Solving the Mountain Car Problem with Reinforcement Learning

S.B. & A.M.

1 The Mountain Car Problem

In this paper, we apply reinforcement learning methods to the 'MountainCar' environment on OpenAI Gym [1], illustrated in figure 1. The agent in this environment is the car, and the goal is to reach the flag. The actions available to the agent are push left, push right and no push. There are two variants of this environment, in one these actions are discrete whilst in the other they are continuous. The challenge in this environment is that the agent cannot reach the flag just by pushing right, as it cannot build enough momentum to climb the hill this way. However, it can build enough momentum from first ascending the hill on the left, and then pushing right from there. This is an example of a control task in which an agent needs to initially move in the opposite direction of a target position, such tasks tend to be challenging without a human designer [2]. After running a baseline model, we will perform experiments to investigate how varying parameters effects the performance of our agent. Below, we define the model parameters for solving the Mountain Car problem with the Q-learning algorithm when the actions are discrete.

State Transition Function

In this environment, any given state at time t (S_t) is represented by a pair of values, position (x) and velocity (v). Both of these values are from closed, bounded sets. Formally:

S_t=(x,v), where x $\in$ [-1.2,0.6] and v $\in$[-0.7,0.7]

The goal state is reached when x=0.5. Note that both elements representing a state are continuous variables. When using Q-learning, the environment needs to be represented as a Markov Decision Process. Therefore, experimenting on this environment will require us to discretise the state space. We have chosen 25 discrete values for both position and velocity. So for example, the position state between -1.2 and -1.164 is treated as one discrete space. We have chosen 25 arbitrarily, a number too small would over simplify the environment and result in very limited exploration by the agent, whilst a number too large would better resemble the true state space resulting in more effective exploration however at the expense of lengthier computational times.

We formally define the state transition function as:

$S_{t+1} = f(S_t, A_t)$

The agent's state at time t+1 is a function of it's current state, S_t, and the action it takes from there, A_t. The action space at time step t is defined below:

A_t=[push left, do nothing, push right]

Reward Function

The reward is a key component in the agent's learning process as we want the agent to maximise it's accumulated reward during the learning process.

On OpenAI Gym [1], the default reward for each action the agent takes is -1. An exception is when the action taken leads the agent to reach the target, for which the reward is zero. So, the more steps taken to reach the flag, the lower the reward accumulated. Thus, maximising accumulated reward in our environment translates to taking less steps. Formally, the reward function is defined as:

$R_{t+1} = -1$, except when the agent reaches the goal state.

The discounted cumulative reward is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

$$= \sum_{k=0}^{T} \gamma^k R_{t+k+1}, \ where \ 0 < \gamma < 1$$

Above, γ refers to the discount rate. This parameter quantifies the importance that the agent gives to future rewards when taking an action. If $\gamma = 0$, the agent only considers immediate rewards. If $\gamma = 1$, future rewards are not discounted at all. As γ is decreased, future rewards are discounted more, thus reducing the influence they have over the agent when choosing an action.

Policy

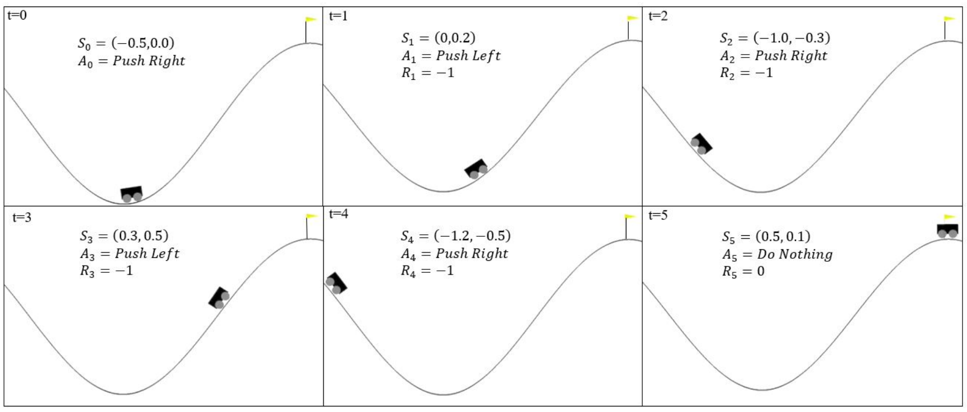
An agent's policy (Π) is defined as the strategy it employs to decide what action to take given it's current state, and can be thought of as mapping the state spaces to action spaces. Policies can either be deterministic or stochastic in nature. Under stochastic policies, probabilities are assigned to each action, which quantify how likely it is that a given action is taken. An agent can either explore it's environment by randomly selecting actions to see what rewards are earned, or by exploiting what it already knows and picking actions which maximise rewards, referred to as a greedy policy. A policy which allows the agent to manage the extent of exploration versus exploitation is known as the ϵ-greedy policy, which is the one we will adopt. In this policy, ϵ is parameter which needs to be defined, and it refers to the probability that an agent explores the environment, i.e. randomly selects an action.

