INM707 Deep Learning: Optimization Coursework

Miguel Bravo, Sahil Dhingra

This project report discusses the application of different reinforcement learning algorithms to two different tasks. Part 1 covers tabular Q-learning while Part 2 covers advanced Deep Reinforcement Learning algorithms – Deep Q-Networks (DQN) and its variants.

Part 1: Basic

1. Introduction: Domain and the Task

This part of the project uses a modified version of a simple Toy text environment from OpenAI Gym – Taxi-v3. The task is a navigation and planning task, which requires the agent – a taxi – to choose the most optimal route to pick-up and drop-off passengers from and to specified locations.

The original Gym environment consists of a 5x5 gridworld with four pick-up and drop-off points chosen at random and has a total of 500 states. This environment was significantly modified to reduce the state-space, while adding complexity to the navigation aspect of the agent's task. The gridworld was reduced in size to 5x4 cells, the pick-up point (P) was fixed to the top-left cell, the drop-off point (D) is assigned randomly, at the start of each episode, to one of the other 3 corners on the grid. The additional navigation complexity was introduced by adding 'pot-holes' (H) to one of four locations in row #2 in each episode, thus requiring the taxi agent to navigate around H. These changes reduced the number of total states to 480. The agent has 5 actions to choose from – 0: down, 1: up, 2: right, 3: left, 4: pick-up, 5: drop-off with different rewards associated with each action.

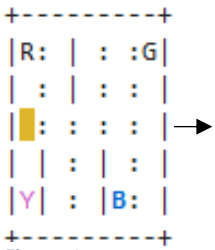


Figure 1: Original Taxi-v3

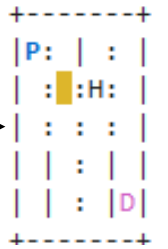


Figure 2: Custom Taxi env

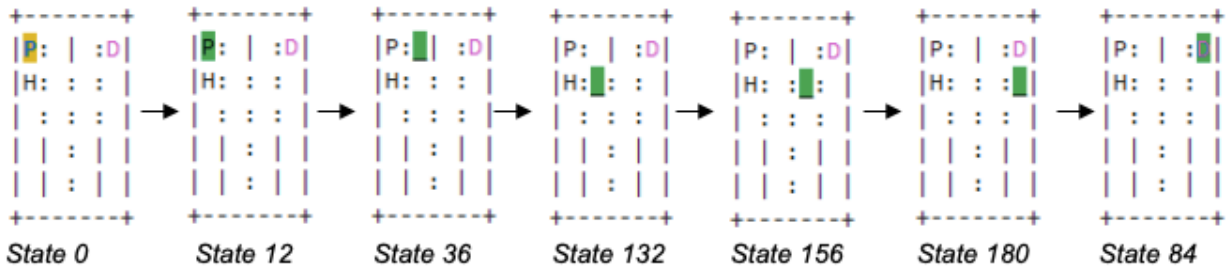
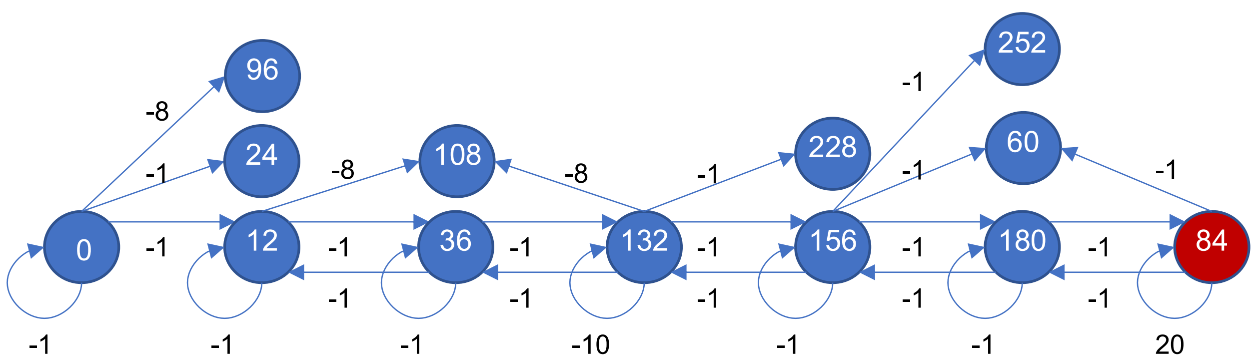


Figure 1: State transitions that maximise return when the episode starts in state 0. States are specific configurations of the grid, which are encoded as a unique number from 0 to 480.

