CM30359: Investigation of Deep Q-learning on Breakout - Group 9

Orr Bezalely
Department of Computer Science
University of Bath
Bath, BA2 7AY
ob461@bath.ac.uk

Erik P. Leclercq
Department of Computer Science
University of Bath
Bath, BA2 7AY
epl28@bath.ac.uk

Julian Jimenez Nimmo
Department of Computer Science
University of Bath
Bath, BA2 7AY
jjn34@bath.ac.uk

Fabio A. Schwandt
Department of Computer Science
University of Bath
Bath, BA2 7AY
fs652@bath.ac.uk

1 Problem definition

Breakout is a single-player Atari game in which the player has five lives to gain points by clearing bricks from the screen using a ball and a paddle. There are six rows of bricks located at the top of the screen, split into three uniquely coloured sections. The bottom section is worth one point, the middle section is worth four points and the top section is worth seven points. The paddle is located at the bottom of the screen and is controlled by the player. When the game starts, the ball appears in one of three starting positions, with either a south-east or south-west trajectory. The player controls the horizontal movement of the paddle, losing a life if the ball has a lower vertical position than the paddle. There are four possible actions: move the paddle left, move the paddle right, no operation and shoot. The shoot action spawns the ball at the start of the game or after losing a life. No operation means that the paddle board will not move and stay in its current position. The ball bounces off the paddle, the borders and the bricks, eliminating the bricks on contact and gaining their respective number of points. The environment will be provided by OpenAI [10].
The game can be represented by the following Markov Decision Process (MDP):

S is the set of possible states, with a state $s \in S$ returned by the environment being a frame of the game represented by a $210 \times 160 \times 3$ RGB image in addition to the number of lives the player has, with terminal states being states where the player has 0 lives. Thus, each state is defined by a combination of the paddle position, ball position, whether each brick exists or not, and the number of lives, so that $|S| \approx 1.6 \times 10^{12}$, calculated by:

$$\underbrace{6 \times 2^{18}}_{\text{Brick configurations}} \times \underbrace{32 \times 85}_{\text{Ball positions}} \times \underbrace{75}_{\text{Paddle positions}} \times \underbrace{5}_{\text{Lives}} \approx 1.6 \times 10^{12}$$

The 4 actions available are: $A(s) = \{NOOP, FIRE, RIGHT, LEFT\}$ for every non-terminal state $s \in S$, being an empty set for terminal states.

Given an environment at state $s \in S$ and an action $a \in A(s)$, the environment then processes the action according to game physics rules using the command env.step(a), which returns:

1. A new state $s' \in S$.
2. A reward $r \in \mathbb{R}$.
3. Meta data containing current number of lives.

This gives us the following transition dynamics and reward function:

Transition dynamics: $P(s' \mid s, a) = \begin{cases} 1 \text{ if } s' \text{ returned by env.step}(a) \\ 0 \text{ otherwise} \end{cases}$

Reward function: $R(s, a) = r = \begin{cases} 1 & \text{brick in layers 1,2 being broken} \\ 4 & \text{brick in layers 3,4 being broken} \\ 7 & \text{brick in layers 5,6 being broken} \\ 0 & \text{otherwise} \end{cases}$

At the initiation of the environment, the starting state is given by env.reset(). Hence, the starting state distribution is $H(s) = \begin{cases} \frac{1}{6} & \text{if } s \in Y \\ 0 & \text{otherwise} \end{cases}$ for $s \in S$

where Y is the set of all possible starting states described above.

2 Background

There are various approaches to solve Breakout. A traditional way to solve an MDP is to use a tabular method. Tabular methods store the value of each state-action pair in a look-up table, then from these values, they decide which action to execute. Examples of such algorithms are SARSA and Q-learning [12], [18]. One drawback of tabular methods is that they are likely to crash due to memory constraints derived from the immense state-space of Breakout. This, alongside the fact that updates only change one state at a time, makes tabular methods unsuitable for the chosen problem.

Another class of methods that can be applied to this problem are function approximated methods, which estimate the value function or policy function. These are further classified into three categories: value-based methods, policy-based methods and actor-critic methods.

Value-based methods learn a value function that estimates the expected return for each possible action-state pair. This function is then used to create a policy function that acts greedily. Linear learners are one group of value-based methods. An advantage of linear learners is that they have strong theoretical convergence guarantees. Additionally, they exhibit quick training time in gradient descent calculations, due to being simple functions. However, there is no guarantee that the features extracted by them are optimal. This is empirically shown by [1] where the best linear learner gets 5.2 points in Breakout. Another approach to value-based methods is to use a Deep Q-learning Network (DQN) [9]. An advantage of DQN is that it is off-policy. This means that the policy updates can use samples from a different policy, which decreases training as fewer samples are required. DQN uses only raw pixels as the input for its neural networks, which is beneficial as neural networks are excellent feature extractors. However, they require a lot of data for training. Additional drawbacks of DQN are that it tends to be unstable during training and has limited theoretical convergence guarantees when compared to linear learners. To overcome these limitations, DQN has been subject to various enhancements to the base algorithm [8], such as Double DQN (DDQN), Dueling Double DQN (DDDQN), Prioritised Experience Replay (PER), and Rainbow [6], [13], [16], [17]. DQN scored 401.2 in Breakout over ten million game frames [8]. In the same period, DDQN achieved a result of 418.5. Despite the alleged improvements, DDDQN only scored 345.3 [6]. PER achieved 381.5 and Rainbow achieved 417.5. These results suggest that DQN and its extensions are a viable candidates for the chosen problem.