It is good practice to allow an agent to initially explore it's environment, because an entirely greedy approach may prohibit an agent from discovering better actions. However, the effectiveness of ϵ-greedy depends on how much rewards vary in an environment. The greater the reward varies, the more exploration is required for an agent to navigate through state-action pairs and find the best actions. Greedy policies then, may be more effective in environments where rewards do not vary much [2]. Given that in our environment, the reward remains constant unless the goal is reached, we will set our ϵ value to be equal to 0.5.

Another optional parameter under the ϵ-greedy policy is λ, the rate of decay. This is the value by which ϵ is reduced by after each episode. The motivation for this is that it's desirable for our agent to gradually exploit what it has learned, whilst still exploring albeit at a lower rate. An ϵ_{min} can also be chosen as a threshold, to avoid employing an entirely exploitative strategy.

2 Graphical Representation

Figure 1: Illustration of a learning episode



In figure 1 we have illustrated a learning episode lasting 6 steps. It also illustrates the reward structure.

In Q-learning, an R-matrix is a tabular representation of the reward function and stores the immediate rewards that an agent receives for all possible actions in a state. Each row in the matrix represents a state and the corresponding reward it receives when transitioning to another state is represented in the columns. No reward can be assigned for a state transition that is not possible.

In the default OpenAI MountainCar environment, as the state spaces are continuous, representing the reward structure as a matrix would not be possible. Our discretised environment has 625 states, and we represent it's reward structure as per the expression below (1):

$$R_{ij} = \begin{cases} -1, & \text{if } \exists\ i \to j \\ 0, & \text{if x=0.5 in j} \end{cases} \tag{1}$$

In the above expression, i represents the agent's current state and j represents a state to transition to. For all possible transitions the immediate reward is -1, and 0 when the agent reaches the target flag position at x=0.5.

3 Initial Parameters for Q-learning

Q-learning is an algorithm which uses temporal-difference learning in environments that can be represented as an MDP. The learning is an iterative process which is formally defined below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \text{max}_A Q(S_{t+1}, A) - Q(S_t, A_t)] \tag{2}$$

We define the additional parameters below:

- Q-value $Q(S_t, A_t)$: This variable is the state-action value function under a given policy, and represents the expected discounted cumulative reward of taking an action in a given state, as per G_t. This is initialised to zero for all state-action pairs, and updated iteratively.
- Learning rate (α): This parameter determines the extent to which the algorithm updates Q-values, i.e. the pace of it's learning, and ranges between 0 and 1. A value of 0 means the agent learns nothing at each iteration, and a value of 1 means the agent completely discards previously learned Q-values.
- $\text{max}_A Q(S_{t+1}, A)$: This variable represents the maximum expected future cumulative reward from each possible state-action pair.

A summary of all parameters in our baseline model is presented in table 1.

Table 1: Baseline Model Parameters

Parameter	Value	Comments
Learning Rate (α)	0.5	It's difficult to determine at the outset what is a good learning rate, but we will use 0.5 initially.
Discount Rate (γ)	0.8	We have chosen a high discount rate, as we suspect given the reward structure the agent needs to give more weight to future rewards.
Policy	Epsilon-greedy	As per policy section.
Epsilon (ϵ)	0.5	As per policy section.
Epsilon Decay	0.002 per episode	We will reduce epsilon each episode so the agent adopts more exploitative actions.
Minimum Epsilon (ϵ_{min})	0.01	This minimum will be reached after 245 episodes.
Number of episodes	5,000	
Max number of steps per episode	25,000	Default is 200 on OpenAI however we have increased to allow sufficient exploration.

4 Q-Matrix Values

For the chosen number of bins in our experiment (25 for both position and velocity) we end up with a Q-matrix of size 625x625 which makes it unfeasible to demonstrate the iterative updates in the matrix. Moreover, the environment allows the possibility of transitions to the same state using all three available actions. Using a matrix to represent Q-value updates given the aforementioned point would not allow us to separately store Q-values in the case when different actions result in transitioning to the same state. To circumvent this, we implement learning using a dictionary structure in Python through which we construct unique key-value pairs for each state-action respectively. This allows us to store and update Q-values separately for each corresponding state-action pair.

Under this implementation, the updates to Q-values follow the same process that would otherwise occur in a Q-matrix, and we demonstrate this over 5-time steps in table 2 for the first episode using the listed steps.

1. Initialise all Q-values as zeros and set the required parameters as per section 3.
2. Choose an initial action from A=[0,1,2] according to the ϵ-greedy policy.
3. Perform the action and receive reward R_0 and enter next state S_1.
4. Determine the future reward as $\text{max}_A Q(S_{t+1}, A)$ and update the Q-value for $Q(S_t, A_t)$ using equation 2.
5. Repeat steps 2-4 till the goal state is reached or 25,000 steps have been taken.