2. State Transition function

The state transition function represents the different states the agent can move to, from the current state, based on the actions available to the agent. Since our environment has 480 states, we show only the state transition function with the agent starting at state 0 and ending in the absorbing state (i.e. state 84) of the current episode. Figure 1 above shows the state transitions using images from the environment while Figure 2 shows this graphically.



0 → {96, 0, 24, 0, 12, 0} | 12 → {108, 12, 36, 12, 12, 12} | 36 → {132, 36, 36, 12, 36, 36} | 132 → {228, 36, 156, 108, 132, 132} | 156 → {252, 60, 180, 132, 156, 156} | 180 → {276, 84, 180, 156, 180, 180} | 84 → {180, 84, 84, 60, 84, 84}

Figure 2: State transition function for one episode where the agent is begins in state 0, represented as a graph (above) and the state transitions associated with the 6 available actions in each state (below)

3. Reward function

State To: From:	0	12	24	36	60	84	96	108	132	156	180	228	252
0	-1	-1	-1				-8						
12	-1	-1		-1				-8					
24	-1		-1										
36		-1							-1				
60					-1	-1				-1			
84					-1	20					-1		
96	-1						-8						
108		-1						-8	-1				
132				-1				-8	-10	-1		-1	
156					-1				-1	-10	-1		-1
180						-1				-1	-1		
228									-1			-10	-1
252											-1	-1	-10

Table 1: Rewards matrix showing the highest rewards available in each state. Unless specified, each cell also has -10 reward associated with actions 4 and 5 (illegal drop-off and pick-up)

The agent receives different rewards based on the action performed. There is -1 reward for a move in any direction and/or for picking-up a passenger. There is a -10 reward for a wrong drop-off i.e. a location that is not D. These rewards were kept unchanged from the Gym taxi environment. A new reward of -8 was assigned to the agent driving over the

pothole to ensure the agent learns to avoid the pothole. Finally, a reward of +20 is given for a successful drop-off. Since the state-space is too large to present the entire R-matrix, an example of reward matrix based on an episode starting in state 0 and reaching terminal state in state 84 is shown in Table 1.

4. Policy

A policy is a mapping from states to actions. The action selection policy used during training is the ε-Greedy policy with decay. Under this policy, actions are chosen at random or from the Q-value table based on the value of epsilon. The higher the epsilon, the higher the probability of choosing a random action, allowing the agent to explore the state space. Using Q-values leads the agent to apply what it has learnt; i.e. exploit the learnt behaviour.

Epsilon was initially set to 1 to ensure the agent can sufficiently explore its environment and build the Q-value table. A decay rate of 0.9999 was used to decrease exploration at the end of each episode and slowly allow the agent to exploit the values learnt. A floor of 0.005 was set for the minimum value of epsilon, so that its value does not drop to zero.

5. Q-Learning Algorithm

The following parameters were selected for training the agent:

Parameter	Value	Description
Epsilon	**1**	Value to determine epsilon-greedy behaviour policy during training to ensure agent adequate exploration of the state space [1]
Epsilon decay	**0.999**	Used to decrease exploration of state space, so the agent can exploit learned behaviour towards the end of training.
Learning rate (alpha)	**0.1**	Value between 0-1. The higher, the more weighting is given to new Q-value updates. A small learning rate was selected for this problem as we are training the agent for 50K episodes, therefore allowing sufficient time for the Q-values to be updated.
Discount factor (gamma)	**0.6**	Reflects the trade-off between importance assigned to future rewards versus current rewards. In our problem, the agent received a positive reward at the end of the episode once a successful drop-off has been completed. However, it is also important to avoid potholes. Therefore, a discount factor skewed slightly in favour of long-term reward was used.

Table 2: Parameters selected for the initial training of the Q-learning agent

6. Q-Matrix

In the first step, a Q-table with zero values is created. The dimensions of the Q-table match the number of states (rows) and the number of actions (columns), so that a Q-value can be assigned to each state-action pair Q(s, a). The agent initially explores the state-space since the starting value of epsilon is 1. This allows the Q-table to slowly start being populated as the agent explores the states, receives a reward, and updates the Q-table. Below are the main steps of how the Q-table is updated.