Policy-based methods do not learn a value function, but directly learn a policy. Unlike value-based methods, they allow a stochastic policy to be learnt. The downside of policy-based methods is that they are all on-policy, meaning they require data from the policy they update instead of using data from old policies. An example of a policy-based method is REINFORCE [19]. A disadvantage of REINFORCE is that is does not bootstrap. This means that it uses full episodes to determine its policy gradient that is used for updating the policy. As a result, the training time is increased drastically. Furthermore, as REINFORCE is likely to converge to non-global maxima, it could require multiple runs of the algorithm to fine the global maximum. From this, it can be concluded that REINFORCE is not an effective method for the chosen domain.

Actor-critic methods use a combination of policy function and a value function, each of which has its own update rule. An advantage of actor-critic methods is that they can bootstrap, which

results in shorter training time. Some examples of actor-critic methods include Deep Deterministic Policy Gradient (DDPG), Soft Actor-Critic (SAC), Twin Delayed DDPG (TD3), Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO) [3], [5], [7], [14], [15]. DDPG, SAC and TD3 are not suitable for the chosen environment because they only operate on a continuous action space. On the other hand, PPO and TRPO can work on both continuous and discrete action spaces. One disadvantage of the two algorithms is their instability as shown in [20]. Another drawback of TRPO is that it is much more complicated to implement and takes longer to train than PPO as it uses second-order derivatives. In Breakout, TRPO scored 34.2 over 500 iterations [14] and PPO scored 274.8 after 40 million game frames. For this reason, PPO is a candidate since it has been empirically shown to be able to perform at a high standard.

3 Method

From the Artificial Intelligence Breakout leaderboard [11], it is clear that DQN and its extensions have empirically performed at the highest calibre for the Breakout environment. On top of that, the research done in the background section suggests that DQN and its extensions require less training time than other viable methods. Therefore, the chosen methods are the DQN algorithm and a couple of its extensions (discussed below). The goal is to replicate the experiments done by Hessel, Modayil, Van Hasselt, *et al.* to verify the results discussed in the background section.

3.1 DQN history

DQN is a function approximated method that is based on the tabular method Q-learning. Convolutional neural networks are used to approximate the state-value function, and they iteratively improve its approximation using gradient descent, calculated with the mean squared error loss function. Naive DQN algorithm was introduced in the 2013 paper [9], with improvements to form the vanilla DQN algorithm in the 2015 paper [8], whose pseudocode can be found in appendix E. These improvements included:

1. Experience replay buffer which trains the agent using samples of past experiences to decorrelate the agent's experiences.
2. Huber loss function instead of mean squared error in order to decrease the effect any one large prediction has on changing all the network's weights.
3. Fixed target network which get temporal difference target values in order for updates to move towards stationary target values.

3.2 Preprocessing inputs

Since the agent cannot determine the direction and velocity of the ball from a single frame, frame stacking is introduced, in which four frames are stacked together to represent a singular state. To decrease the computational load on hardware, a series of image preprocessing methods were implemented. Firstly, frame skipping was used to only consider every fourth frame, reducing the computational load by 75%. Since colours in the images do not impact the optimal strategy, the images are converted to greyscale. This reduces the memory required by 66.7% because each pixel is only using one colour channel instead of three. Finally, each 210×160 image is down-scaled to 84×84, which does not hinder functionality and decreases the memory required by a further 79%. [8] recommends to clip the rewards to be between -1 and 1. This means that the agent will focus on getting many rewards, rather than focusing on getting the highest possible reward. To speed up training, an auto-shoot functionality was implemented which automatically respawns the ball after the agent misses it.

3.3 Update rule

Given a sample transition from state $s \in S$ to state $s' \in S$ by taking action $a \in A(s)$ with reward $R(s,a)$, the target value is $y = \begin{cases} R(s,a) & \text{if } s' \text{ is terminal} \\ R(s,a) + \gamma \max_{a'} \hat{Q}(s',a') & \text{otherwise} \end{cases}$

and the actual value is $\hat{y} = Q(s, a)$, where $Q, \hat{Q}$ are the original and target networks respectively, and γ is the discount factor.

The gradient descent step (update rule) then uses the Huber loss between y and $\hat{y}$ defined as
$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$ where $\delta = 1$ in this case.

3.4 Network architecture

The design of the neural network follows the architecture described in [8]. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by preprocessing. The architecture of the neural network is described in Table 1. The output layer is a fully-connected linear layer with a single output for each valid action in the environment with four in total.