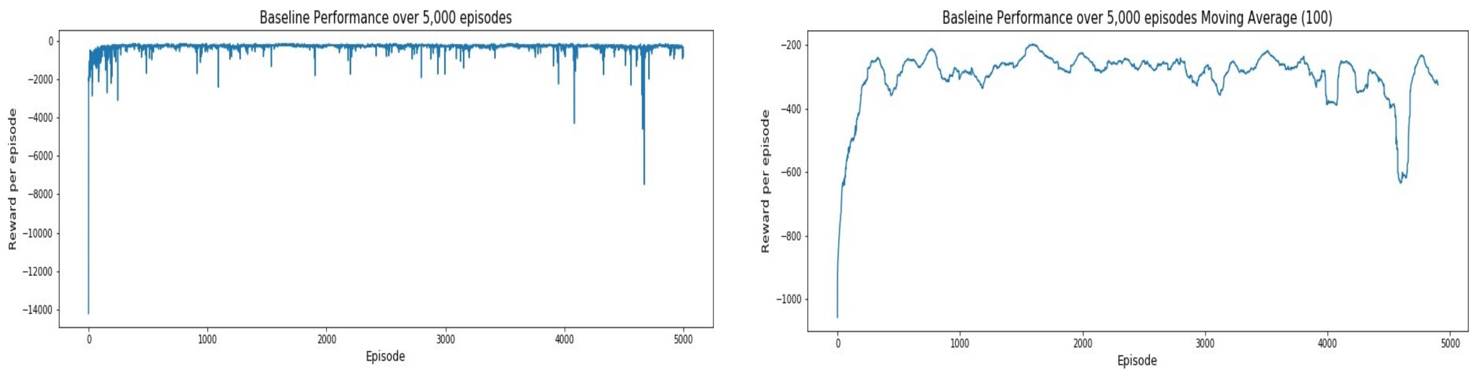
Table 2: Illustration of updating Q-Values

Time Step	**Variables for each Time Step**	**Dictionary Key**	**Dictionary Value** $Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_A Q(S_{t+1}, A) - Q(S_t, A_t)]$
t=0	$S_0 = (24, 24),\ A_0 = 0,\ R_1 = -1$ $S_1 = (10, 12),\ \max_A(S_1, A) = 0$	$Q\{(24, 24), 0\}$	0 + 0.5(-1 + 0.8 x 0 - 0) = **-0.5**
t=1	$S_1 = (10, 12),\ A_1 = 0,\ R_2 = -1$ $S_2 = (10, 12),\ \max_A(S_2, A) = 0$	$Q\{((10, 12), 0\}$	0 + 0.5(-1 + 0.8 x 0 - 0) = **-0.5**
t=2	$S_2 = (10, 12),\ A_2 = 1,\ R_3 = -1$ $S_3 = (10, 12),\ \max_A(S_3, A) = 1$	$Q\{(10, 12), 1\}$	0 + 0.5(-1 + 0.8 x 0 - 0) = **-0.5**
t=3	$S_3 = (10, 12),\ A_3 = 2,\ R_4 = -1$ $S_4 = (10, 12),\ \max_A(S_4, A) = 2$	$Q\{(10, 12), 2\}$	0 + 0.5(-1 + 0.8 x 0 - 0) = **-0.5**
t=4	$S_4 = (10, 12),\ A_4 = 1,\ R_5 = -1$ $S_5 = (10, 12),\ \max_A(S_5, A) = 0$	$Q\{(10, 12), 1\}$	-0.5 + 0.5(-1 + 0.8 x -0.5 - (-0.5)) = **-0.95**

In the above demonstration, the agent begins at state (24,24), and takes action 0. This takes the agent to state (10,12), from which the agent takes actions 0, 1, 2 and then 1 again, all which result in the agent remaining in state (10,12). However due to the key-value structure, separate Q-Values are stored for each state-action pair.

5 Baseline Results: Performance vs Episodes

Figure 2: Baseline Results



(a) Actual Results (b) Smoothed Results

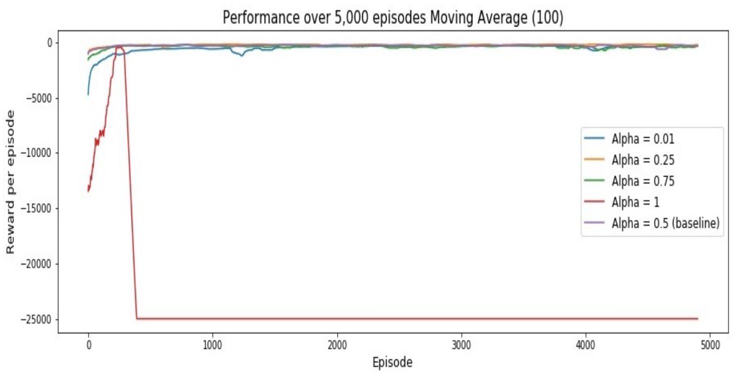
In figure 2a, we have plotted the total reward per episode. Figure 2b represents the same information as a moving average of order 100, which represents the mean of 100 episodes. This enables us to detect any underlying trend of performance over time.

