Deep Reinforcement Learning to Play Breakout

Madeline Kelly
Department of Computer Science
University of Bath
mk822@bath.ac.uk

Joe Molyneux-Saunders
Department of Computer Science
University of Bath
jmpms20@bath.ac.uk

Joe Rosenberg
Department of Computer Science
University of Bath
jer60@bath.ac.uk

1 Introduction

Reinforcement learning is an area of machine learning that studies methods for learning how to act optimally using a scalar reward signal as feedback. In this report, we apply reinforcement learning methods to find a control policy for the Atari game Breakout. We implement and test four variations of Deep Q-Learning [7] on Breakout. Our experiments show that of the four methods, the variation that showed the best learning and performance was Deep Q-Learning with Dueling Networks.

2 Problem Description

The game's setting consists of a ball, a paddle and six rows of eighteen bricks lining the top of the screen. Walls cover the top and sides of the screen, but not the bottom. The ball moves around the screen in a straight line and bounces off the other objects in the play area. When the ball hits a brick, the brick breaks and the player receives points. Bricks in higher rows are worth more points.

The objective of the game is to score as many points as possible before the game ends. Towards this end, the player must move the paddle to stop the ball from passing off the bottom of the screen, as the game terminates after the ball leaves the screen five times.

The Atari 2600 renders images from the game at a rate of 60 frames per second, and each frame is a 160×192 resolution image. There are four effective actions in Breakout, moving the paddle left or right, releasing the ball and taking no action. The ball must be released to begin the game and each time after the ball leaves the screen.

The number of possible game states is very large, owing to the large number of configurations that each element of the game can be in. Assuming that we can't break a brick in a column without breaking all of the bricks below it[1], each column of bricks has 7 possible states. Since there are 18 columns, the number of brick states alone is at least $7^{18} \approx 1.6 \times 10^{15}$. The number of game states is even higher than this due to the many different velocities and positions of the ball and positions of the paddle.

Due to the very large state space, we can not apply tabular reinforcement learning methods to this problem. Storing Q-values for every state would require prohibitive amounts of memory, and visiting enough states to make good Q-value estimates would take a prohibitive amount of time.

[1]This assumption gives a drastic underestimate of the number of possible brick states. This is because there are many possible states that are ignored under this assumption. For example, it is possible to break bricks in the middle of a column if a neighbouring column has been cleared.

Project report for CM50270 Reinforcement Learning at the Department of Computer Science, University of Bath, UK. May 2020.

3 Approach

To approach this problem using reinforcement learning, we must model it as a Markov decision process. We use frame data (images) from the game to build our state representation. However, we must be careful to maintain the Markov property with our chosen state representation. Using a single frame as our state representation is not suitable, as it does not contain information about the direction that the ball is travelling in. If this information is not included in a state, the transition probabilities from a state will depend on previous states, as they give information about the direction that the ball is travelling in. To get around this, we choose to construct a state representation from multiple frames as in [7]. Further details are given in Section 3.1.

Our action space contains four actions: do nothing, move the paddle left, move the paddle right, and release the ball. We call these actions 'noop', 'left', 'right' and 'fire'.

We use the natural choice of reward function: the number of points received during a step. Our desired objective (maximise the number of points scored in an episode) and the agent's objective is then perfectly aligned.

There are two main approaches to reinforcement learning, value based methods and policy based methods [1]. Previous work on Atari games shows excellent performance of different variants of the value-based method Deep Q-Learning, whereas the policy gradient method A3C is far less successful [6]. For this reason, we focus on Deep Q-Learning to address this problem. We used four different variations of Deep Q-Learning to address this reinforcement learning problem. These were Vanilla Deep Q-Learning, Dueling Deep Q-Learning, Double Deep Q-Learning and Double Deep Q-Learning with prioritised experience replay.

We choose to use a deep convolutional neural network to approximate the Q-function as in [7]. The reason being is that it automates the process of feature engineering. Deep neural networks are able to use their hidden layers to construct features from the raw state representation [3]. Because the hidden layer weights are optimised to maximise accuracy, we end up with features that are useful for making accurate Q-value predictions.