Table 1: Architecture of Neural Network Used in DQN

Layer	(Stride, Filter, Kernel)	Activation Function
Convolutional Layer 1	$(4, 32, 8 \times 8)$	ReLu
Convolutional Layer 2	$(2, 64, 4 \times 4)$	ReLu
Convolutional Layer 3	$(1, 64, 3 \times 3)$	ReLu
Fully Connected Layer 1	512 Rectifiers	ReLu
Fully Connected Layer 2	4 Rectifiers	None

3.5 Extensions to DQN

The DQN algorithm displays a few issues, some of which are addressed by the extensions Double DQN and Dueling Double DQN. For this reason, Double DQN and Dueling Double DQN will be implemented as extensions.

3.5.1 Double DQN

One shortcoming of DQN is the overestimation of Q-values. This is because the Q-values are only calculated for explored states and their neighbours. As a result, the network enters a loop where non-optimal actions taken increase their Q-values which lead them to be selected more frequently. Hence, there is a need to decouple the actions selection from the Q-values generation [16]. DDQN attempts to do that by choosing the action using the original network and evaluating the Q-value using the target network. This is done by calculating the target value used in the update rule as:
$y = \begin{cases} R(s, a) & \text{if } s' \text{ is terminal} \\ R(s, a) + \gamma \hat{Q}(s', \text{argmax}_{a'}, Q(s', a')) & \text{otherwise} \end{cases}$
where $s, s' \in S$ are the previous and current state respectively, with action $a \in A(s)$ the action taken to yield s' from the environment, $Q, \hat{Q}$ are the original and target networks respectively, and γ is the discount factor.

3.5.2 Dueling Double DQN

Another drawback of DQN is the need to explore all state-action pairs to determine whether a given state is advantageous or not. However, some states are always undesirable to be in, no matter which action is taken. Therefore, exploring all state-action pairs is not necessary. As a result, the need arises to decouple the state values from action-state values [17]. DDDQN calculates the value of a state separately to the value of the action taken in that state, using the formula
$Q(s, a) = V_1(s) + (V_2(s, a) - \frac{1}{A(s)} \Sigma_{a' \in A(s)} V_2(s, a'))$
where $s \in S$ is a given state, $a \in A(s)$ is an action, V_1 represents the value of being in state s, and V_2 represents the added value of action a in state s. This requires a new network architecture, which is as follows: The input size of the network is $84 \times 84 \times 4$ and output size of the first stream is 1 and the output size of the second stream is 4. Both V_1 and V_2 are passed through common convolutional layers. Following that, each one goes to its respective neural network, as shown in Table 2.

Table 2: Architecture of the Neural Network Used in DDDQN

Layer	(Stride, Filter, Kernel)	Activation Function
Convolutional Layer 1	$(4, 32, 8 \times 8)$	ReLu
Convolutional Layer 2	$(2, 64, 4 \times 4)$	ReLu
Convolutional Layer 3	$(1, 64, 3 \times 3)$	ReLu
Fully Connected Value Layer 1	512 Rectifiers	ReLu
Fully Connected Value Layer 2	1 Rectifier	None
Fully Connected Action Layer 1	512 Rectifiers	ReLu
Fully Connected Action Layer 2	4 Rectifiers	None

4 Results

DQN and its add-ons DDQN and DDDQN were trained on the same set of final hyperparameters 5, found in Appendix A, which were inspired by the DQN and DDQN papers [8], [16] as well as from results given from empirical tests as presented in 7, found in Appendix C. The agent's training was halted at 4 millions frames to guarantee fair comparison.

As seen in figure 1, DQN performed the best, achieving higher scores, while DDQN performed the worst, with DDDQN outperforming it. Training speeds and episode counts varied for each agent, with DDDQN taking significantly longer to train at 13 hours and around 3750 episodes, followed by DDQN at 12 hours and around 4000 episodes and DQN at 11 hours and around 3500 episodes.

On further inspection of results presented in the DDQN and DDDQN papers [16], [17], DDQN surpassed DQN in Breakout when using tuned hyperparameters. More specifically, a decreased target network update rate and the already utilised lower minimum ε of 0.01. DDQN was trained on the tuned hyperparameters 6, achieving better scores than its previous DDQN counterpart, as well as surpassing DDDQN. Tuned DDQN took significantly longer to train at 15 hours, as well as the longest episode count at around 5000 episodes.