A reminder that an episode terminates after 25,000 steps, or if the agent reaches the goal state before then. In figure 2a we can see that in the first episode, the reward is equal to -14,000. However over the next 250 episodes, we can see the score rapidly improves, this shows the first phase of our agent's learning. Additionally, it looks as though the agent's performance converges from the 250th episode onwards, with some variance thereafter. We can see this variance much more clearly in figure 2b. Interestingly, the variance does not seem to reduce over time. Owing to the initial epsilon value of 0.5 and the decay rate, the agent is taking exploitative actions with probability 0.99 from episode 245 onwards. This coincides with the point at which performance begins to converge. Despite the convergence, we can see in figure 2a there are instances of a drop in performance at around episode 4,000 and 4,750, which indicates not all Q-values are optimal. This may be due to discretising the state space. This approach limits the agent's interaction with the environment, as we are reducing the actual dimensions of the state space, which may affect the agent's attempt at achieving optimality.

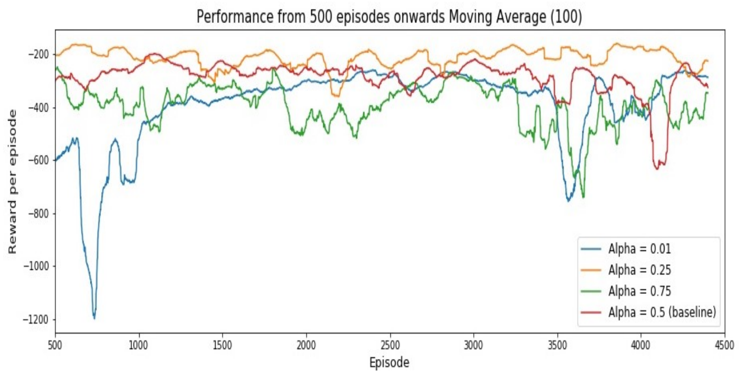
6 Analysing Performance with Varying Parameters

In this section, we vary some of the parameters of our algorithm and analyse how these affect learning performance.

Figure 3: Varying Alpha Results



(a)



(b)

6.1 Alpha

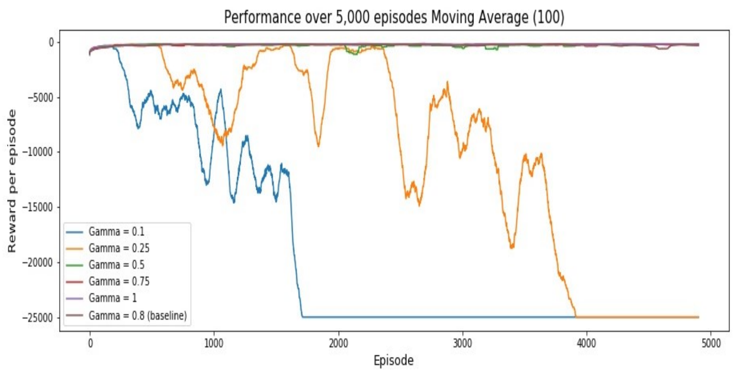
In figure 3a, we have illustrated performance with varying values of alpha. The agent's performance is worst when α=1; we see that the agent's initial reward per episode is much lower compared to the other α values, however it's performance does improve until about episode 300. From here on the agent never reaches the target state, consistently getting the lowest reward of -25,000. When α=1, the agent completely disregards prior Q-values at every time step and is observably poor at learning in our environment, however we cannot fully explicate the agent's behaviour over the learning period in this instance (i.e. the initial learning followed by the drastic drop in performance), and would need to experiment further in order to so.

We observe that the agent takes longer to converge when α=0.01. This can be seen through the gradual increase in accumulated reward from episode 0 to 750. This is expected as the agent gives very little consideration to new information, which means the Q-values are updated very slowly throughout an episode naturally resulting in slower convergence times. In figure 3b we have excluded results for α=1.0, and have a look at performance from 500 episodes onwards for a closer examination. We can see notable drops in performance at various intervals when $\alpha = 0.01$ (episode 750), 0.5 and 0.75 (episodes 3,500 and 4,100). When $\alpha = 0.25$ however we see the performance remains relatively consistent. Whilst keeping other parameters fixed, it seems for our chosen environment that increasing the learning rate improves performance to an extent, after which it diminishes.

