an alternative route. This is arguably difficult to model and a known problem in the environment that is faced a few times.

The training times of the model were also relatively high leading to training for longer hours. The required longer training times caused problems for multiple iterations of the agent optimizations.

6 Future Work

In the event of additional time available for the completion of the project, there are a few improvements and implementations that could be applied. One of these would be the further enhancement of the DQN method, such as the application of Huber Loss which is less sensitive to large errors caused by the MSE, which normally would result in large network weight updates. Another improvement would be reward clipping/normalising where rewards would be clipped in the range $[-1, 1]$ and the gradients of the loss function before performing an update. Also, track the observed rewards and normalise them as we progress.

The use of alternative Deep Q-Learning techniques in order to solve the problem, such as Double (van Hasselt, Guez and Silver, 2015) or Dueling Deep Q Networks (Wang, de Freitas and Lanctot, 2015),would be beneficial. In addition, more reinforcement algorithms could be implemented to training agents. For instance, Proximal Policy Optimisation (PPO) and Rainbow are two of the most popular algorithms that could be implemented in the future.

After successfully finishing the first zone of the video game, the natural next step for a Machine Learning implementation would be to move onto the next zone (Marble Zone, Spring Yard Zone, Labyrinth Zone, Star Light Zone, Scrap Brain Zone, Final Zone). The zones of the game become

increasingly more complex, with larger distances for the agent to cover, multiple interconnected paths and introduction of more dangers such as water, lava and rocky terrain. Some zones would be possible but difficult to solve with the current implementation and some others completely unbeatable.

At the moment the agent is focusing on the distance travelled in the game. In future work, custom reward functions could be created where the agent would be rewarded for collecting coins and killing enemies. Furthermore, negative rewards could be explored, for example from dying caused by an enemy.

This project focused on the video game aspect of the game even though the game is typically played with video and audio. In the 2021 paper, Faraaz Nadeem (Nadeem, 2021) proposed that the agent could learn from the musical feedback. he concluded that game audio informs useful decision making, and that audio features are more easily transferable to unseen test levels than video features.

7 Personal Experience

Prior to this project, the team had very little experience with Reinforcement Learning. This project enabled the team to apply the information gained from the course to a real life problem and produce a working model with satisfactory results. The extra time spent to truly grasp the underlying mechanics of the neural network was necessary and extremely beneficial to the project. This resulted in an overall more deepened understanding of Reinforcement Learning. It is a diversified field with plenty of possibilities and significant differences from other Machine Learning approaches.

As mentioned in section 1, Sonic the Hedgehog - Genesis environment is indeed a complex environment with a lot of challenges. The training of agents required relatively high duration that allowed us to test only a limited number of agents. Another challenge posed was the implementation of the Double DQN Agent along with Prioritised Experience Replay (PER). A working code was created, but the training loop took awfully long to execute since it trained 3 episodes in 90 minutes, and for decent results, at least 250 episodes were required.

References

Bhonker, N., Rozenberg, S. and Hubara, I., 2016. Playing SNES in the retro learning environment. *Corr* [Online], abs/1611.02205. 1611.02205, Available from: http://arxiv.org/abs/1611.02205 [Accessed 4 May 2022].

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. *Corr* [Online], abs/1606.01540. Available from: http://arxiv.org/abs/1606.01540 [Accessed 3 May 2022].

Evans, J., 2021. Lesson 14 - dqn deep-dive [Online]. Available from: https://moodle.bath.ac.uk/pluginfile.php/1100399/course/section/174649/Lecture%2014%20-%20DQN%20Deep%20Dive.pdf [Accessed 26 April 2022].

Hasselt, H. van, Guez, A. and Silver, D., 2015. Deep reinforcement learning with double q-learning. *Corr* [Online], abs/1509.06461. 1509.06461, Available from: http://arxiv.org/abs/1509.06461 [Accessed 30 April 2022].

Mnih, V., K.K.D.S.D., 2015. Human-level control through deep reinforcement learning. *Nature* [Online], 518, p.529–533. Available from: https://doi.org/10.1038/nature14236 [Accessed 1 May 2022].

Nadeem, F., 2021. [Online]. Available from: https://archives.ismir.net/ismir2021/paper/000059.pdf [Accessed 2 May 2022].

Nichol, A., Pfau, V., Hesse, C., Klimov, O. and Schulman, J., 2018. Gotta learn fast: A new benchmark for generalization in RL. *Corr* [Online], abs/1804.03720. 1804.03720, Available from: http://arxiv.org/abs/1804.03720 [Accessed 2 May 2022].

Tom Schaul, John Quan, I.A. and Silver, D., 2016. Prioritized experience replay [Online]. [Online]. Available from: https://arxiv.org/pdf/1511.05952.pdf [Accessed 28 April 2022].