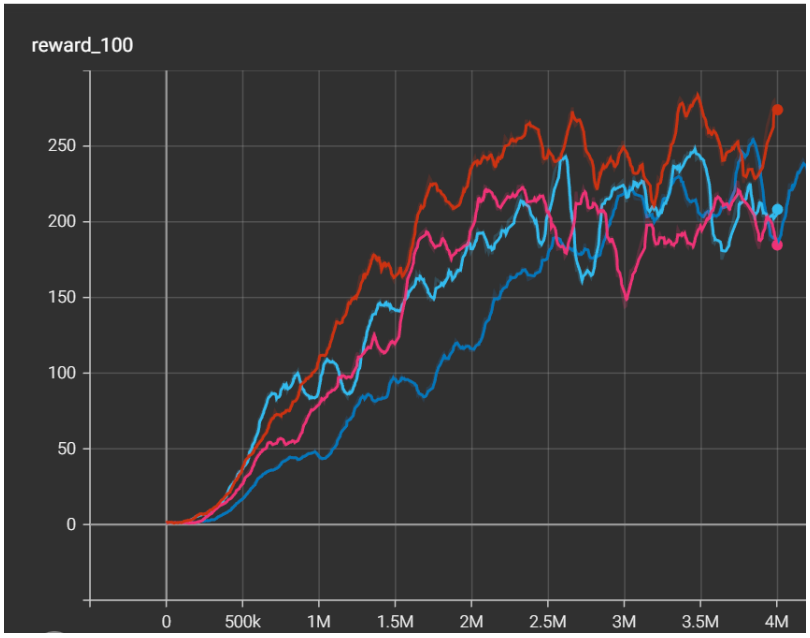

Figure 1: Comparison of smoothed mean rewards over 100 episodes during training of DQN (scarlet orange), DDQN (pink) and DDDQN (light blue) trained on same set of hyperparameters, and Tuned DDQN (dark blue) on tuned hyperparameters, smoothed out by 0.6. The X-axis represents the number of frames and the Y-axis represents the average game scores.

Following DQN's evaluation of performance of the models [8], the model was tested for 30 episodes with an ε of 0.05. Additionally, the same test was performed on ε of 0.001 in accordance with the DDQN paper [16]. These tests had a limit of 18000 frames per episode to avoid loops. The average of these episodes with the different ε values, is presented in Table 3. These results were compared to two separate baselines. The random baseline and the professional human baseline. The random baseline was set by random inputs and achieved a score of 1.4. The professional human baseline was set by a human in a controlled environment after two hours of training and scored 31.8, as taken from [8]. The highest theoretical score in Breakout is 432.

Table 3: Table showing results of DQN, DDQN, DDDQN and Tuned DDQN algorithms for $\varepsilon = 0.05$ and $\varepsilon = 0.001$ in the conditions described above

Agent	DQN	DDQN	DDDQN	Tuned DDQN
$\varepsilon = 0.05$	235.6	163.9	163.3	203.3
$\varepsilon = 0.001$	361.5	308.0	232.1	317.9

5 Discussion

The results obtained from all the agents, as shown in Table 3, demonstrate learning. All agents significantly surpassed the random agent and the professional human player. There is still opportunity for improvement, as the highest achievable score is 432 in Breakout.

The agents did not achieve the same results as those described in the DQN, DDQN and DDDQN papers [8], [16], [17]. This likely occurred due to constraints on training time, as there was only one personal computer to perform the training on. Another factor that affected the results is the choice of hyperparameters that were most suitable for the given computational constraints.

Figure 1 shows significant drops, predominantly seen between frames 2.5 million to 3.5 million in all the methods. One possible explanation for this drop could be that the agents are experiencing catastrophic forgetting. A possible solution to this problem would be to increase the size of the experience replay buffer, which was not possible in these tests due to hardware limitations.

Figure 1 also displays signs of oscillations predominantly between frames 2.5 million to 4 million, in all methods except Tuned DDQN. An explanation to this oscillation could be that the original network's Q-values are not moving towards a stationary target. A solution for this would be decreasing the update rate of the target network, as performed on the Tuned DDQN.

Figure 1 shows that the agents' learning rates roughly follow a sigmoid curve, where learning rates are slow at the beginning, quick in the middle and slow at the end. The agents' learning rates at the beginning are similar, which follows literature results [6].

Furthermore, the DQN and DDDQN results align with literature [17], where DDDQN is worse than DQN. This is surprising as DDDQN is supposed to be an improvement to DQN. It is worth noting that this is an environment specific issue, rather than a general one, as DDDQN outperforms DQN in most Atari environments. One reason DDDQN underperforms in Breakout could be due to the type of features being extracted by the convolutional neural networks. If the features extracted are highly correlated, then the agent might struggle to differentiate between which feature belong to the state value stream and which belong to the action value stream. This could lead to some features being incorrectly labelled as having more impact on one stream than the other, which would lead to a worse policy learnt.

However, comparing the final results of DDQN and DQN, the results differ from the ones presented in the DDQN paper [16]. In the paper, DDQN outperforms DQN, whereas the evaluation conducted here suggests that DQN is the best performer. Tuned DDQN achieves results closer to the reported scores, perhaps due to decreased oscillations and less signs of catastrophic forgetting. It may be possible for the Tuned DDQN to surpass DQN, either with increased training time or further tuning of its hyperparameters.