3.1 Pre-processing

The images are pre-processed using the same method as Mnih et al. (2015) [7]. This method is outlined in Algorithm 1. Steps 1-2 are used to remove any flickering in the images. This is necessary so that the state representation captures any flashing artifacts present in the game. Steps 3-5 reduce the dimensionality of the input state and Step 6 stacks the pre-processed frames so that they are ready for input into the network.

In our experiments, we set the frame queue size n to 4 and set the frame skip m to 4. Hence, for each selected action, the action is repeated in the environment for $m = 4$ frames. The max pool of the most recent two frames is taken and the resultant frame is converted to gray-scale, down-sampled and cropped. Lastly, the pre-processed frame is added to the Frame Queue $\mathcal{F}$. The current Frame Queue $\mathcal{F}$ is then returned as the current state. To this effect, the state passed to the agent is the current pre-processed frame plus the previous three pre-processed frames. The full algorithm is outlined in Appendix D.

3.2 Basic Network Architecture

The basic network, used as a basis for all four algorithms explored in this report, is visualised in Appendix A. The network has three convolutional layers, each with a rectified linear unit activation. The first layer has 32 filters of size 8 x 8 with stride 4. The second layer has 64 filters of size 4 x 4 with stride 2. The third layer has 64 filters of size 3 x 3 with stride 1. This output is then flattened and input into a fully connected layer of 512 units with a rectified linear unit activation. The output layer is a fully connected layer with linear activation consisting of four units, one for each action. This is the same network architecture used by Mnih et al. (2015). The network weights were initialised according to the method described by He et al. (2015) [5].

3.3 Basic Hyper-parameters

The parameter ϵ is used in ϵ-greedy action selection. It determines the probability with which a random action should be selected. We choose the same piecewise-linear annealing strategy for ϵ as in Mnih et al. (2015) [7], where ϵ is first decreased rapidly from 1 to 0.1, and then decreased slowly from 0.1 to 0.01. This is a particularly important choice for Breakout as the ball does not move until the action 'fire' is taken. Due to random initialisation of the network weights at the start of training, the agent may initially not choose 'fire' for many time steps unless ϵ is large. This can lead to very slow training at the start, as episodes will run for a long time with little useful feedback if the ball is not moving. For Double DQN, it is also important that the agent is initially allowed to explore actions randomly before the target network is updated and it learns to start each game and round with the 'fire' action.

We run each algorithm for 5 million iterations. We would have liked to run them for longer, but were unable to do so due to time and resource constraints.

The replay size and minibatch size are important since we want the transitions selected at each update to be uncorrelated. We use the same minibatch size as Mnih et al. (2015) [7]. However, we chose a replay memory size of 100,000 where they used a replay memory of size 1,000,000. This was because of memory limitations of the system used for training. Since we chose to make our replay memory ten times size smaller than Mnih et al. (2015), we also decided to make the replay start size (the number of transitions before we start learning) ten times smaller too. Another reason we did this is because we did not have enough time or resources to run the training for longer, so we wanted the learning to start quicker.

The discount factor used was the same as by Mnih et al. (2015) [7], $\gamma = 0.99$. Hessel et al. (2018) [6] used a learning rate of 0.0000625. Because we were training for fewer iterations we needed learning to happen quicker, so we chose to make the learning rate slightly higher at 0.00025. The target network was set to update every 10,000 iterations as in Mnih et al. (2015) [7]. Finally, the maximum length of an episode (where episode is defined as a full game) was set to 10,000 iterations. This was to prevent the training being slowed by long episodes. A full list of the hyperparameters used for training can be found in Appendix C.

3.4 Dueling Deep Q-Learning

A dueling network approximates the value of the state independently to the action taken. This is useful in some cases when the knowledge of the value of an action is not necessary. In particular this idea applies to breakout, since the action taken often doesn't matter until the ball is near the paddle.

