Basic

1. Defining an environment and problem to be solved

The environment is defined as a two-dimensional 5x5 grid, where an agent looks to navigate around the environment and collect the resource prior to reaching the destination or goal state. The problem is laid out is as a Markov Decision Process (MDP) with a finite set of states, actions, and a reward and state transition function which will be further discussed in the subsequent sections. Figure 1 provides an illustration to the environment and problem statement. Here, the agent, resource and goal state are illustrated in Figure 1 as the person, gold bars and house respectively.

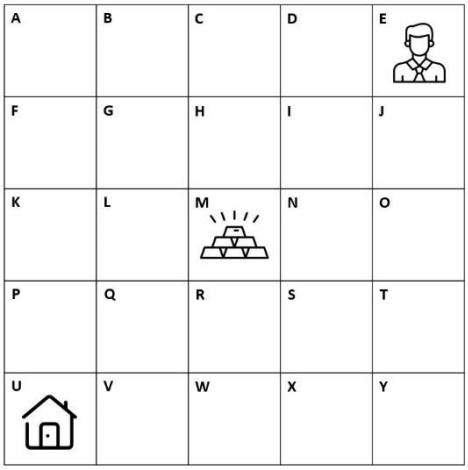

Figure 1: 5x5 Grid Environment

The objective of our agent is to maximise its return (i.e. accumulated reward) by navigating through the environment from its starting position (cell 'E') to collect the resource (cell 'M') and transport it to the goal state (cell 'U') in the least number of moves or timesteps.

The optimal policy will be one which locates and collects the high reward (i.e. resource), minimises negative rewards (i.e. avoid wandering aimlessly or taking unnecessary steps) and arrives at the goal state in the least number of moves or timesteps.

2. Defining a state transition and reward function

2.1. State transition function

Given that the environment is a two-dimensional 5x5 grid, a state is represented by a single cell in the grid and the available actions are the cells that the agent is able to navigate to from its current state (i.e. current position). For the purposes of a fairer comparison on the impact of changes in the hyperparameters and reducing the computational intensity, the states of the agent, resource and goal state are fixed (i.e. their positions are fixed and will not be randomly initialised with every episode).

The state transition function is given by: $s_{t+1} = \delta(s_t , a_t)$. That is that the next state at timestep t+1, s_{t+1}, which the agent transitions to is a function of its current state, s_t, and action (from its current state) at timestep t, a_t. The state space is given by each cell in our 5x5 grid (thus 25 states altogether) with an action space, A, consisting of all possible actions from each state. Our agent is able to move in four directions (right, left, up and down) where permitted. For example, if our agent is along the borders of the grid, it would not be able to move out of the grid. Cell 'U' represents the goal state where an episode of training is terminated once or if our agent reaches it. Table 1 maps an example of the state transitions from an arbitrary state, cell 'P'.

Current State (s_t)	Available Actions (a_t)	Next State (s_{t+1})
P	Up	K
	Right	Q
	Down	U

Table 1: State Transitions from P

2.2. Reward function

The reward an agent gets from moving to a new state by taking an action from the list of available actions and current state is given by: $r_{t+1} = r(s_t , a_t)$. Namely, the reward at the next timestep t+1, r_{t+1}, is a function of its current state, s_t, and choice of action at timestep t, a_t.

Specific to this study, a reward of 300 is allocated to the resource. To avoid a situation where the resource becomes an absorbing state (i.e. agent continuously moves back-and-forth between the resource and an adjacent state), the resource is set up such that it can only be collected once. Once collected, the reward is adjusted to a negative reward of -5. What's more, the reward of the goal state is initialised with a negative reward of -5 and adjusted to a positive reward of 200 after the resource is collected. This is to avoid the agent thinking that there are two optimisation tasks – one to collect the resource and another to arrive at the goal state. Additionally, to penalise the agent from making unnecessary moves and incentivise an optimal policy with the least number of moves or timesteps in achieving its goal, transitions between all other states incur a negative reward of -5.

Table 2 provides an overview of the reward matrix. An element $r_{i,j}$ in the reward matrix represents the reward the agent incurs from moving from state i to state j, from timestep t to t+1. Actions that the agent is unable to take from its current state are marked as empty cells in the reward matrix.

$s_t \backslash s_{t+1}$	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
A		-5				-5																			
B	-5		-5				-5																		
C		-5		-5				-5																	
D			-5		-5				-5																
E				-5						-5															
F	-5						-5				-5														
G		-5				-5		-5				-5													
H			-5				-5		-5				300 (-5 after collection)												
I				-5				-5		-5				-5											
J					-5				-5						-5										
K						-5						-5				-5									
L							-5				-5		300 (-5 after collection)				-5								
M								-5				-5		-5				-5							
N									-5				300 (-5 after collection)		-5				-5						
O										-5				-5						-5					
P											-5						-5				-5 (200 after collection)				
Q												-5				-5		-5				-5			
R													300 (-5 after collection)				-5		-5				-5		
S														-5				-5		-5				-5	
T															-5				-5						-5
U																-5						-5			
V																	-5				-5 (200 after collection)		-5		
W																		-5				-5		-5	
X																			-5				-5		-5
Y																				-5				-5	

Table 2: Reward Matrix

3. Setting up Q-learning parameters and policy

Q-learning depends on a set of key parameters, namely: learning rate (α), discount factor (γ), behaviour policy, target policy, number of episodes and timesteps.

α represents the learning rate, where $\alpha \in [0,1]$. It dictates how fast learning takes place or equivalently, how much new information should be weighed over old information. Taking both extremes, an α of 0 means that Q-values are never updated and nothing is learned while an α of 1 means that the agent learns quickly but not necessarily accurately.