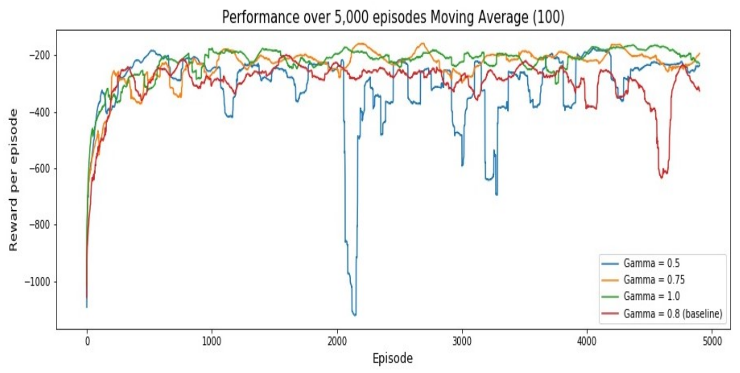
6.2 Gamma

Note, α=0.25 for models in this section as that was the best value indicated above.

Figure 4: Varying Gamma Results



(a)



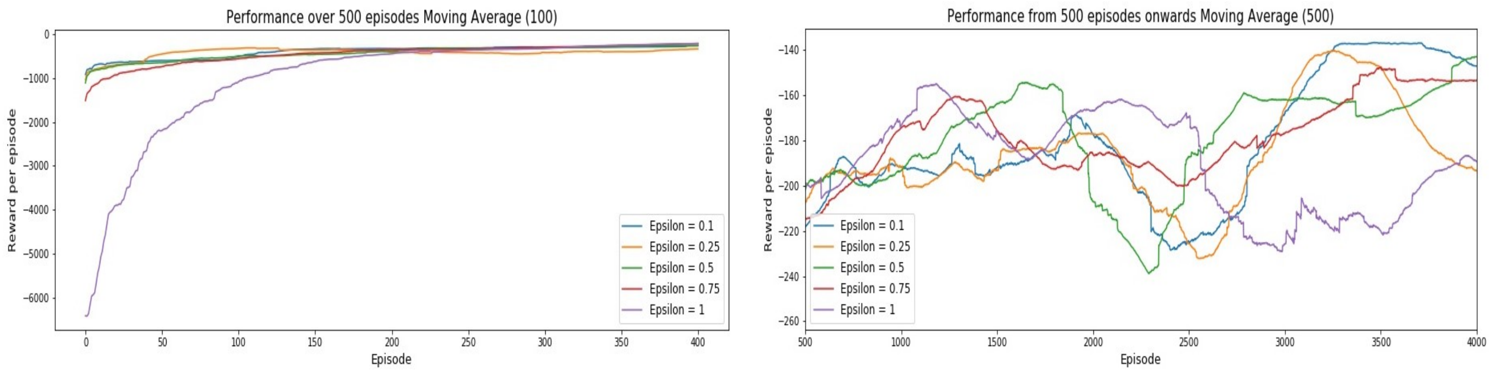
(b)

In figure 4a, we see that low values for γ (0.1 and 0.25) produce poor performance, as the agent fails to increase it's total reward during the learning process. This suggests that the agent in our environment needs to give sufficient importance to future rewards. The agent can only earn a reward of -1 in our environment, thus, if future actions are heavily discounted, then the agent cannot distinguish between actions which reach the target state against those that don't. In figure 4b, we focus on results for where γ is equal to 0.5 and above. In terms of pace of convergence, there isn't much that separates higher values of γ in this regard. We can observe that for higher values of γ the performance of the agent is more stable. For example, when γ=0.5, there is high variance in performance between episodes 2,000 and 3,500. However with higher γ values, such as 1.0, the variance is lower.

6.3 Policy

Note, for the below models, α=0.25 and γ=1.0.

Figure 5: Varying Epsilon Greedy Policy

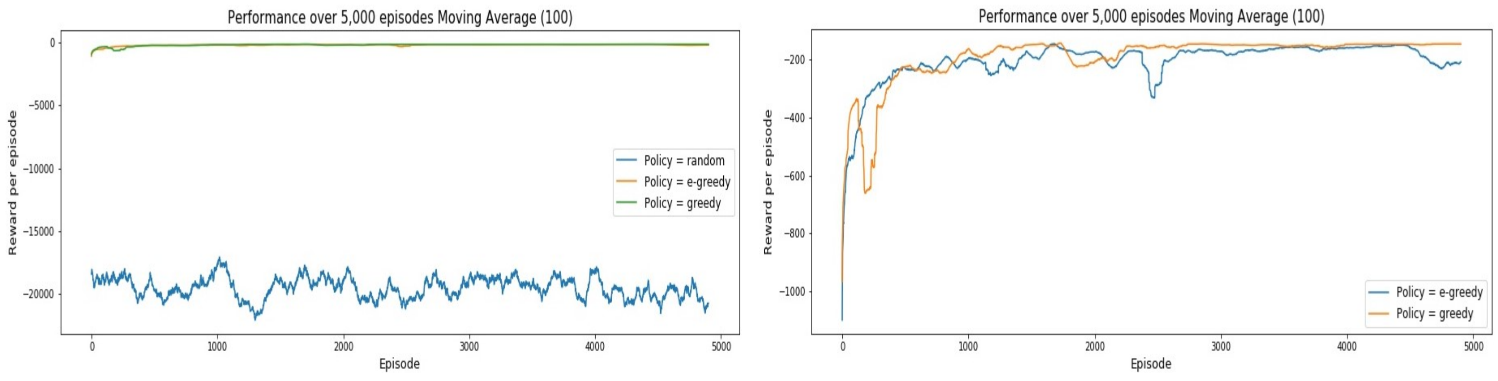


(a) (b)