Step 1: Agent performs an action according to ε-Greedy policy
Step 2: Receive reward and move to next state
Step 3: Calculate new Q-value using the Q-learning algorithm and update the Q-table

$$Q_{\text{new}}(s,a) = (1-\alpha)\,.\,Q(s,a) + \alpha.\,(r + \gamma\,.\,\max Q(s',a'))$$

Equation 1: Q-table update rule. New action value Q(s,a) is a mixture of the existing value and the max. value at the next state

The above steps are repeated until either the Q-table stops updating (i.e. reaching convergence) or a certain number of training episodes have been completed. In our project, we used the latter as the stopping criteria. At the end of the training, we have a Q-table with values that our agent can use with certainty to plan and navigate the environment without errors.

To present the Q-matrix in a format similar to the rewards matrix, the Q-table was adapted to show how the agent would make decisions based on the final Q-table when starting in state 0 – as shown in Table 3.

Q- Matrix	0	12	24	36	60	84	96	108	132	156	180	228	252
0	-1.030	-0.051	-1.618				-8.618						
12		-0.051		1.582				-5.418					
24	-0.051		1.582										
36		-0.051		1.582					4.304				
60					16.400	29.000				8.840			
84					16.400	50.000					16.400		
96	-1.030						-8.621	-5.419					
108		-0.052							4.304				
132				1.582				-5.418	-4.696	8.840		1.582	
156					16.400				4.304	-0.160	16.400		4.304
180						29.000				8.839	16.347		
228									4.304			-7.428	4.286
252										8.840		1.503	-4.736

Table 3: Final Q-Matrix for the episode starting in state 0

7. Results and Analysis

Performance during training:

The agent's performance over the training episodes is shown above. We measure the performance on four metrics – **1.** the average reward achieved per episode, which should be between 6 – 14 depending on where the agent spawns, **2.** The number of steps taken to complete the episode, which should be between 7 – 14, **3.** The number of penalties received for incorrect pick and drop-off, which should be zero and **4.** The number of penalties received for driving over the pothole, which should also be zero.

The charts in Figure 3 show that the agent achieves ideal performance at approximately 30k episodes with average rewards increasing to positive from large negative rewards initially. The passenger penalties take longer to converge at around 40k episodes while the pothole penalties reduce to zero close to 50k episodes.