γ represents the discount factor, where $\gamma \in [0,1]$. It determines the importance of future rewards and taking an extreme with γ of 0, the agent only considers immediate rewards and has no consideration of future rewards. Conversely, a γ of 1 means that the agent is far-sighted and has strong considerations for future rewards as it weighs all rewards equally (regardless of timestep).

A behaviour policy governs the action selection of our agent. Employing an ε-greedy policy requires an ε to be defined, which is the probability of an action to be chosen at random for exploration, versus a greedy action (i.e. action with maximum Q-value) to be taken for exploitation with probability (1- ε). ε of 0 means that actions are always chosen greedily while an ε of 1 means that actions are always chosen randomly. Additionally, to account for exploration being more favourable early on (as Q-values are mostly zeroes) versus exploitation later on, ε-decay can be adopted. With ε-decay, rates of decay given by scalars between 0 and 1 are multiplied onto ε at every timestep depending on whether the agent explored or exploited in the previous timestep. Other behaviour policies include the Boltzmann policy, which follows a similar argument with its parameter, T.

The number of episodes and timesteps will determine whether our agent is able to arrive at the goal state and if the Q-values are able to converge. With very few episodes and timesteps, the agent will not have sufficient time to explore and exploit the environment. On the other hand, with sufficient episodes and timesteps, the agent is able to explore and exploit the environment sufficiently, resulting in Q-values ultimately converging.

Table 3 below provides a summary to the Q-learning parameters and policy used.

Parameter		**Value**
α		1.0
γ		0.8
Behaviour Policy		ε-greedy
ε-Decay	ε	0.9
	Rate of decay if previously explored	0.999
	Rate of decay if previously exploited	0.9999
Target Policy		Greedy
Number of Episodes		1000
Number of Timesteps		500

Table 3: Q-Learning Parameters and Policy

By adopting a greedy target policy, the Q-learning update rule simplifies to:

$$Q^{new}(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[(r_{t+1} + \gamma max_a Q(s_{t+1},a)) - Q(s_t,a_t)]$$

4. Running the Q-learning algorithm and representing its performance

To navigate through the environment and find an optimal policy, Q-learning is employed to maximise the expected return over all states. It is a "model-free" approach which uses its latest experience to update its policy, and its values are stored in a Q-matrix.

The Q-matrix is initialised as a zero matrix but can be initialised arbitrarily instead. With each episode, the agent is initialised at a starting state (i.e. cell "E" for this study). Within each episode and with each timestep, the agent chooses an action, a_t, from its current state, s_t, via its behaviour policy (i.e. ε-greedy). From there, the agent observes the reward, r_{t+1}, subsequent states, s_{t+1}, and updates the Q-values as follows: $Q^{new}(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[(r_{t+1} + \gamma max_a Q(s_{t+1},a)) - Q(s_t,a_t)]$. The current state, s_t, is updated to the next state, s_{t+1}, and is repeated for each episode until either the agent arrives at the goal state or runs out of timesteps.

Below provides an example of a single learning episode.

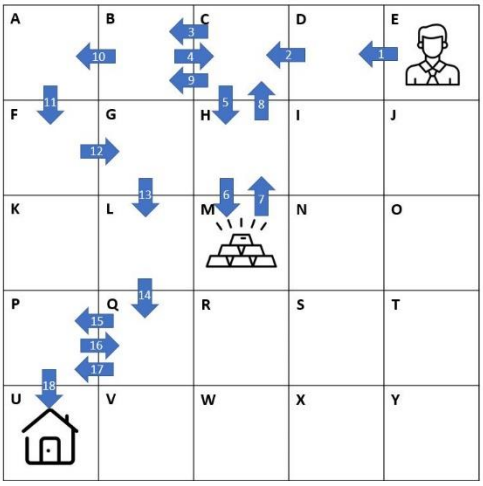

Figure 2: State Transitions for a Single Episode

Timestep	Behaviour	Epsilon	$Q(s_t, a_t)$	Q-value Update
t_1	Explore	0.9000	Q(E, D)	Q(E, D) + α*(r(E, D) + γ*max(Q(D, C), Q(D, I), Q(D, E)) - Q(E, D))
t_2	Explore	0.8991	Q(D, C)	Q(D, C) + α*(r(D, C) + γ*max(Q(C, B), Q(C, D), Q(C, H)) - Q(D, C))
t_3	Explore	0.8982	Q(C, B)	Q(C, B) + α*(r(C, B) + γ*max(Q(B, A), Q(B, C), Q(B, G)) - Q(C, B))
t_4	Explore	0.8973	Q(B, C)	Q(B, C) + α*(r(B, C) + γ*max(Q(C, B), Q(C, D), Q(C, H)) - Q(B, C))
t_5	Explore	0.8964	Q(C, H)	Q(C, H) + α*(r(C, H) + γ*max(Q(H, C), Q(H, G), Q(H, I), Q(H, M)) - Q(C, H))
t_6	Explore	0.8955	Q(H, M)	Q(H, M) + α*(r(H, M) + γ*max(Q(M, H), Q(M, L), Q(M, N), Q(M, R)) - Q(H, M))
t_7	Explore	0.8946	Q(M, H)	Q(M, H) + α*(r(M, H) + γ*max(Q(H, C), Q(H, G), Q(H, I), Q(H, M)) - Q(M, H))
t_8	Explore	0.8937	Q(H, C)	Q(H, C) + α*(r(H, C) + γ*max(Q(C, B), Q(C, D), Q(C, H)) - Q(H, C))
t_9	Explore	0.8928	Q(C, B)	Q(C, B) + α*(r(C, B) + γ*max(Q(B, A), Q(B, C), Q(B, G)) - Q(C, B))
t_{10}	Explore	0.8919	Q(B, A)	Q(B, A) + α*(r(B, A) + γ*max(Q(A, B), Q(A, F)) - Q(B, A))
t_{11}	Explore	0.8910	Q(A, F)	Q(A, F) + α*(r(A, F) + γ*max(Q(F, A), Q(F, G), Q(F, K)) - Q(A, F))
t_{12}	Explore	0.8901	Q(F, G)	Q(F, G) + α*(r(F, G) + γ*max(Q(G, B), Q(G, F), Q(G, H), Q(G, L)) - Q(F, G))
t_{13}	Exploit	0.8893	Q(G, L)	Q(G, L) + α*(r(G, L) + γ*max(Q(L, G), Q(L, K), Q(L, M), Q(L, Q)) - Q(G, L))
t_{14}	Exploit	0.8892	Q(L, Q)	Q(L, Q) + α*(r(L, Q) + γ*max(Q(Q, L), Q(Q, P), Q(Q, R), Q(Q, V)) - Q(L, Q))
t_{15}	Exploit	0.8891	Q(Q, P)	Q(Q, P) + α*(r(Q, P) + γ*max(Q(P, K), Q(P, Q), Q(P, U)) - Q(Q, P))
t_{16}	Explore	0.8890	Q(P, Q)	Q(P, Q) + α*(r(P, Q) + γ*max(Q(Q, L), Q(Q, P), Q(Q, R), Q(Q, V)) - Q(P, Q))
t_{17}	Explore	0.8881	Q(Q, P)	Q(Q, P) + α*(r(Q, P) + γ*max(Q(P, K), Q(P, Q), Q(P, U)) - Q(Q, P))
t_{18}	Explore	0.8872	Q(P, U)	Q(P, U) + α*(r(P, U) + γ*max(Q(U, P), Q(U, V)) - Q(P, U))