We hypothesised that due to the mostly constant reward in our environment, a lower initial epsilon value will result in good performance. We have repeated the learning process with varying initial values of epsilon in our epsilon-greedy policy to evaluate our hypothesis, with results illustrated in figure 5. In terms of the first 500 episodes shown in figure 5a, the maximum value of epsilon possible, 1.0, results in the slowest convergence. This is to be expected, as more of the agent's actions will be random.

Note, for analysis post-500 episodes in figure 5b, we have used a moving average of order 500, as it was difficult to distinguish any trends with a lower order. Based on our results it is difficult to discern if any initial value of epsilon is best.

Figure 6: Varying Policy Results



(a) (b)

In addition to the above, we also experimented with different policies and present results in figure 6. In figure 6a we show results with 3 policies, namely random, epsilon-greedy (with ϵ=0.5 initially) and greedy. Random and greedy policies were mentioned in the policy section. We can see that the performance was worst with a random policy, and the agent failed to learn and improve over the episodes. The training time was significantly longer under this policy also.

In figure 6b we can see that the ϵ-greedy policy has a much smoother learning process in the initial episodes, with performance consistently improving. Under the greedy policy however, learning in the initial episodes is not as smooth, with some significant dips in performance around episode 250. From around episode 1,000, the greedy policy overtakes ϵ-greedy in performance and remains so throughout the learning process except between episodes 1,800-2,000.

7 Advanced Methods: Deep Reinforcement Learning

From here on we employ Deep Reinforcement Learning (DRL) methods in the Mountain Car environment. Using tabular Q-learning necessitated the discretisation of the state-action space, and representing this environment through a finely discretised state-action space would increase the size of the Q-value table exponentially. Learning by iteratively updating all entries in such a large table can become computationally expensive. Rather than using a lookup table to determine the Q-value of a state-action pair, DRL methods use non-Linear functions to approximate them. These functions are able to map output to input samples across the state-action space thus overcoming the restriction of a manageable discrete state-action space and provide capability of handling large state-spaces. Using DRL should improve the agent's interaction with our chosen environment and might contribute to better performance. We use two DRL techniques to study the influence they have on the agents learning in two separate scenarios. In one we use a Deep Q-Network (DQN) in a continuous state-space with discrete actions and in the other we use an Advantage Actor Critic (A2C) in a continuous state-action space. This should allow for a more varied comparison of DRL methods while still maintaining comparability to the previous experiment.

7.1 Deep Q-Network

In a DQN, a deep neural network is created that takes a state from the environment as an input, and outputs estimated Q-values for all actions in the action space. The objective of this network is to find optimal Q-values for a given state. Before explaining the learning step, we first introduce the concept of *experience replays*, first applied in the DRL context in [3]. Once the network is initialised with random weights, we begin the initial episode and retain the agent's experiences for several time steps. An experience at time t is formally defined as:

$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$

For a timestep t, we retain information about the initial state the agent was in, the action taken, the immediate reward received from that action, and the next state after taking the action. These experiences are stored in the *replay memory*, and the number of experiences retained in this memory is to be specified. Random samples are then chosen from the replay memory as our training input for the network, referred to as the *memory buffer*. Sampling consecutively would mean our training data is highly correlated, which is undesirable because this can cause high variability when updating the network's weights between neurons. A drawback of taking random samples is that all experiences are given equal importance, whilst in reality some state-action transitions may be more critical than others [4].

For each experience in the memory buffer, we pass s_t into the network, which outputs estimated Q-values for each available action. We focus on the Q-value for the action taken by the agent a_t, stored in the memory buffer. It is this Q-value that we wish to optimise, and we use the expression in the square brackets in the right hand side of the equation (2) for this, by calculating the temporal difference error, defined below:

$$error = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{3}$$

We have the r_{t+1} term in our experience replay, γ is a parameter we specify, and $Q(s_t, a_t)$ is an output from our forward pass to the network. We need to calculate the $\max_a Q(s_{t+1}, a)$ term. This is calculated by passing s_{t+1} from the experience into the same network, and taking the maximum Q-value from its output. This approach makes this algorithm *off-policy*, as it learns via a greedy strategy [4]. Note, it is also still learning via temporal difference as our initial algorithm.

This *error* is what we wish to minimise throughout the learning process, and we do this by tuning the weights of our network, such that if we were to pass the same state, the error would be smaller. We will use an Adam optimiser to update our weights.