Wang, Z., Freitas, N. de and Lanctot, M., 2015. Dueling network architectures for deep reinforcement learning. *Corr* [Online], abs/1511.06581. 1511.06581, Available from: http://arxiv.org/abs/1511.06581 [Accessed 29 April 2022].

Zakharenkov, A. and Makarov, I., 2021. Deep reinforcement learning with dqn vs. ppo in vizdoom. *2021 ieee 21st international symposium on computational intelligence and informatics (cinti)* [Online]. pp.000131–000136. Available from: https://doi.org/10.1109/CINTI53070.2021.9668479 [Accessed 2 May 2022].

Zone: 0, 2010. [Online]. Available from: http://www.soniczone0.com/games/sonic1/greenhill/#act1 [Accessed 4 May 2022].

Appendices

Appendix A: JERK Algorithm Description

Algorithm 1 The JERK algorithm. For our experiments, we set $\beta = 0.25$, $J_n = 4$, $J_p = 0.1$, $R_n = 100$, $L_n = 70$.

Require: initial exploitation fraction, β.
Require: consecutive timesteps for holding the jump button, J_n.
Require: probability of triggering a sequence of jumps, J_p.
Require: consecutive timesteps to go right, R_n.
Require: consecutive timesteps to go left, L_n.
Require: evaluation timestep limit, T_{max}.
$S \leftarrow \{\}$, $T \leftarrow 0$.
repeat
  if $|S| > 0$ and $RandomUniform(0,1) < \beta + \frac{T}{T_{max}}$ **then**
    Replay the best trajectory $\tau \in S$. Pad the episode with no-ops as needed.
    Update the mean reward of τ based on the new episode reward.
    Add the elapsed timesteps to T.
  else
    repeat
      Go right for R_n timesteps, jumping for J_n timesteps at a time with J_p probability.
      if cumulative reward did not increase over the past R_n steps **then**
        Go left for L_n timesteps, jumping periodically.
      end if
      Add the elapsed timesteps to T.
    until episode complete
    Find the timestep t from the previous episode with the highest cumulative reward r.
    insert (τ, r) into S, where τ is the action sequence up to timestep t.
  end if
until $T \geq T_{max}$

(Nichol et al., 2018)

Appendix B: DQN Algorithm Description

Algorithm: Deep Q-Learning with Experience Replay and Fixed Target Network
Initialise replay memory D to capacity N
Initialise action-value network $\hat{q}_1$ with arbitrary weights $\boldsymbol{\theta}_1$
Initialise target action-value network $\hat{q}_2$ with weights $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$

For episode $= 1, M$ **do**
  Initialise initial state S_1
  For $t = 1, T$ **do**
    With probability ϵ select random action A_t
    With probability $1 - \epsilon$ select action $A_t = \text{argmax}_a \hat{q}_1(S_t, a, \boldsymbol{\theta}_1)$
    Execute action A_t, observe reward R_t and next state S_{t+1}
    Store transition (S_t, A_t, R_t, S_{t+1}) in D
    Sample random minibatch of transitions (S_j, A_j, R_j, S_{j+1}) from D
    Set $y_j = \begin{cases} R_j + 0, & \text{if } S_{j+1} \text{ is terminal} \\ R_j + \gamma \max_{a'} \hat{q}_2(S_{j+1}, a', \boldsymbol{\theta}_2), & \text{otherwise} \end{cases}$
    Perform gradient descent step $\nabla_{\boldsymbol{\theta}_1} L_\delta \left(y_j, \hat{q}_1(S_j, A_j, \boldsymbol{\theta}_1) \right)$
    Every C steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$
  End For
End For

(Evans, 2021)

Appendix C: Double DQN Algorithm Description

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

(Wang, de Freitas and Lanctot, 2015)

Appendix D:

State	Score
AngelIslandZone Act2	8758.3 ± 477.9
CasinoNightZone Act2	8662.3 ± 1402.6
FlyingBatteryZone Act2	6021.6 ± 1006.7
GreenHillZone Act2	8166.1 ± 614.0
HillTopZone Act2	8600.9 ± 772.1
HydrocityZone Act1	7146.0 ± 1555.1
LavaReefZone Act1	6705.6 ± 742.4
MetropolisZone Act3	6004.8 ± 440.4
ScrapBrainZone Act1	6413.8 ± 922.2
SpringYardZone Act1	6744.0 ± 1172.0
StarLightZone Act3	8597.2 ± 729.5
Aggregate	7438.2 ± 624.2

Figure 7: Detailed evaluation results for humans for different Zones and Acts.
(Nichol et al., 2018)