Table 4: Q-value Updates and Epsilon Decay from a Single Episode

With a single training episode, the Q-learning parameters and Q-matrix are initialised and the agent navigates through the environment. ε is set to a high value to favour exploration over exploitation early on, as the Q-values are still mostly zeros, and decays with every timestep. Given this, the agent is likely to take unnecessary steps and arrive at the goal state by chance. In this example, the agent moves back-and-forth between 2 sets of states (C and B, and Q and P) in timesteps 3 and 4, and 16 and 17. Additionally, in timestep 18, the agent arrives at the goal state "U" through exploration, and the learning episode terminates, but could have equally as likely moved to states "K" or "Q" instead. However, with more learning episodes, Q-values are more representative and exploitation is favoured (since ε decays). With both in tandem, we can expect convergence of an optimal policy and the agent maximising its accumulated reward through: collecting the resource, arriving at the goal state, and doing it in the least number of moves or timesteps.

After training for 1000 episodes, below provides plots of the average cumulative reward accumulated and the average number of timesteps required from start to goal state over episodes.

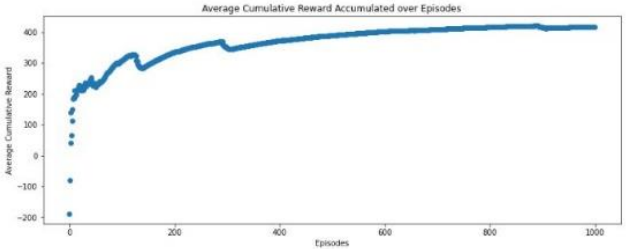

Figure 3: Average Cumulative Reward

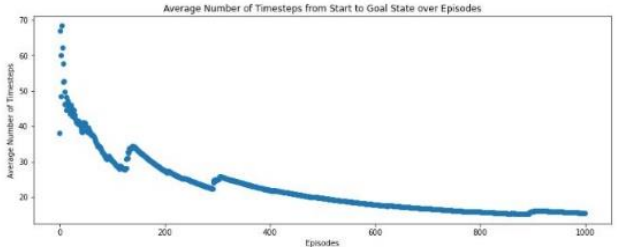

Figure 4: Average Number of Timesteps

Indeed, the flattening of both curves over the episodes suggest that, generally, with each episode, the agent learns more about the environment and eventually its accumulated reward and number of timesteps it needs to arrive at the goal state reaches a maximum and minimum respectively, through an optimal policy. Furthermore, the high rate of exploration and pre-convergence of the Q-values are shown on both plots by the relatively spurious points from the 1st to around the 170th training episode.

5. Repeating the experiment with different parameter values, and policies

5.1. Varying parameters (α and γ) and policies (rate of ε -decay)

The experiment was repeated but with varying learning rates, discount factors and rate of ε -decay.