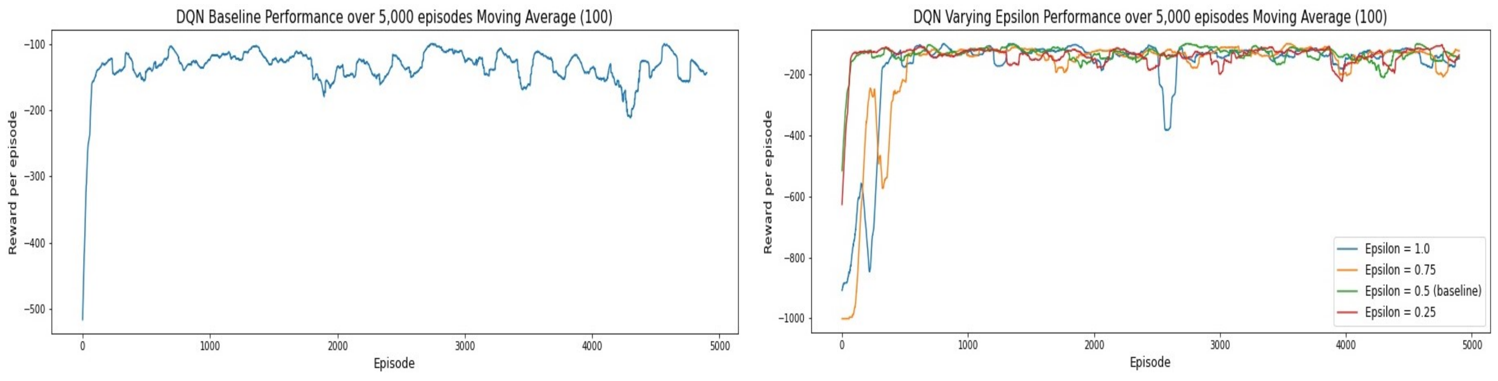
Our motivations for using this algorithm is because up till this point, our discretised state-space meant our Q-values did not fully reflect all possible state-action transitions and hence may potentially limit the agent's capacity to learn an optimal policy. Note, the action space when using this algorithm is still discrete.

In table 3, we summarise the parameters used in our DQN model.

Table 3: DQN Model Parameters

Parameter	Value	Comments
Discount Rate (γ)	0.99	Our prior analysis showed that the agent performs well with a high discount rate.
Epsilon (ϵ)	0.5	We will use a value 0.5 initially and then experiment with this parameter.
Epsilon Decay	0.002 per episode	
Minimum Epsilon (ϵ_{min})	0.01	This minimum will be reached after 245 episodes.
Network Architecture	{1x24, 1x48, relu activations}	2 hidden layers with 24 and 48 neurons respectively.
Learning rate	0.001	A learning rate which is not too large or small.
Replay memory size	20,000	
Memory buffer size	120	For each step, our network will be trained with 120 experiences.
Number of episodes	5,000	
Max number of steps per episode	1,000	We only increased this to 1,000 for DQN as we suspect this will be sufficient for agent to converge.

Figure 7: DQN Results



(a) Baseline DQN (b) Varying Epsilon Value

In figure 7, we illustrate performance from our baseline model as well as results when we vary the initial ϵ value. The epsilon value reaches it's minimum at episode 245. Prior to this point, we see that the agent's performance is increasing at a high rate, but after this point the agent's performance converges to a total reward per episode of about -150, based on an average of 100 episodes. This fluctuates between -200 and -110. In our previous experiment using Q-learning, we see in figure 2b that our baseline model converged to a reward of about -300 per episode, varying between -400 and -225. Thus, DQN converges to a better score by roughly 50%, and also exhibits a lower variance. This is evidence that calculating optimal Q-values via a function approximator over a tabular representation of these Q-values results in significant improvement in performance for the Mountain Car environment. Training time is also another important factor when evaluating algorithms. It took 2 minutes to train our basic Q-learning baseline model. In contrast, it took 150 minutes to train our DQN baseline model. This is a 75-fold increase which is significant. Given our network architecture, there are a total of 1,419 connections between neurons which are updated each episode. Thus, whilst achieving much better performance, the complexity of the DQN results in much longer training times.

In figure 7b, we can see that the agent takes longer to converge for higher ϵ values of 1.0 and 0.75 compared to 0.5 and 0.25. This is expected as the agent is taking random actions for a longer period in the training phase. Interestingly, the agent converges slightly quicker when ϵ=1.0 than 0.75. There is also a dip in performance around episode 2,600 when ϵ=1.0. The quickest convergence is when ϵ=0.25. This suggests that for our environment, the agent operates well taking exploitative actions early on when learning under a DQN. This is in line with our hypothesis that due to mostly constant reward structure, taking more exploitative actions works well. Once converged, it is difficult to say that performance is best for any particular ϵ value. The agent converges to a similar score for all values and exhibits a similar amount of variance in performance also.