Dueling Deep-Q learning is almost identical to Vanilla Deep Q-learning, the only difference is in the network architecture. The duelling network estimates the state value function and the state-dependent action advantage function. The estimated state value and action advantage value can then be combined to estimate the q-values for the state-action pairs. The convolutional layers are identical to that of the original network except that next, the network splits into two streams of fully connected layers consisting of 512 rectified linear units. One of these streams is connected to an output layer of linear units the same size as the number actions of available to approximate the action advantage. The other stream is connected to one linear unit that approximates the state value. These are then combined to estimate the q-values [10]. A visualisation of the network architecture applied to breakout can be seen in appendix B.

An important part of the dueling network is the method used to combine the action advantage and the state value to estimate the q-value. For the purpose of this report we choose to use approach chosen by Wang et al. described by equation 1.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \tag{1}$$

where Q is the Q-function, V is the state value function, A is the action advantage function, s is the state, a is the action, $\mathcal{A}$ is the set of possible actions and θ is the set of convolutional network parameters, and α and β are the parameters corresponding to the separate streams of the network.

3.5 Double Deep Q-Learning

In Deep Q-Learning, the $\max$ operator in the Q-value update means that there can be large over-estimates of the action values from the next state. By using the same function to select and evaluate the action, the $\max$ operator is more likely to select over estimated values and this is what can lead to value estimates that are too large [4].

The idea behind Double Deep-Q Learning is to use a double estimator that estimates the maximum of a set of stochastic values. This is done by breaking the maximisation function into action selection and action evaluation by using two different value networks [4]. Van Hasselt, Guez and Silver (2016) propose using the target network for value estimation and the online value network for action selection [9]. Let Q denote the Q-function, R the reward, S the state, γ the discount factor, θ the value network parameters and θ' the target network parameters. Then the Deep Q-Learning update can be written as:

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t) = R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t); \theta_t) \tag{2}$$

and the Double Deep-Q Learning update as:

$$R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{3}$$

Aside from the change in the network update, the rest of the algorithm follows the same structure as Vanilla Deep Q-Learning algorithm.

3.6 Double Deep Q-Learning with Prioritised Experience Replay

Vanilla Deep Q-Learning assumes information gained from each transition is equal and therefore samples from the replay memory uniformly. However, this is often not the case. A natural extension to Double Deep Q-Learning is to add a priority metric to the replay memory such that *"more important"* transitions are revisited more often than *"less important"* transitions. Ideally, we would like to revisit transitions that we *"learn more"* from, more frequently. This can be achieved using a prioritised experience replay [8].

In order to prioritise which transitions are revisited more often than others we need to evaluate an appropriate priority metric for each. Schaul et al. (2015) [8] select the temporal-difference (TD) error as their priority metric. The TD error of transition $(S_{j-1}, A_{j-s}, R_j, \gamma_j, S_j)$ is given by the following:

$$\delta_j = R_j + \gamma_j Q_{target}(S_j, argmax_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1}) \tag{4}$$

A large TD error is analogous to a transition which we can expect to learn a lot from. The authors also suggest other potential priority measures however they conclude that in near-deterministic environments (like ours) there is little change in performance over TD error. As such, we chose to use the TD error as the priority metric in our experiments.

Since TD errors are only updated when transitions are sampled, transitions with small TD error are likely to not be revisited again before they are removed from the replay memory entirely [8]. This invokes lack of diversity in the sampled transitions potentially leading to over-fitting. Hence, it is necessary to introduce a stochastic sampling method to find a middle ground between greedy and uniform sampling. Thus, the probability of sampling a transition is given by:

$$\mathcal{P}(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5}$$

where α is a hyper-parameter which tunes the sampling probability between uniform and greedy e.g. $\alpha = 0$ is equivalent to uniform sampling and p_j is given by $p_j = |\delta_j| + \epsilon$. Two variants of p_j are presented by the authors, we use proportional prioritisation over rank-based prioritisation since the authors report greater gains in Breakout for the former.