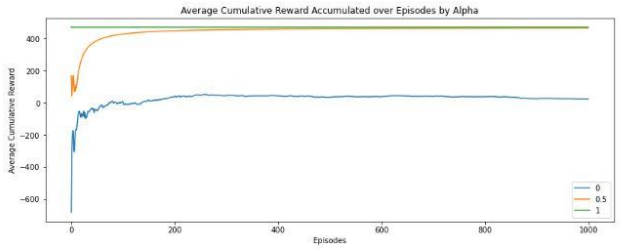

Figure 5: Average Cumulative Reward by Alpha

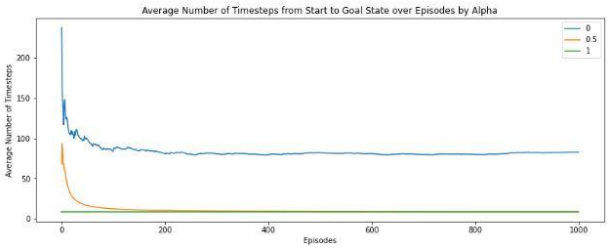

Figure 4: Average Number of Timesteps by Alpha

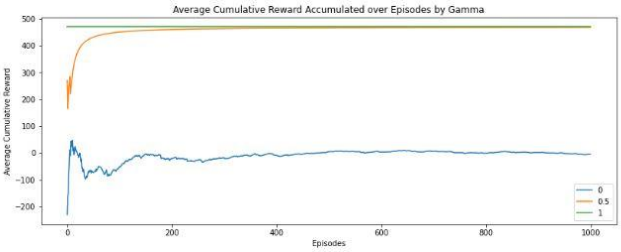

Figure 7: Average Cumulative Reward by Gamma

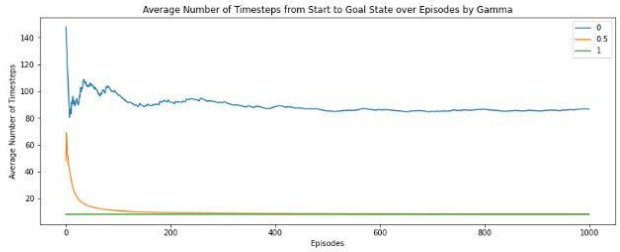

Figure 8: Average Number of Timesteps by Gamma

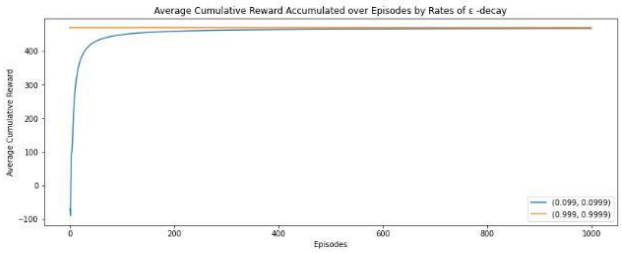

Figure 9: Average Cumulative Reward by Rate of Decay

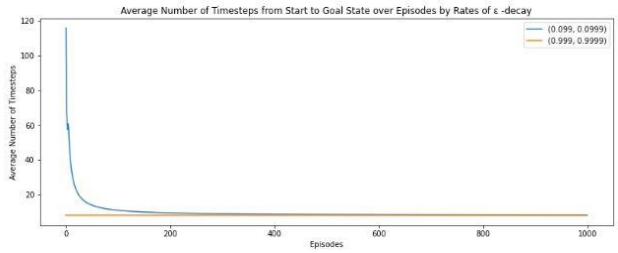

Figure 10: Average Number of Timesteps by Rate of Decay

5.2. Grid search over parameter values and policies

The parameters and policies are varied in tandem, and grid search is carried out to identify optimal parameter values and policies by measuring the average cumulative reward accumulated, average number of timesteps required from start to goal state and the rate of reaching the goal state (i.e. the number of times the agent arrives at the goal state in the 1000 episodes of training).

Alpha	Gamma	Rates of Epsilon Decay (Explore, Exploit)	Average Cumulative Reward	Average Number of Episodes	Rate of Reaching Goal State
0	0	(0.099, 0.0999)	17.445	79.6	0.999
0	0	(0.999, 0.9999)	13.055	83.946	0.999
0	0.5	(0.099, 0.0999)	4.64	84.262	0.998
0	0.5	(0.999, 0.9999)	5.58	86.604	1
0	1	(0.099, 0.0999)	-6.86	86.174	0.996
0	1	(0.999, 0.9999)	20.54	81.878	1
0.5	0	(0.099, 0.0999)	268.85	42.212	1
0.5	0	(0.999, 0.9999)	269.025	43.38	0.997
0.5	0.5	(0.099, 0.0999)	467.35	8.53	1
0.5	0.5	(0.999, 0.9999)	470	8	1
0.5	1	(0.099, 0.0999)	470	8	1
0.5	1	(0.999, 0.9999)	470	8	1
1	0	(0.099, 0.0999)	129.575	57.582	0.999
1	0	(0.999, 0.9999)	134.8	52.6	1
1	0.5	(0.099, 0.0999)	463.8	8.526	1
1	0.5	(0.999, 0.9999)	470	8	1
1	1	(0.099, 0.0999)	470	8	1
1	1	(0.999, 0.9999)	470	8	1

Table 5: Grid Search of Parameter Values and Policies

6. Results analysis

Section 5 suggests that different parameters, parameter values and policies impact the performance but to varying degrees.

As mentioned in Section 3, α represents the learning rate and dictates how fast learning takes place and Figures 5 and 6 concur with our expectations. Namely, when α is 0, Q-values are never updated as the Q-learning update simplifies to $Q^{new}(s_t,a_t) \leftarrow Q(s_t,a_t)$ and thus, the agent never learns about the environment and moves aimlessly throughout the environment. This is evident by the average cumulative reward accumulated of 0 (single high positive rewards are balanced out by multiple smaller negative rewards) and requiring 80 timesteps, on average, to reach the goal state which is significantly more than other α values. Even with an α of 0.5, the agent consistently accumulates more rewards on average and requires less timesteps to reach the goal state; the agent is still learning about the environment and Q-values are updated but at a slower rate compared to higher values of α (e.g. α of 1) and this is evident from the: increase in average cumulative reward accumulated, decrease in average number of timesteps over the episodes and eventual convergence.

We have mentioned that γ represents the discount factor and determines the importance of future rewards. Similarly, Figures 7 and 8 supports our understanding. Since we set α to be 1, a γ of 0 reduces our Q-learning update rule to $Q^{new}(s_t,a_t) \leftarrow r_{t+1}$. In other words, the agent only considers immediate rewards and in the context of this problem, the agent will view all states, that are not the resource or goal state, to be the same. From the figures, we get a similar view as to when discussing the impact of α, though marginally worse. On average, the agent accumulates around -20 in rewards instead. Comparing this to that of other γ values, we see that this is significantly worse. With a γ of 0.5, the agent still prefers immediate rewards but has some considerations of future rewards. For example, from cell "I", cell "H" which is adjacent to cell "M" will be viewed more favourably as compared to cell "D". And with higher values of γ, this is more evident. Even with low but non-zero values of γ, the agent is learning about the environment and eventual convergence of Q-values, but at a slower rate.