The agents showed a variety of strategies during their training. The first strategy the agents learnt was to return the paddle to the centre of the screen after hitting the ball. This gives the agent the

shortest average possible path to any location in the screen as it cannot predict where the ball is going to land. This allows the agent to play more consistently compared to an agent positioned at the corners. Alongside this strategy, the agent learnt to bounce the ball in a specific direction so that the ball will return to the paddle's position, reducing the required movement of the paddle. The next strategy the agent learnt was to tunnel through the bricks to reach the top of the screen as quickly as possibly. This leads to the agents trapping the ball between the top layer and the ceiling, giving it a high chance to break many high-value bricks in a single bounce. This behaviour is incentivised by the high γ value of 0.99, as it prioritises actions which lead to a better outcome in the far future. This is irrespective to the higher layers of bricks giving more points, which would only incentivise this behaviour further, as the rewards are cropped to 1. Lastly, the agents sometimes bounce the ball off the walls in order to get the ball in otherwise unreachable locations, such as on top of the bricks.

6 Future work

Future work involves training the agents multiple times, averaging the results to ensure that the current results are not outliers. Furthermore, to increase the score achieved, the hyperparameters will be fine-tuned and the training time will be increased. One method to increase the size of the experience replay buffer hyperparameter while avoiding the use of better hardware is storing individual frames in the experience replay buffer, only restacking them once they are being sampled. This comes with the disadvantage of more computation, and so the training time will increase.

Furthermore, other algorithms could be compared against the current algorithms. [6] suggest that Noisy DQN and Distributional DQN (C51) add-ons improve the score, as well as the combinations of all of the add-ons: Rainbow. Implementing these add-ons will facilitate the comparison of more DQN extensions.

Finally, the Gym library offers higher difficulty game modes for Breakout. Training the agents on these game modes will showcase the scalability of the different agents on more complex environments.

7 Personal experience

Initially, we decided as a group that we wanted to create an agent that could solve a problem from the Atari environment. We settled on Breakout, mainly because all group participants had previously played the game and had some level of nostalgic connection to it. Another Atari game we considered is Skiing, but we deemed it unsuitable since it is a sparse reward environment.

The research conducted in the background brought us to DQN and its extensions as the algorithm choice. We then decided to partition into two groups. One group focused on the practical implementation of the model using pair programming, whereas the other primarily worked on understanding the theory behind the algorithm and its extensions. At the start of the implementation we realised we needed a simpler environment to get comfortable with the Gym library. For this reason, we began by implementing DQN on the Cart Pole environment. Once the main principles of the Gym environment were understood, the implementation of DQN was converted to Breakout. Once DQN was successfully implemented, it was not very difficult or time-consuming to add DDQN and DDDQN because the code was modular and well structured.

One difficulty encountered was the necessary usage of a GPU to which only one member had access to, meaning we had limited training times. We therefore used Google Colab [4] for shorter tests while simultaneously running longer tests on that member's personal computer. Larger tests could not be ran on Google Colab due to the limited memory it provides.

The two subgroups communicated throughout, teaching each other insights learnt in their respective processes.

References

[1] 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*, vol. 47, pp. 253–279, 2013.

[2] J. Evans, *Reinforcement Learning - Week 8: DQN Algorithm, lecture slides*, 2022. [Online]. Available: `https://moodle.bath.ac.uk/mod/page/view.php?id=1146241` (visited on 01/07/2023).

[3] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*, PMLR, 2018, pp. 1587–1596.

[4] Google Research. [Online]. Available: `https://colab.research.google.com/` (visited on 01/07/2023).

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*, PMLR, 2018, pp. 1861–1870.

[6] M. Hessel, J. Modayil, H. Van Hasselt, *et al.*, "Rainbow: Combining improvements in deep reinforcement learning," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[10] OpenAI. [Online]. Available: `https://www.gymlibrary.dev/` (visited on 12/27/2022).

[11] PapersWithCode. [Online]. Available: `https://paperswithcode.com/sota/atari-games-on-atari-2600-breakout` (visited on 12/27/2022).

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

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

[14] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, PMLR, 2015, pp. 1889–1897.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

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

[17] 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*, PMLR, 2016, pp. 1995–2003.

[18] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[19] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[20] Z. Zhang, X. Luo, T. Liu, *et al.*, "Proximal policy optimization with mixed distributed training," in *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*, IEEE, 2019, pp. 1452–1456.

Appendices

Appendix A: Hyperparameters

Table 4: First set of hyperparameters

Hyperparameter	Value
Discount Factor Gamma	0.99
Minibatch size	32
Experience replay buffer size	100000
Target Network update frequency	3000
Update frequency	4
Learning rate	0.0001
Initial exploration epsilon	1
Initial exploration epsilon	0.01
Replay buffer start size	5000
No op max	30

Table 5: Final set of hyperparameters

Hyperparameter	Value
Discount Factor Gamma	0.99
Minibatch size	32
Experience replay buffer size	100000
Target Network update frequency	10000
Update frequency	4
Learning rate	0.0001
Initial exploration epsilon	1
Initial exploration epsilon	0.01
Replay buffer start size	5000
No op max	30

Table 6: Tuned DDQN set of hyperparameters

Hyperparameter	Value
Discount Factor Gamma	0.99
Minibatch size	32
Experience replay buffer size	100000
Target Network update frequency	20000
Update frequency	4
Learning rate	0.0001
Initial exploration epsilon	1
Initial exploration epsilon	0.01
Replay buffer start size	5000
No op max	30

Appendix B: Results

These results were produced on a Lenovo Legion 5 Pro with 32GB of memory, with a 3070 mobile GPU and a Ryzen 5800H CPU . Using other hardware may affect the training speed of the agents, as well as the memory requirement for the chosen size of the experience replay buffer. The results obtained can be replicated using the final set of hyperparameters 5, up to a 4 million frame limit. To obtain the Tuned DDQN results, utilise the tuned hyperparameters 6.