Now it is possible to integrate Proportional Prioritised Experience Replay into the Double Deep Q-Learning framework with little alteration. Specifically, when we store a transition we also store a corresponding priority equal to $p_t = max_{i<t} p_i$. When sampling a mini-batch we sample according to the distribution $\mathcal{P}(j)$ and compute the corresponding importance-sampling weight

$$w_j = \frac{(N \cdot \mathcal{P}(j)^{-\beta}}{max_i w_i} \tag{6}$$

where N is the replay memory size. The importance-weights are necessary to correct the bias introduced by the new distribution over transitions. Next, we update the transition priorities according to the new TD error (which we compute anyway). Finally, we update the model weights according to:

$$\theta = \theta + \eta \sum_{j=1}^{k} w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1}) \tag{7}$$

4 Results

From Figure 1, we can see that the best training performance is given by Vanilla Deep Q-Learning and Double Deep Q-Learning since the average reward in training is consistently above that of the other algorithms past around 3,000 episodes. Double Deep Q-Learning and Double Deep Q-Learning with Prioritised Replay both take the longest before a significant increase in learning is observed. However, the rate of increase in score for Double Deep Q-Learning is higher than with prioritised replay. Vanilla Deep Q-Learning and Duelling Deep Q-Learning start learning quicker however Deep Q-Learning begins to perform better after around 2,000 episodes. The learning curve for Prioritised Double DQN is notably shifted towards the right. This is due to our implementation requiring that the replay buffer be pre-filled. However the agent was nonetheless trained for the same number of iterations, otherwise the comparison would not be fair and hence its graph is shifted.

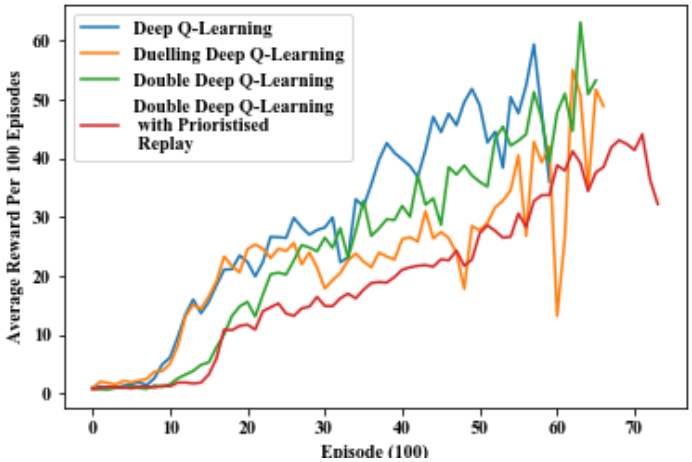

Figure 1: Moving Average Reward (100 episodes) per training epoch (100 episodes).

Vanilla Deep Q-Learning reached the highest score in training, followed by Double Deep Q-Learning with Prioritised Repay, then Duelling Deep Q-Learning and finally Double Deep Q-Learning. Despite having the lowest maximum value reached in training, Double Deep Q-Learning had the highest average score across 100 episodes. This was closely followed by Vanilla Deep Q-Learning and Duelling Deep Q-Learning. Double Deep Q-learning had a much lower average score, less than half the value of the other algorithms. These results can be seen in table 1.

Table 1: Average score is for 100 test episodes played by each agent with its learned policy using an ϵ-greedy policy ($\epsilon = 0.01$). Max training score is the highest score achieved by each agent during training.

Model	Max training score	Average score (100 episodes)
Vanilla Deep Q-Learning	**354**	76.66
Duelling Deep Q-Learning	306	**78.91**
Double Deep Q-Learning	279	72.93
Double Deep Q-Learning with Prioritised Replay	335	29.80

5 Discussion

Our experiments suggest that Duelling Deep Q-Learning proved to find the best solution out of the four deep reinforcement algorithms tried. Across the 100 test episodes, Duelling networks achieved an average episode score of *78.91*, Vanilla DQN followed closely with an average score of *76.66*, while Double Deep Q-Learning and Prioritised Double DQN scored *72.93* and *29.80* respectively.

From the results published by Hessel et al. (2018) [6] we would have expected that our algorithms would have scored, from best to worse, Double DQN, DQN, Prioritised Double DQN and Dueling. In our experiments (table 1), however, we found a different ordering. This result might be explained by the fact that we trained our agents for significantly fewer episodes than in [6], thus the ordering seen by Hessel et al. (2018) may only have materialised in later training iterations. Furthermore, Hessel et al. (2018) report the score for Duelling Double DQN while ours is for Duelling DQN, hence they are not directly comparable.