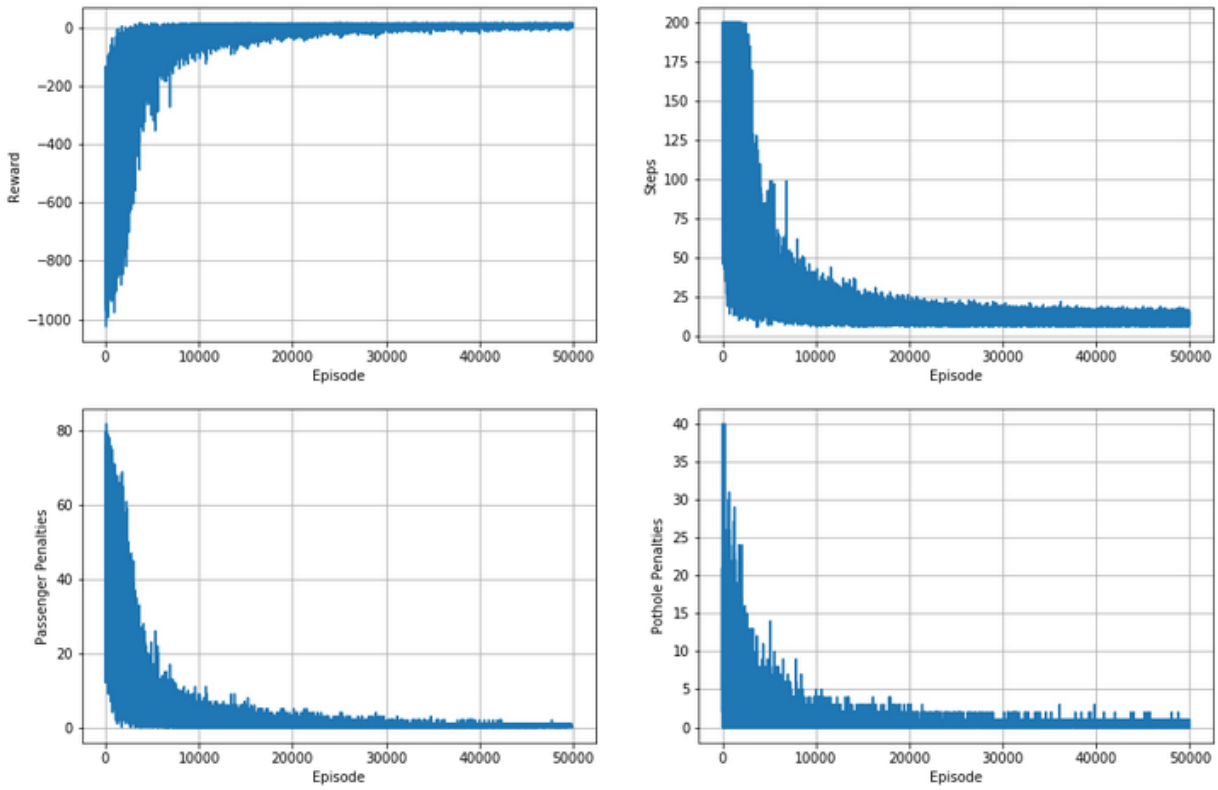


Figure 3: Agent's performance improvement over the training episodes. ***Top left*** *– the total reward per episode.* ***Top right*** *– total steps taken by the agent to complete an episode.* ***Bottom left*** *– total penalties associated with incorrect pick-up and drop-off.* ***Bottom right*** *– total penalties associated with driving over the pothole.*

Comparison of performance on different parameter values/policies:

The agent was trained with a range of parameter selections of alpha and gamma, and starting values for epsilon – to see the effect on the agent's behaviour. Selections for alpha ranged over [0.1, 0.5, 0.7], and gamma ranged over [0.1, 0.6, 0.9]. Experiments with epsilon saw it ranging over [1, 0.8, 0.6, 0.4, 0.2, 0.1], with alpha and gamma fixed at 0.1. Training was conducted over 50k episodes for each run. The results of these experiments are shown in the figure below.

Discussion of results:

From the hyperparameter testing, it is apparent the agent converges quickly to optimal behaviour regardless of choice of alpha or gamma, as can be seen in Figure 4. The only possible exception is [alpha=0.7, gamma=0.6] where the agent displays slightly smoother results than the other experiments over the first 10k episodes, and marginally superior behaviour approximately between episodes 2k – 4k.

What does appear to have a greater impact on convergence is the starting value of epsilon; that is, the agent's epsilon-greedy behaviour policy during training. As the starting value decreases – that is, as the agent favours exploitation more – convergence occurs more quickly. This suggests the environment holds no real surprises for the agent. As the agent is allowed to rely more on what it has learned, the quicker it reaches optimal behaviour, without falling into dead ends.

What is particularly noteworthy is the remarkable speed at which the agent learns to avoid the penalty from illegal passenger pickups and drop-offs, as it favours exploitation more. This seems to be the main driver behind the quicker convergence towards maximal reward with decreasing epsilon.