The rate of ε-decay orchestrates how fast the agent is more likely to favour exploitation versus exploration. With a slower ε-decay (i.e. scalars of 0.9999 and 0.999), exploration is more likely to be favoured for more timesteps and episodes versus a faster ε-decay (i.e. scalars of 0.099 and 0.0999), where the agent is more likely to transition to favouring exploitation earlier on. We should expect a policy that favours exploration early on to outperform policies that favour exploitation early on since our Q-values are mostly zeros initially and exploitation runs the risk of non-optimal solutions, especially when high positive rewards are discovered early on. Figures 9 and 10 shows that with both policies, there is eventual convergence but at a slower rate for a fast ε-decay policy. This is consistent with our expectations but with fewer training episodes and timesteps, and a more complex environment with multiple resources and large positive rewards early on but followed by even larger negative rewards, we can expect the difference in policy performance to be more apparent.

In our grid search, we also measure the rate of the agent arriving at the goal state over the 1000 episodes but we see that most combinations of parameter values and policies still result in the agent arriving at the goal state. Additionally, we are further shown that in the context of this study, the rate of ε-decay did not play a large role in affecting the average cumulative reward and number of timesteps required. Again, with less timesteps and episodes, and a more complex environment, the impact of the different policies and impact onto the rate of arriving at the goal state is likely to be more apparent. However, for this study, we can see that α played a larger role in the performance of the agent, as compared to γ. Generally, we see that experiments with high α's and low γ's outperformed those with low α's and high γ's. One exception though, is when α is 1 and γ is 0. There we see it being outperformed by experiments when α is 0.5 and γ is 0; an example of when learning is fast but not accurate. This is likely due to the Q-value update rule where α is multiplied onto the entire error term and with α = 0, regardless of the value of γ, the error term will still be zero. This is shown in the first 6 entries in our grid where the average number of timesteps is around 84, with an average cumulative reward of around 9.

Advanced

7. Implementing Deep Q-Learning/Deep Q-Network (DQN) with two improvements

7.1. Environment

The environment we'll work with is the CartPole environment from gym.openai's list of environments. The agent controls the cart and looks to balance the pole, that originally stands vertically in the middle of the cart, by moving left and right. Figure 11 provides an illustration of the environment, where the cart and pole are coloured in black and brown respectively.

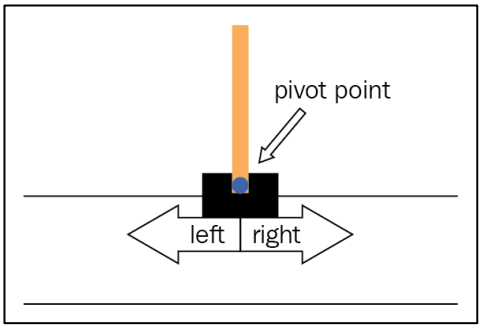

Figure 11: CartPole Environment

The objective of our agent is to maximise its return (accumulated reward) by moving left and right to keep the pole upright for as long as possible. Following this, an optimal policy will be one which balances the pole for as long as possible whilst ensuring not to exceed any terminal thresholds.

7.2. State transition function

The state transition function is given by: $s_{t+1} = \delta(s_t, a_t)$. That is that the next state at timestep t+1, s_{t+1}, is a function of its current state, s_t, and action at timestep t, a_t. The state space is given by a 4D vector representing the cart position and velocity, and the angle and angular velocity of the pole. Additionally, the action space, A, is made up of two actions: pushing the cart right (given by +1) or left (given by -1). An episode of training is terminated when one of the following is true: pole angle exceeds 12° in either direction, cart position exceeds 2.4 in either direction, or after 200 timesteps.

7.3. Reward function

The reward an agent gets is given by: $r_{t+1} = r(s_t, a_t)$. Namely, the reward at the next timestep t+1, r_{t+1}, is a function of its current state, s_t, and choice of action at timestep t, a_t. Specifically, the agent will incur a reward of +1 for every timestep that the pole remains upright, and 0 otherwise.

7.4. DQN

With this environment, standard Reinforcement Learning (RL) techniques may not suffice. Though possible, it is practically inefficient and logistically infeasible to represent the rewards and Q-values in a tabular format (e.g. a matrix). Thus, we employ DQN which utilises a deep neural network as a function approximator to estimate the Q-values from state observations. However, training the network sequentially on a set of state observations may not be optimal due to temporal dependencies (e.g. the state at t+1 is dependent on the state at t). To get around this, we adopt Experience Replay. With it, we build a replay memory, D, of capacity N which stores a set of experiences, e_t, and replaces older experiences with newer ones when the capacity is reached. With each experience, we observe the state, action and reward at time t, and state at t+1. In other words, $e_t = (s_t, a_t, r_t, s_{t+1})$. From our replay memory, we are able to randomly sample a minibatch of experiences instead.

When training, we intitialise a replay memory to capacity N and a neural network with random weights. For each episode, we iterate over a set of timesteps where we select an action based on our behaviour policy, execute it in our environment and observe the reward and next states. From there, we store our experience in our replay memory and randomly sample a minibatch of experiences. With our minibatch, Q-values and target Q-values are obtained by parsing current and next states through the DQN respectively. Target Q-values are derived by multiplying the maximum Q-value of the next states by a discount factor, γ, and adding the reward, r_t. Gradient descent is then performed on the loss (computed by the Mean Squared Error) between the derived Q-values and target Q-values. Gradient clipping was also adopted for stability in our DQNs.