X-axis represents number of frames and Y-axis represents average game scores. All graphs are smoothed by a value of 0.6.

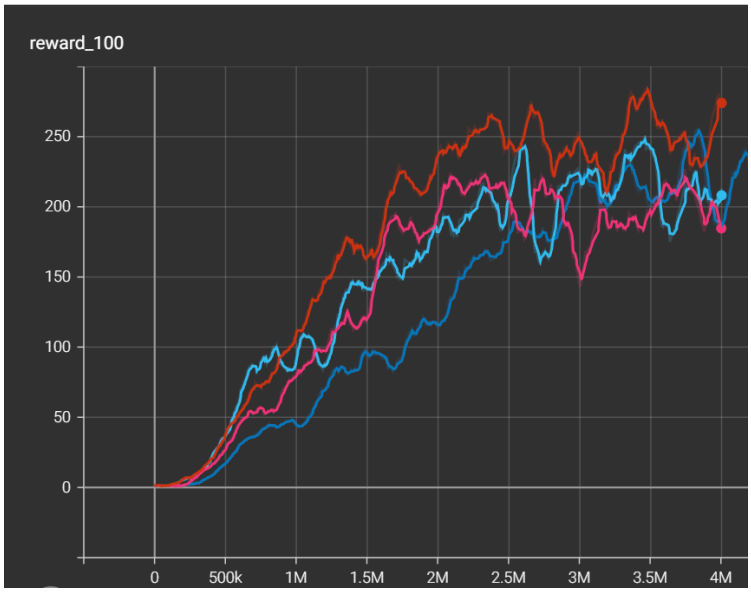

Figure 2: Comparison of smoothed mean rewards over 100 episodes during training of DQN (scarlet orange), DDQN (pink) and DDDQN (light blue) trained on final set of hyperparameters 1 and tuned DDQN (dark blue) on tuned hyperparameters 6

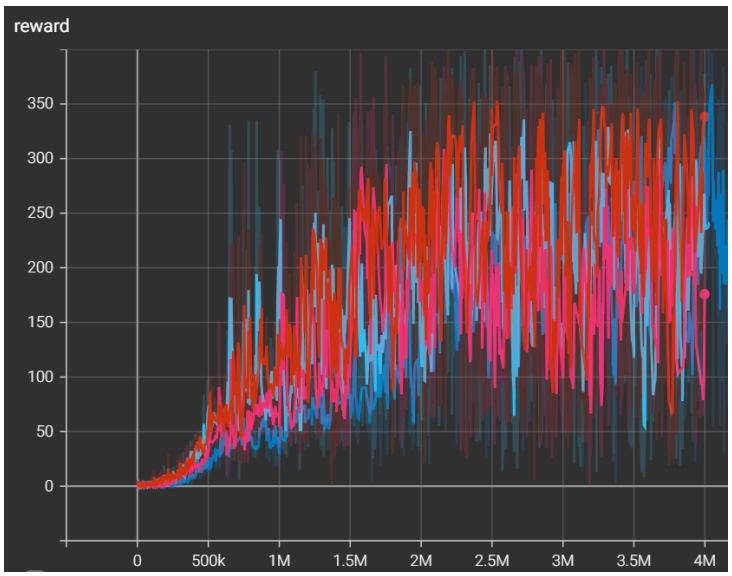

Figure 3: Comparison of smoothed rewards during training of DQN (scarlet orange), DDQN (pink) and DDDQN (light blue) trained on second set of hyperparameters 5 and DDQN (dark blue) on tuned hyperparameters 6

Appendix C: Hyperparameter tuning

For hyperparameters tuning, the agents were trained up to a 3550 episode limit. Initial test was with a 3k target network update rate, and after observing better results with a 10k update rate, hyperparameter tuning was performed in the second set of hyperparameters. Hyperparameters tested were target network update rate (3k vs 10k), learning rate (0.0001 vs 0.00025), minimum epsilon (0.01 vs 0.1), terminal state consideration during life loss, and clipping vs no clipping rewards. These results led us to create the final set of hyperparameters 5.