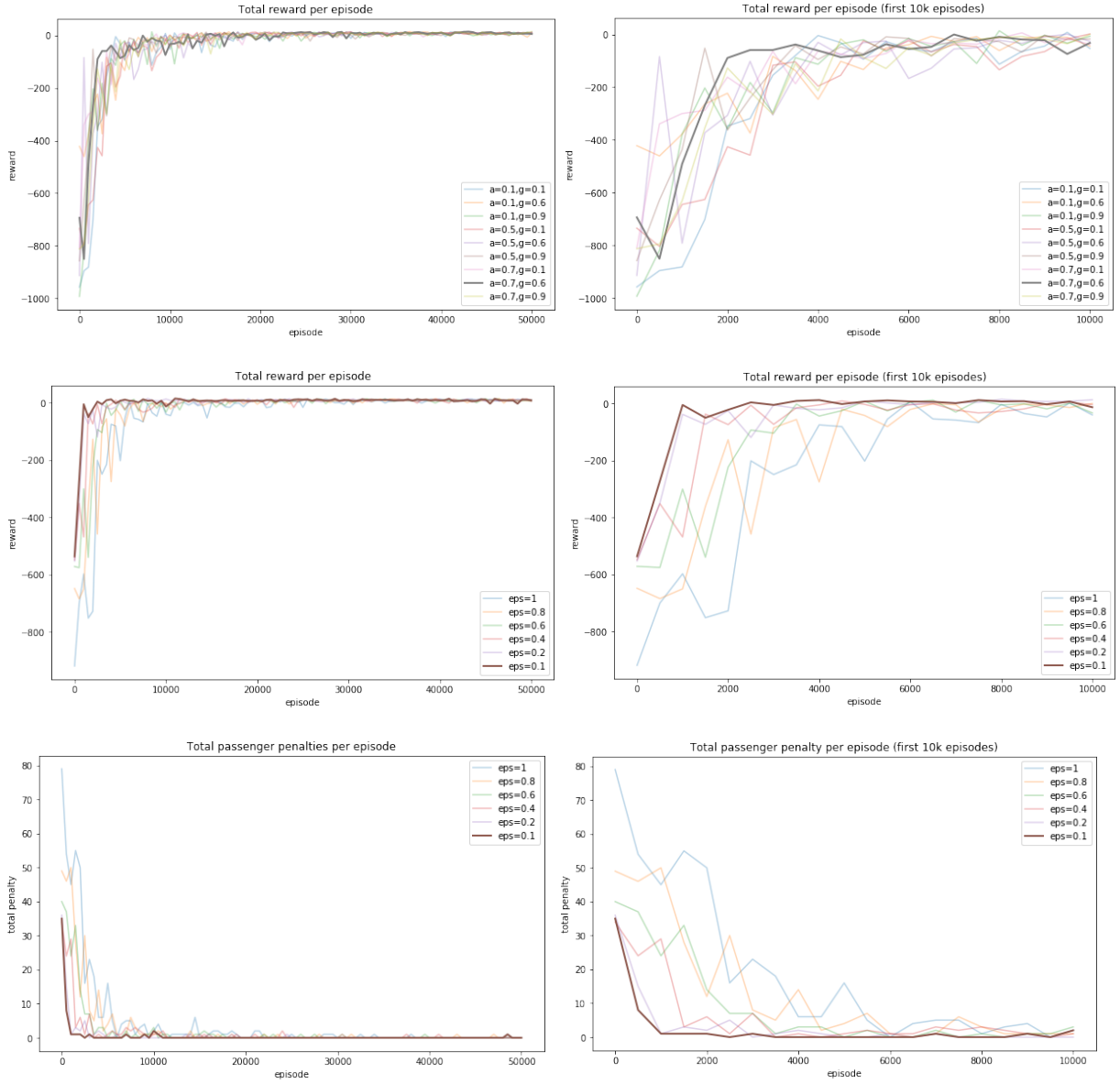


Figure 4: Example results from parameter experimentation. ***Top row*** *– total reward per episode for different alpha and gamma selections.* ***Middle row*** *– total reward per episode for different epsilon starting values.* ***Bottom row*** *– total passenger penalty per episode for different epsilon starting values.*

Part 2 – Advanced

For the advanced part of this coursework, we apply the Deep Q-Network (DQN) agent to the Atari game, Space Invaders. We also experiment with improvements to DQN using Double DQN (DDQN), Prioritised Replay, and Dueling architecture.

To perform this task, we use the Space Invaders-v0 environment from OpenAI Gym. This environment requires the agent to learn a successful policy for Space Invaders directly from the game's raw video display and associated rewards (i.e. game score). Our choice to use DQN is motivated by previous RL research which has shown it achieves super-human performance on Atari games [2]. In general, DQNs are well suited for tasks involving large state spaces and a small discrete action space – which is the case in Space Invaders. As per the literature we expect this agent to learn a good policy for this task and outperform a baseline agent performing a random action at each time-step.

1. Architecture

DQN learns using a deep neural network to approximate the Q-function which maps state-action pairs to their return. The resulting policy is then derived by always taking the action with maximal return, as estimated by the network. To learn from the game's visual display we used a Convolutional Neural Network, with 3 convolutional layers, a fully connected layer and an output layer as shown in Figure 5. This architecture design is inspired from the original paper on DQN by Mnih et al [1]. The number of network outputs were reduced from 6 to 5 as we reduced the number of actions available to the agent (see Table 4).