7.4.1. Double-DQN

One potential drawback of using a standard DQN is the risk of overestimating Q-values as the same Q-network is used to select and evaluate an action. To circumvent this, a Double-DQN architecture may prove useful. In a Double-DQN, we set up two networks, namely an online and target network for action selection and value evaluation respectively. Additionally, during training, the Q-values shift while our target value shifts as well and to get around this, on every N episodes, we update our target network parameters to that of the online network.

7.4.2. Dueling-DQN

Another potential improvement is the use of a Dueling-DQN instead. With a standard DQN, the Q-network learns on combinations state-actions but with a Dueling-DQN, the problem is separated into two estimations: the state value V and advantage A. V measures how good it is to be in the given state and A provides a relative measure of importance of each action (or how advantageous selecting an action is compared to others at the given state). The two streams (A and V) can then be merged and aggregated (after normalising A by its mean) to get an estimate of the Q-values. The surrounding motivation is that it may not be necessary to know Q-values of each action at every timestep (e.g. a state-action pair which does not impact the environment in a relevant way).

7.5. Q-learning and Q-network parameters and policy

Tables 6 and 7 below provides a summary to the Q-Learning and Q-Network parameters and policy used.

Parameter		Value
γ		0.999
Behaviour Policy		ε-greedy
ε-Decay	ε	1
	Rate of decay	0.999
	Minimum ε	0.001
Target Policy		Greedy
Number of Episodes		1000
Number of Timesteps		Until terminal

Table 6: Q-Learning Parameters and Policy

Parameter	Value
α	0.01
Capacity	100000
Batch Size	256
Number of Hidden Layers	1
Size of Hidden Layer	64
Optimiser	ADAM
Activation Function	ReLu
Number of Episodes to Update Target Network	10

Table 7: Q-Network Parameters

7.6. Results

The experiment was repeated but with varying improvements to the standard DQN. From there, the best performing DQN is selected and a set of its hyperparameters are tuned. The results are given below:

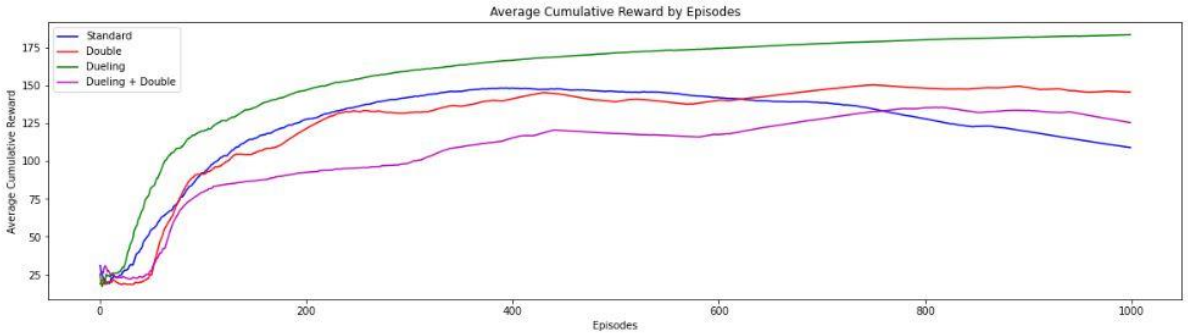

Figure 12: Average Cumulative Reward Accumulated by DQNs

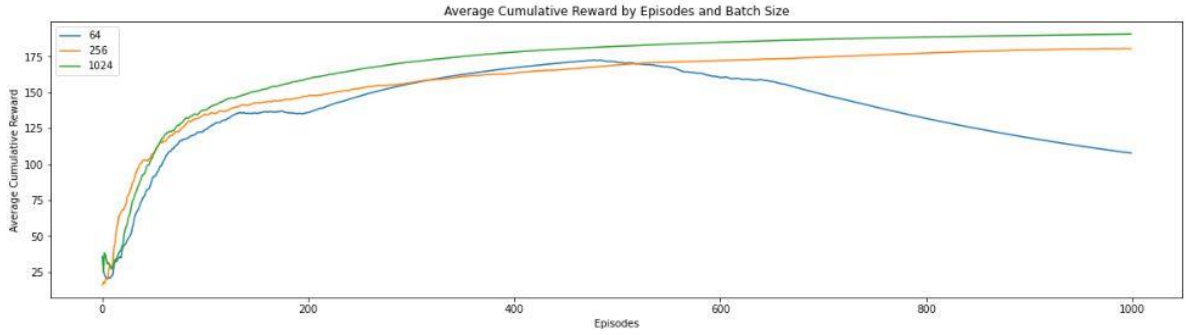

Figure 13: Average Cumulative Reward on Dueling-DQN by Batch Size

8. Results Analysis

Overall, the suite of DQNs performed decently on the CartPole environment, with better performance from the adoption of improvements (i.e. Double, Dueling, Double and Dueling) to the standard DQN.

The standard DQN serves as an experiment to see how it would perform on the CartPole environment with no improvements and from Figure 12, we see that it performs decently up until the 400th episode where its performance plummets down, up until the last training episode. This could be the result of instability of the neural network (despite gradient clipping) from the Q-values and target values shifting in the same direction and/or adverse updates to the neural network's parameters (e.g. weights).