Prioritised Double DQN performed significantly worse during evaluation than the other three variants we tested (see table 1). The literature [6] has also found Prioritised Double DQN to score worse than both Vanilla DQN and Double DQN. Therefore, our result is consistent with those seen by others. We suspect that this might be due to something more fundamental. As such, we hypothesise that prioritised sampling in the case of Breakout may be detrimental to learning. In prioritised sampling, transitions with greater TD error are revisited and learnt from more often. In the context of a sparse reward (as is the case in Breakout) this more often than not translates to a large difference between the target and online network Q values. To this effect, during training, the online network quickly converges towards the target network. However, the target network is randomly initialised and thus the online network converges to a sub-optimal early on which the agent struggles to recover from for the remainder of the training. A similar conjecture is given by Zhong et al. (2017) [11].

Several difficulties were encountered while implementing the algorithms with respect to run time and memory usage. Precisely, in order to compete a reasonable number of training iterations (in a sensible amount of time) with a sufficiently large replay memory, certain optimisation were made. With regards to run time, we used the built in python operator *itemgetter* for collecting samples from the replay memory, this method was markedly faster than using list comprehensions according to our experiments. In addition, both the model and replay buffer were stored in GPU memory. To this effect, GPU optimisations in the *pytorch* library could exploited and there was no need to copy transitions between CPU and GPU memory during training, further reducing run time. However, as the system used for training had limited GPU memory, alterations to reduce the memory requirements of the replay buffer were needed. To achieve this, images (states) were cast to *uint8* type. When compared with *float*, *uint8* requires 4-fold less memory.

With more time, we would have liked to implement the remaining three extensions combined by Hessel et al. (2018) [6], Multi-step Learning, Distributional Q-Learning and Noisy Nets. After which, we would have liked to experiment with different combinations of the six extensions to DQN to find which performed best in the case of Breakout. It would have been interesting to see if we could replicate the excellent performance of Distributional Q-Learning seen in [2] or improve upon it by combining with another extension. Finally, we would have liked to run our algorithms for a greater number of iterations with larger replay buffers. For instance, Minh et al. (2015) used a replay buffer size of 1 million transitions and trained for 50 million frames [7], much more than what we were able to achieve on our hardware.

6 Conclusion

In this report we explored four deep reinforcement learning algorithms for learning control policy for the well know Atari game, Breakout. Specifically, we implemented Deep Q-Learning, Double Deep Q-Learning, Deep Q-Learning with Duelling Networks and Double Deep Q-Learning with Prioritised Replay. We trained each agent for 5 million frames with a replay memory size of 100 thousand transitions. Our experiments found that Duelling Networks achieved the greatest average score across the 100 episode test. While, the final average training score per 100 episodes saw Vanilla DQN and Double Deep Q-Learning perform joint best.

References

[1] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[2] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449–458. JMLR. org, 2017.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[4] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[6] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[10] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[11] Yangxin Zhong, Borui Wang, and Yuanfang Wang. Reward backpropagation prioritized experience replay. 2017.