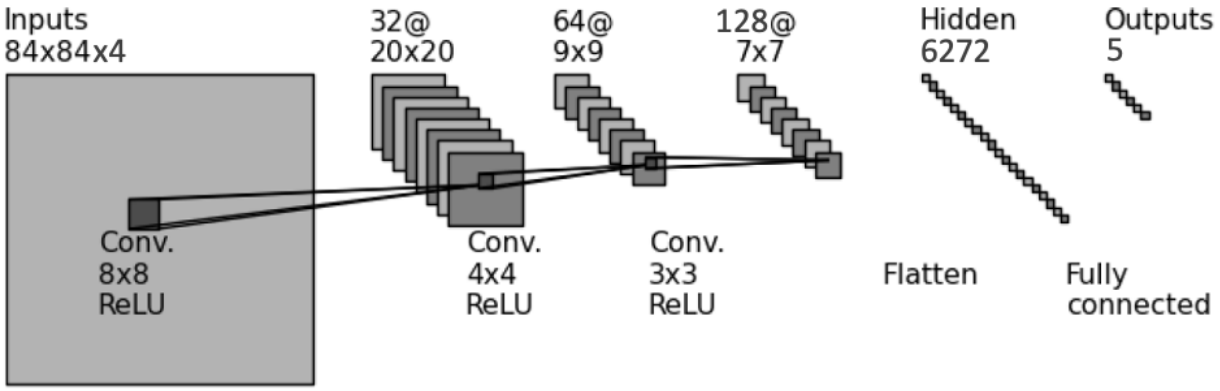


Figure 5: CNN architecture used. The model has 3 convolutional layers, and 2 fully connected layers. It takes 4 84x84 stacked frames as input, and outputs a state, action value for each of the 5 available actions

2. DQN – Approach and Hyperparameters

Pre-processing: The Atari environment produces frames of dimension 210x160 pixels in RGB format. To reduce the dimensionality, the frames were pre-processed. The frames were cropped to capture the playing area only, reducing the size to 185x95 pixels and then converted to grayscale and resized to 84x84 pixels. The original frame and the pre-processed frame are shown in Figure 6.

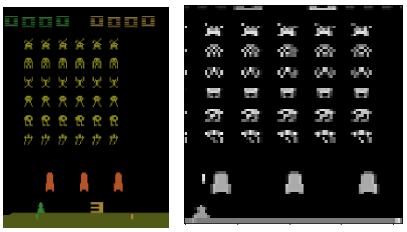


*Figure 6: **Left** - original frame. **Right** – pre-processed frame*

Frame Stacking and Skipping: To provide the agent with a sense of motion, multiple frames are stacked together and fed as input to the CNN. Frames are stacked in groups of four with frame-skipping to ensure they are 3 frames apart from each other. This is to provide sufficient variability between frames to detect motion.

Figure 7: Stack of four frames, skipped by 3 frames

We initially produced our own custom implementation of this, but ultimately adopted an existing implementation from OpenAI Stable Baselines for greater efficiency.

Experience Replay: Our DQN model uses a memory buffer to enable the agent to learn. The memory holds the agent's experiences (state, action, reward, next state) from previous time-steps. At each step, a random sample is drawn from the memory and used to perform a mini-batch update of the network's weights to reduce the temporal difference error, optimising the model's Q-value predictions and hence its policy.

Hyperparameter selection: the following parameter selections were used for training the final DQN and its variants, after experimenting with a number of choices. We opted for selections that adequately traded off stability of training with computational cost.

Parameter	Value	Description
Min epsilon	**0.01**	Determines the final exploration rate of actions during training, according to the epsilon-greedy policy.
Initial epsilon	**1**	Determines the initial exploration rate of actions.
Epsilon decay	**50%**	Rate at which exploration decreases during training. Exploration decayed to 50% by mid-training; to allow for adequate exploitation.
Episodes	**5K**	Number of episodes the DQN agent (and its variants) were trained for. It was not increased higher due to computational limitations.
Initial memory size	**10K**	Minimum number of experiences loaded into replay memory before training can begin.
Max memory size	**200K**	Maximum capacity of the replay memory before replacing older experiences with newer ones.
Batch size	**128**	Size of random sample drawn from the replay memory to perform mini-batch update of the network weights.
Target network update frequency	**200**	Number of steps before agent's target network is updated. By keeping target net weights frozen between updates, this improves performance
Learning rate	**0.0003**	Rate at which online network's weights are updated in backpropagation. We opted for a low rate to make updates more stable.
Gamma	**0.99**	We use a high gamma value to make our agent prioritise later rewards.
Agent actions	**5**	Number of actions available to the agent. 'Do nothing' was removed since this empirically gave better performance than the full action space