Indeed, by adopting a Double-DQN, we potentially account for some of the shortcomings mentioned above and see improvements in the average cumulative reward. As mentioned in Section 6.6, Double-DQN accounts for the overestimation bias that may exist in a standard DQN and for the context of this environment, it seems to be sizeable and prevalent given the average improvement of 50 additional timesteps (of keeping the pole upright) by the end of the training; though the improvements may be larger with more training episodes. In the context of CartPole, overestimation bias may lead to unstable training and poor-quality policies where Q-values are overestimated, adverse actions are taken (e.g. moving too far to the right/left) and the pole ultimately falling.

Amongst the list of improvements, we see that Dueling-DQNs consistently outperformed other DQNs over all the training episodes. We see an average improvement of around 75 and 25 additional timesteps from the standard DQN and Double-DQN respectively by the end of training. This suggests that splitting the Q-value estimation into two streams, state value V and advantage A, may be beneficial. Indeed, thinking of the CartPole environment, there exists states where the agent's action does not affect the environment in a relevant way, though maybe not as prevalent as in games like Enduro. For example, imagine states where the pole is toppling over and quickly approaching the terminal thresholds. Then regardless of the action of the agent, it will still likely result in the pole toppling over. Thus, Dueling-DQN may be appropriate for CartPole. By recording the agent's performance over each episode, we see a vastly different performance from the start of training to end of training. Early on in training, it was found that the pole topples over naturally with minimal to no corrective action by the agent. Nearing the final training episodes, the agent instead continuously makes multiple minor adjustments to its position to keep the pole upright. Interestingly, it should be noted that incorporating both improvements (Double and Dueling) had adverse effects with it being consistently outperformed by Double-DQN and Dueling-DQN but still outperformed the standard DQN.

That being said, the above results are not entirely conclusive as the DQNs were trained on a set of parameters and policies stated in Section 6.8 with no hyperparameter tuning as of yet. With more optimal hyperparameters (e.g. α, γ, frequency of target network update, shape of neural network, etc), we may find that Dueling-DQNs are outperformed by other DQN improvements but training DQNs are computationally intensive. Additionally, other state spaces and inputs to the neural networks can be considered as well - for example, difference in screen captures between timesteps rather than a 4D vector. With additional capacity, it would be to ideal to perform a grid search on the DQN and hyperparameter space and consider different state spaces and inputs. Nonetheless, Figure 13 shows how the Dueling-DQN performance varies by batch size whilst keeping all else fixed. We should expect for larger batch sizes to result in better performance over smaller ones. With larger batch sizes, the loss between Q-values and target values are more stable as individual losses are averaged out over the large batch size. This ultimately results in more stable neural networks. Indeed, Figure 13 shows that a batch size of 1024 marginally outperforms that of 256 by about 5 additional timesteps, while both greatly outperforms the batch size of 64. In fact, we see a decline in performance after 500 episodes with the batch size of 64 which may be a result of instability and adverse updates to the neural network's parameters, as mentioned earlier. However, one may need to weigh the trade-off between marginal improvements in performance and time taken for training, as larger batch sizes may be more costly from the forward pass of the neural network.

9. Implementing DQN on an Atari Learning Environment

9.1. Environment

The Atari learning environment we'll work with is the Breakout environment as shown in Figure 14. The agent controls the rectangular paddle and looks to send the ball into the rainbow set of bricks (reward of +1) whilst keeping the ball in play.

The objective of our agent is to maximise its return by moving left, right, or not moving at all, to intercept the ball and bouncing it towards the rainbow bricks above. An optimal policy will be one which clears as many rainbow bricks prior to running out of lives.

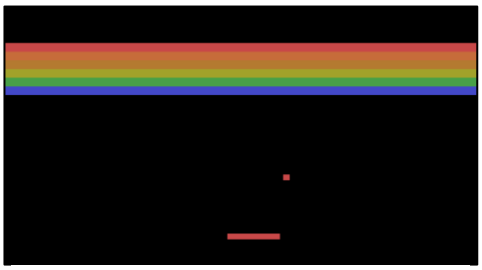

Figure 14: Breakout Environment

The state space is given by an RGB image, represented by an array of shape (210, 160, 3). Additionally, the action space, A, is made up of four actions: not moving, firing (only done when resetting the environment to fire the ball from the paddle), moving left and right. An episode of training is terminated when all 5 lives are lost or all the blocks are cleared.

9.2. DQN

Similar to in Section 7, with the Breakout environment, standard RL techniques may not suffice. Again, though possible, it is practically inefficient and logistically infeasible to represent the rewards and Q-values in a tabular format (e.g. a matrix). Thus, we employ DQN and a suite of improvements as will be further elaborated in the following subsections. An overview of DQN has been discussed and can be found in Section 7.4.

9.2.1. Double-DQN

As highlighted in Section 7.4.1, standard DQNs run the risk of overestimation bias and employing a Double-DQN mitigates the risk by setting up two networks: an online and target network for action selection and value evaluation respectively. In the context of the Breakout environment however, Double-DQNs may not be as impactful as our findings with CartPole. With Breakout, overestimation bias may still be a detriment to the agents ability in intercepting the ball but if it manages to, then the agent's action is no longer relevant (until the ball bounces back) and may be met with just scoring fewer points (i.e. hitting less bricks) or getting less rewards than expected or at worst, bouncing the ball off of the sides without collecting any reward. Contrasting this to CartPole where the agent's action continuously impacts the state space, overestimation bias may lead to the agent positioning the cart (and the pole) in positions that ultimately lead to the pole falling over. Nonetheless, the performance of a Double-DQN will be explored.