Figure 4: Comparison of smoothed mean rewards over 100 episodes during training of DQN (dark blue), DDQN (scarlet orange) and DDDQN (light blue) trained on first set of hyperparameters 4

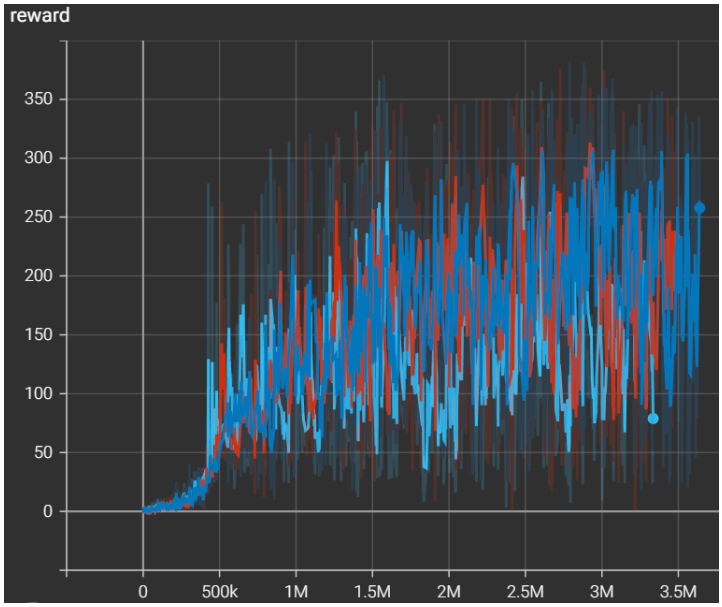

Figure 5: Comparison of smoothed rewards during training of DQN (dark blue), DDQN (scarlet orange) and DDDQN (light blue) trained on first set of hyperparameters 4

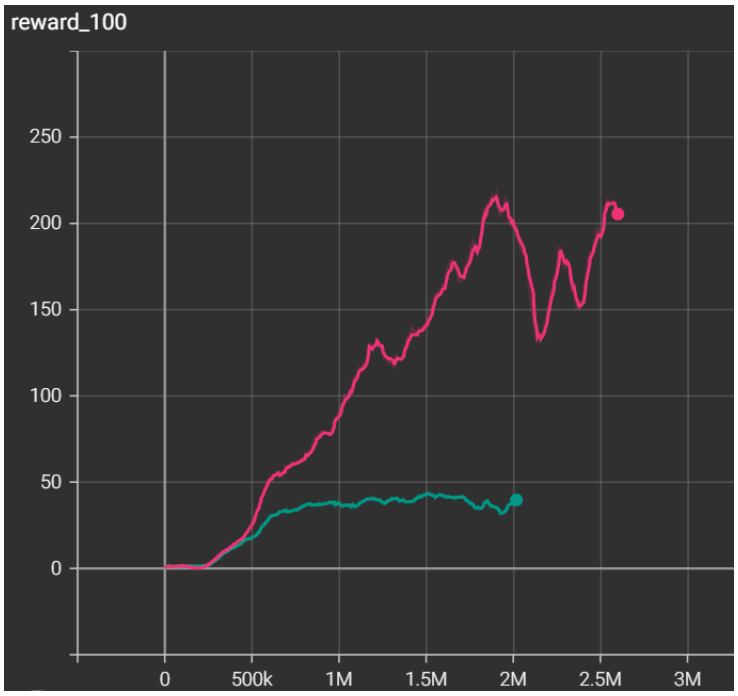

Figure 6: Comparison of smoothed mean rewards over 100 episodes during training of DQN (pink) trained on final set of hyperparameters 5 and DQN (green) with the same hyperparameters except learning rate of 0.00025

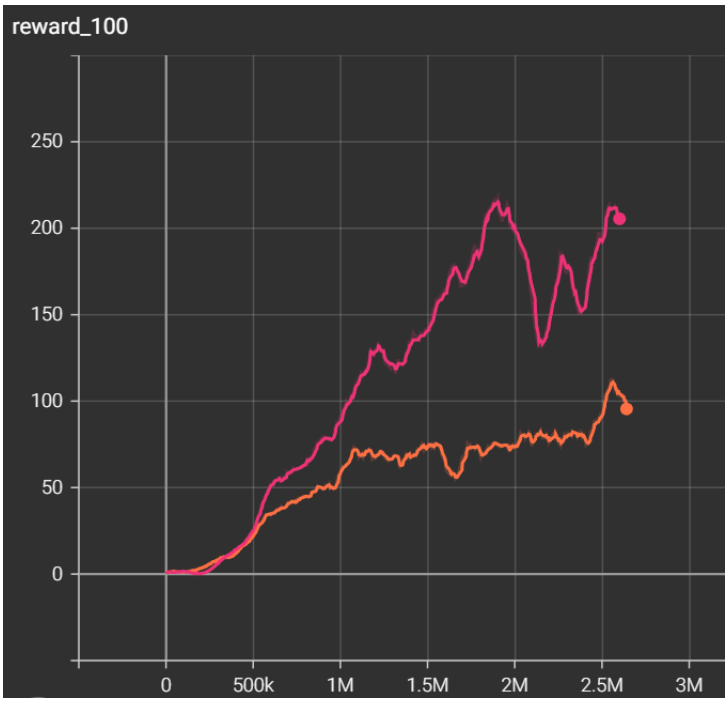

Figure 7: Comparison of smoothed mean rewards over 100 episodes during training of DQN (pink) trained on final set of hyperparameters 5 and DQN (orange) with the same hyperparameters except minimum epsilon of 0.1.