Table 4: Hyperparameters selected for training the DQN agent

3. Results

The training results for the DQN model are shown in Figure 8. We use two metrics to evaluate the agent's performance – episode scores and average Q-values. The scores per episode on the left, while noisy, show a slight upward trend up to episode 4000. The average Q-values, on the right, show a steady increase during training suggesting improvement in the agent's policy. However, the scores start to decrease after episode 4000 when epsilon is low and never attain human performance levels, which suggests the model needs improvement. There could be several factors causing sub-par scores – one is the buffer size (200k), which starts getting replaced around 2.5k episodes leading the agent to forget previously learnt state, action combinations. Another factor could be that the agent likely requires significantly more training with more intensive exploration before it can start to exploit a successful policy. Indeed, the literature indicates convergence on this task is achieved after hundreds of thousands of episodes [1] – which is beyond our computational resources for this project.

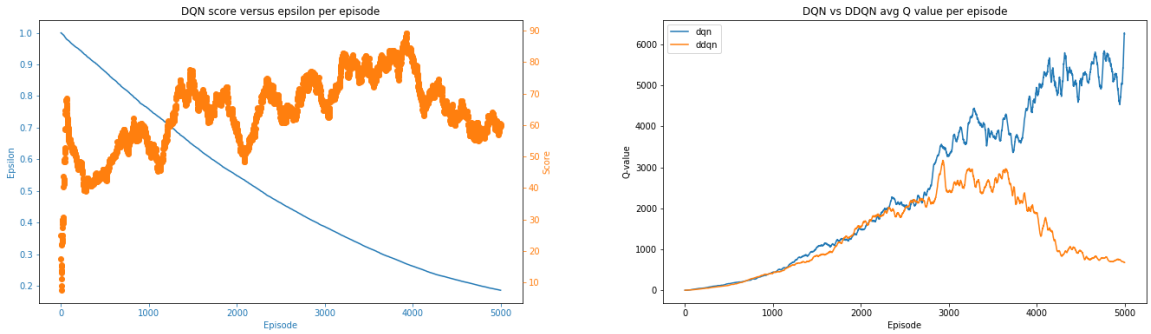


Figure 8: Training statistics for the DQN and DDQN agent. ***Left –*** *Total reward (running average) achieved by the agent in each episode, overlaid with epsilon exploration rate.* ***Right –*** *Average Q-value predicted by DQN and DDQN agent at the end of each episode.*

Double DQN (DDQN): DQN has been shown to overestimate Q-values which can be detrimental to performance. The rightmost plot in Figure 8 shows this is true of our agent, with predicted Q-values significantly higher than reported figures in the literature [1]. DDQN overcomes the issues of overestimating Q-values by decoupling action selection from evaluation when calculating the target the agent learns against. The online network selects the action, while the target network estimates its value. We applied DDQN and it resulted in a significant decrease in Q-value estimates as shown in Figure 8. This also translated into slightly more stable and upward-trending scores during training – though DDQN test performance remained in line with DQN as shown in Figure 9. However, the Q-values are still significantly higher than normal.

Further improvements: To boost the agent's performance, we implemented two additional improvements; Prioritised experience replay and Dueling architecture. Hessel et al. showed these and other additions – known as rainbow DQN – result in significantly improved performance [3]. Prioritised replay involves sampling experiences from the memory with probability proportional to their last encountered temporal difference error during training – allowing the agent to prioritise learning from unexpected experiences [4]. Dueling architecture consists of two separate streams of computation to estimate the

value and action advantage functions – improving learning without any underlying changes to the DQN algorithm [5]. Applying these techniques resulted in training two more agents: Double-DQN with Prioritised Replay (PDDQN), and Dueling Double-DQN with Prioritised Replay (DPDDQN).