9.2.2. Dueling-DQN

As mentioned in Section 7.4.2, another potential improvement is the use of a Dueling-DQN instead which separates the estimation into state value V and advantage A, and merges them to get an estimate of the Q-values. V measures how good it is to be in the given state and A provides a relative measure of importance of each action (or how advantageous selecting an action is compared to others at the given state). With Breakout, Dueling-DQNs can be suspected to be fairly beneficial and appropriate as there are numerous state-action pairs which do not impact the environment in a relevant way. For example, consider when the ball is travelling from the paddle towards the bricks, then any action by the agent is redundant. Alternatively, consider instances when the ball tunnels and repeatedly bounces between bricks and the sides. Then here the environment is being impacted but the action of the agent is irrelevant. Section 9.6 will assess the performance of a Dueling-DQN.

9.2.3. Prioritised Experience Replay (PER)

In addition to the abovementioned improvements, PER can be considered as a further improvement to the standard DQN. With PER, each experience is assigned a priority according to its temporal difference error (TD-error) and experiences are sampled according to its priority (higher probability of being sampled for experiences with higher priority), rather than uniformly. The surrounding motivation is that some experiences are more important than others (in terms how much is learnt) and with uniform random sampling, these important experiences are sampled very rarely. This problem is further exaggerated in environments with sparse rewards or when considering that recent states only occur rarely. That being said, this introduces bias to the sampling (i.e. high priority samples are sampled more frequently) and thus importance sampling weights are incorporated to account for it. High priority and low priority samples will have lower and higher importance sampling weights respectively. PER introduces two new hyperparameters, α and β which dictates how much randomness is incorporated into sampling and controls how much importance sampling weights affect the learning respectively. With Breakout, PER may be appropriate as well when considering experiences where the paddle manages to intercept the ball as compared to experiences when the ball is mid-flight from paddle the brick.

9.3. Proximal Policy Optimisation (PPO)

Apart from Q-learning algorithms, policy optimisation algorithms can be considered as an alternative – PPO in particular. Rather than estimating action values, PPO looks to estimate a parameterised policy, π_θ, instead by maximising the expected return and optimising it by updating the parameters, θ, of the neural network via gradient ascent (i.e. in the direction of the policy gradient). What's more, to account for potential instability in the neural network, updates are clipped to avoid big changes in the policy by some bound, ε. The intuition is that with some environments, learning about the policy may be easier and more obvious than the action values. And with Breakout, the policy seems obvious: intercept the ball regardless of what it collides with (since there is no penalty to the ball colliding against the walls). However, PPO is an example of an on-policy method and

hence is not sample efficient. Namely, new trajectories (a series of states, actions and rewards) will need to be sampled on every episode. Thus, computational intensity will need to be considered.

9.4. Results

The experiment was repeated with varying improvements to the standard DQN via tune, using the default hyperparameters prescribed and over 100 episodes. Unfortunately, due to the computational intensity, a PPO was unable to be trained but the code and snapshots of training iterations can be seen in the code pdf file. Nonetheless, from there, the best performing DQN is selected and a set of its hyperparameters are tuned. The results are given below:

Improvements			Mean
PER	Double	Dueling	Reward
True	True	True	2.67
True	False	True	5.36
True	True	False	5.68
True	False	False	6.62
False	True	True	1.91
False	False	True	3.62
False	True	False	1.83
False	False	False	2.37

Table 8: Grid Search of DQN Improvements

α	β	Mean Reward
0.1	0.0	3.42
0.6	0.0	5.1
1.0	0.0	4.71
0.1	0.4	4.31
0.6	0.4	3.23
1.0	0.4	4.78
0.1	1.0	3.99
0.6	1.0	1.95
1.0	1.0	2.33

Table 9: Grid Search of PER on α and β

10. Results Analysis

Table 8 shows the performance of different combinations of improvements to the standard DQN. From it we see Dueling-DQN and DQN with PER outperforming the standard DQN. In fact, DQN with PER performed the best amongst the different combination of improvements with a mean reward of 6.62. This could be a result of there being a few sets of important experiences, as mentioned in Section 9.2.3. Additionally, Dueling-DQN performed better than the standard DQN, as expected in Section 9.2.2, with a mean reward of 3.62 compared to 2.37. It should be noted though that in this experiment, Double-DQN performed worse than the standard DQN, potentially for the lack of overestimation bias as discussed in section 9.2.1, but without optimal hyperparameters for each set of DQNs and with only 100 training episodes, these results have a number of caveats and are far from conclusive. With more capacity, it would be ideal to tune the DQNs individually over a suite of hyperparameter values and policies, and over longer training periods.

Nonetheless, Table 9 explores how performance of DQN with PER varies with different values of α and β. As mentioned in Section 9.2.3, α dictates how much randomness is incorporated into sampling while β controls how much importance sampling weights affect the learning. The closer α gets to 0, the closer it is to uniform random sampling and thus regardless of the value of β, the sampling weights for all experiences are uniform which is what we see in standard Experience Replay. Conversely, as α approaches 1, we'll only see the highest priority experiences be sampled. On the other hand, a β that is close to 0 suggests that all experiences are weighed equally which will fail to account for the sampling bias that arises from high values of α. A β that approaches 1 indicates that all experiences are weighted by the inverse of its probability of being sampled. Throughout training, β is annealed up to a high value (usually 1).

Indeed, from Table 9, we see that when β is 0, an α of 0.6 performs best as it's likely able to strike a balance between capturing prioritisation in the experiences and sampling bias. With an α of 1, highest priority experiences are sampled but without a counterbalancing factor (since β is 0), sampling bias will be present. Conversely when β is 1, an α of 0.1 performs best due to potentially counterbalancing the high β and inching the importance sampling weights closer towards a uniform one. From the other point of view, we see that a β of 0.4 performs best for α's on both ends of the spectrum (i.e. 0.1 and 1). This further reinstates the trade-off in PER between being able to sample experiences based on importance and incurring sampling bias. That being said, the results would be more conclusive with more training episodes and tuning of other hyperparameters.