A Vanilla Deep Q-Learning Neural Network Architecture

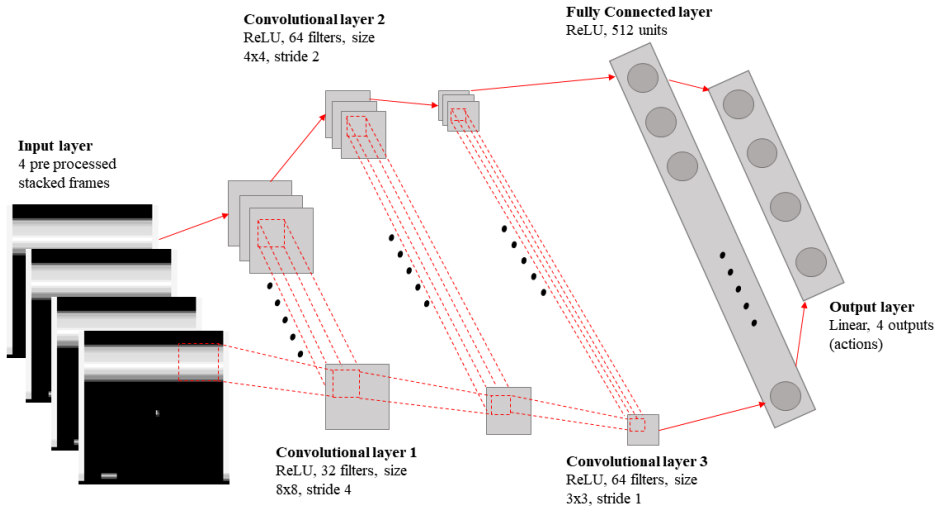

Figure 2: Network architecture used for Vanilla Deep Q-Learning

B Duelling Deep Q-Learning Neural Network Architecture

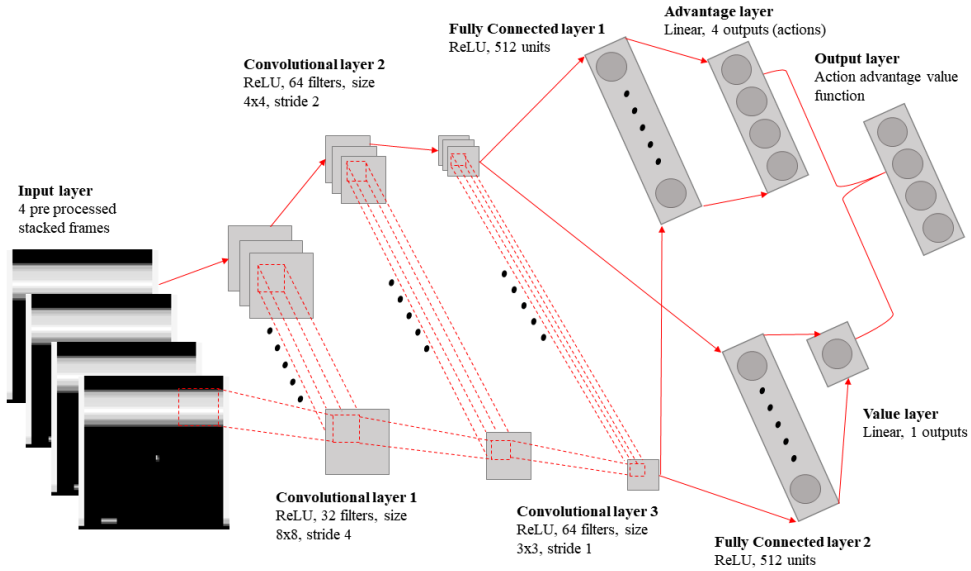

Figure 3: Network architecture used for Duelling Deep Q-Learning

C Hyperparameters Used for Training Each Agent

Table 2: Hyperparameters

Name	Description	value
Epsilon	Starts at 1.0, linearly decreased to 0.1 over the first 10% iterations, then linearly decreased from 0.1 to 0.01 over the remaining iterations	max: 1, min: 0.01
Iterations	Number of iterations the algorithm was run for	5,000,000
Minibatch size	Number of training cases to sample from the replay memory at each network update	32
Replay memory size	Number of transitions stored in the experience memory	100,000
Replay start size	Number of experiences before learning begins	5000
Discount factor	Discount factor used in q-value update	0.99
Learning rate	Learning rate used in network optimiser	0.00025
Max episode length	The maximum number of time steps in an episode	10,000
Target network update	The number of iterations before each update of the target network	10,000

D Pre-processing Algorithm

Algorithm 1 Input pre-processing

Initialise: Frame Queue $\mathcal{F}$ of size n, Frame skip m
Inputs: Selected action a
Ouputs: Current Frame Queue $\mathcal{F}$ (n x 80 x 80) of pre-processed frames

1. Apply selected action a to environment m times
2. Take the max pool of the most recent two frames
3. Convert max pool RGB frame to gray-scale
4. Down-sample frame by 50%
5. Crop frame to (80 x 80) and append to Frame Queue $\mathcal{F}$
6. Return current $\mathcal{F}$ (n x 80 x 80) of pre-processed frames