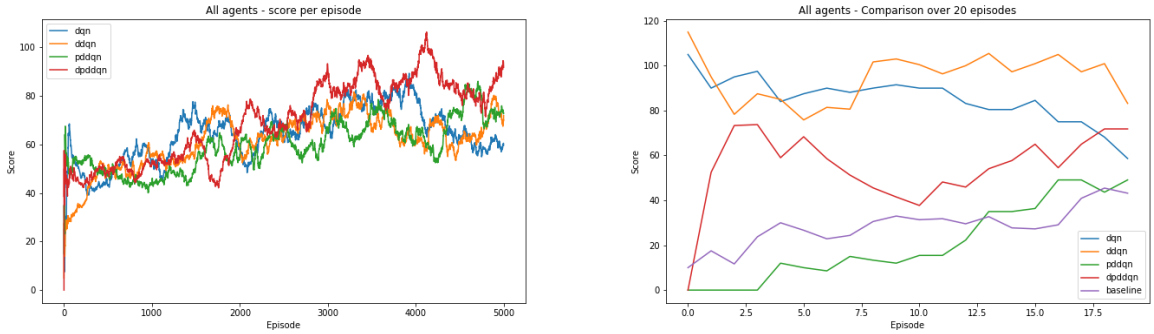


*Figure 9: **Left** - Training score (running average) comparison across all agents. **Right** – Test score (running average) comparison over 20 episodes for all agents, including 'random action' baseline agent.*

Figure 9 shows, on the left, the performance of the four models during training. DPDDQN showed marginal performance improvement over the other models, while PDDQN reached the same scores as DDQN. Therefore, we did not observe the results in [3], possibly due to fewer computational resources.

We tested the agents' performance using the final policy models on 20 episodes. The results on the right in Figure 9 show that the DDQN and DQN models achieved the best scores. More importantly, all models (barring PDDQN) outperformed the 'random action' baseline agent with DDQN and DQN performing the best. Table 5 below shows the average scores of the models compared to the baseline agent. The tests demonstrate that our agent learnt to play Space Invaders better than a random agent but did not reach human level performance demonstrated in previous work.

	Baseline	DQN	DDQN	PDDQN	DPDDQN
Best 20-episode performance (avg. +/- std)	**38 +/- 33**	**73 +/- 41**	**92 +/- 51**	**33 +/- 46**	**60 +/- 60**

Table 5: Test score comparison versus baseline. Trained agents' average total reward (and standard deviation) over 20 episodes, using a fully greedy policy (i.e. no random actions) – compared against baseline agent taking only random actions

Based on the results observed in this project, there are several areas where we could make improvements to our agents. First, training the agents for at least one order of magnitude longer could achieve significantly better policies. Another technique would be to optimise the epsilon decay to better balance exploitation vs exploration. Finally, increasing the maximum memory size to at least +1M frames and the initial memory size to 80K in line with the approach taken by Hessel et al [3]. However, this is contingent on having the necessary compute resource available to deal with the increased computational burden created by these changes.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," arXiv:1312.5602, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou and H. King, "Human-level control through deep reinforcement learning," *Nature,* no. 518, pp. 529-533, 2015.

[3] H. v. Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[4] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[5] Z. Wang, T. Schaul, M. Hessel, H. v. Hasselt, M. Lanctot and N. d. Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *Thirty-third International Conference on Machine Learning*, 2016.

[6] T. Schaul, J. Quan, I. Antonoglou and D. Silver, "Prioritized Experience Replay," in *Fourth International Conference on Learning Representations*, San Juan, 2016.

Individual Contribution

This project was a highly collaborative effort between my coursework partner, Sahil Dhingra, and I. We both worked closely together on all aspects of the project in a highly fluid manner, engaging in collaborative practices such as pair programming and continually reviewing each other's inputs to the report. This makes it difficult to separate out our respective contributions, and it was clear to me our overall individual efforts on the project were fair and commensurate. That said, my main contributions are listed below:

- Implementation of the custom taxi environment in the basic section
- Results write-up for the basic section of the report
- Implementation of DQN variants (i.e. Double DQN, Prioritised Replay, Dueling) in the advanced section
- Approach and results write-up for the advanced section of the report

Appendix

Implementation

As part of our coursework submission, we have also separately included the code used to implement our work for both the basic and advanced sections of the project.

Advanced Section – Full Results

Graphical outputs showing our full set of training results for the different DQN agents implemented in the advanced section.