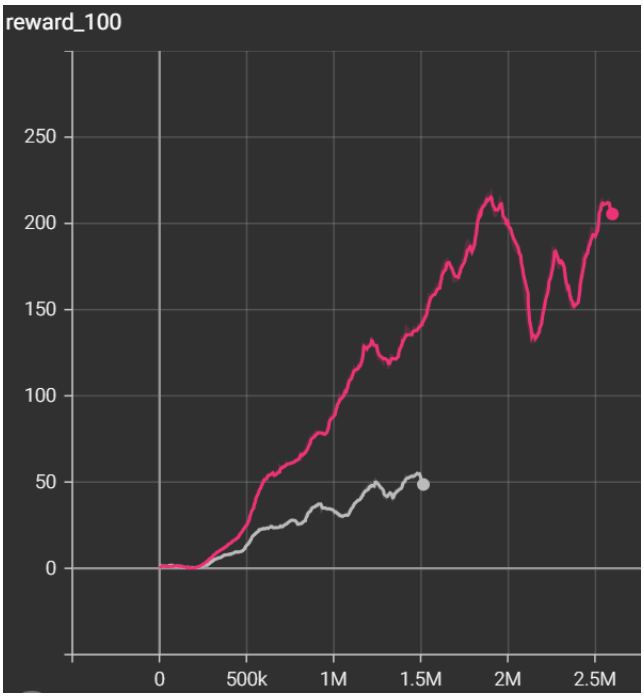

Figure 8: Comparison of smoothed mean rewards over 100 episodes during training of DQN (pink) trained on final set of hyperparameters 5 and DQN (grey) with the same hyperparameters except not considering life loss as terminal in experience replay buffer.

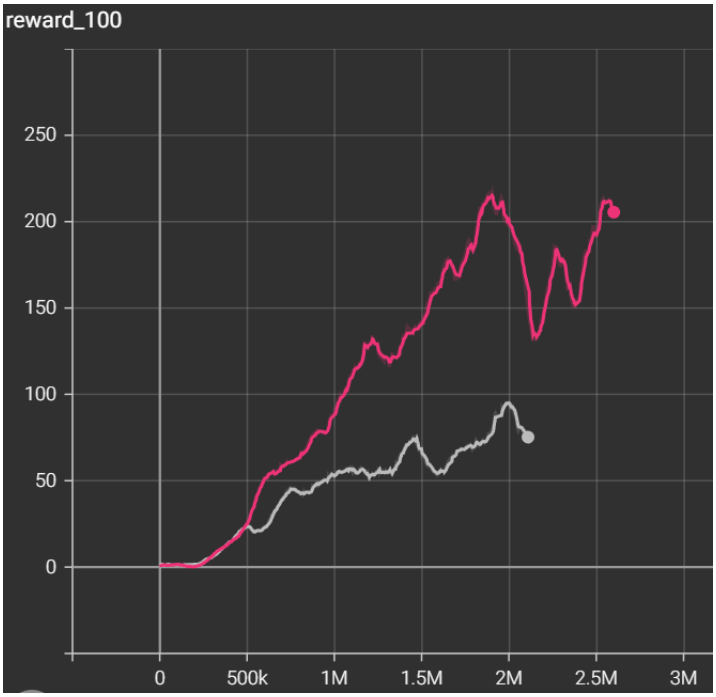

Figure 9: Comparison of smoothed mean rewards over 100 episodes during training of DQN (pink) trained on final set of hyperparameters 5 and DQN (grey) with the same hyperparameters except no clipping of rewards.

Appendix D: Video demonstration

Video demonstration of the agent's performance can be found here: `https://www.youtube.com/watch?v=ppTR7J1miM0`

Appendix E: DQN pseudocode

Algorithm 1 DQN algorithm

Initialise replay memory D to capacity N
Initialise action-value network $\hat{q_1}$ with parameters $\theta_1 \in \mathbb{R}^d$ arbitrarily
Initialise target action-value network $\hat{q_2}$ with parameters $\theta_2 = \theta_1$
for each episode **do**
  Initialise S
  Choose action A in state S using policy derived from $\hat{q_1}(S, \cdot, \theta_1)$
  Take action A, observe reward R and next-state S'
  Store transition (S, A, R, S') in D
  for Store transition (S_j, A_j, R_j, S'_j) in minibatch sampled from D **do**

$$y = \begin{cases} R_j & \text{if } S'_j \text{ is terminal} \\ R_j + \gamma \text{max}_{a'} \hat{q_2}(S'_j, a', \theta_2) & \text{otherwise} \end{cases}$$

    $\hat{y} = \hat{q_1}(S_j, A_j, \theta_1)$
    Perform gradient descent step $\nabla_{\theta_1} L_\delta(y, \hat{y})$
  end for
  Every C time-steps, update $\theta_2 = \theta_1$
end for

This pseudocode was taken from [2].