7.2 Policy Gradient Methods: Advantage Actor Critic

As mentioned previously, for this section the environment now has a continuous state-action space. The states are continuous as they were in the DQN section. However, now the agent's action space is continuous and can take values between -1 and 1 which dictates the amount of force used to move left or right respectively. The agent also receives a reward 100 upon reaching the goal state. The reward is now defined as:

$$R_{t+1} = \begin{cases} -(applied\ force)^2 \times 0.1, \ applied\ force \in [-1.0, 1.0] \\ 100, \text{upon reaching goal state} \end{cases} \tag{4}$$

Actor-Critic (AC) is a model-free, on-policy DRL algorithm that combines policy gradient methods and value function approximation using two neural networks known as the Actor and Critic respectively. The basic idea is that using a parameterized distribution to represent policy the actor network models a policy that maximises cumulative rewards. It does this by adjusting its weights using gradients obtained from the policy gradient theorem as per equation 5. The critic network approximates a value function (equation 6) using temporal difference learning as per equation 7. In doing so the critic is able to learn the expected sum of discounted rewards for a given state and influences the actor's learning to generate actions that maximise the reward.

$$\nabla_\theta J(\theta) = E_{\pi\theta}[\nabla_\theta log\pi_\theta(s,a)Q_{\pi\theta}(s,a)] \tag{5}$$

$$V_{\pi\theta}(s) = E[R_t|s_t = s] \tag{6}$$

$$V_{\pi\theta}(s_t) - R + \gamma(V_{\pi\theta}(s_{t+1})) \tag{7}$$

$$A_{\pi\theta}(s,a) = Q_{\pi\theta}(s,a) - V_{\pi\theta}(s) \tag{8}$$

The Advantage Actor Critic (A2C) uses a baseline function which helps reduce variance in the policy gradient. Subtracting the baseline function from the policy gradient, it can then be represented using an advantage function as per equation 8. For our experiment we use the state value function as a baseline and a TD error as an estimate of the advantage function, which allows us to replace equation 5 with equation 9.

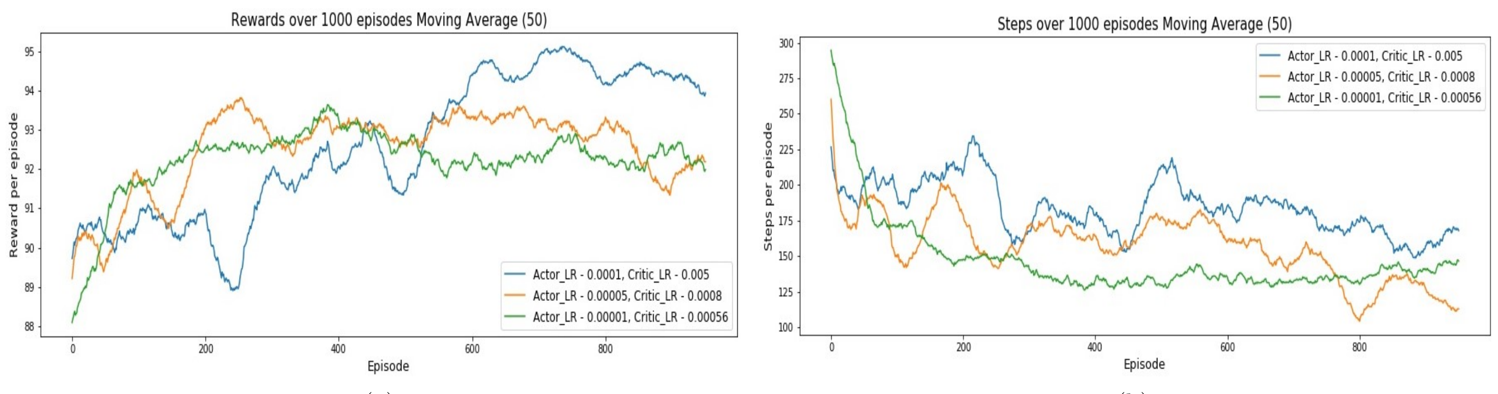
$$\nabla_\theta J(\theta) = E_{\pi\theta}[\nabla_\theta log\pi_\theta(s,a)\delta_{\pi\theta}] \tag{9}$$

For the A2C model the input to the actor network is the continuous state comprised of position and velocity of the car. For the input state the actor network outputs two values as mean and standard deviation of a Gaussian distribution. This distribution is the stochastic policy which is sampled to determine the action to take in the environment yielding response as reward and next state information. Using this response to calculate the TD error, we minimise the critic's error against the target action value function and maximise the actor's objective function using an Adam optimiser to update each network's weights. In this manner the network updates itself for each step across all episodes. The stochastic policy provides an element of exploration needed to the optimal policy. Parameters for our A2C model are detailed in table 4.

Table 4: A2C Parameters

Parameter	**Value**	**Comments**
Discount Rate (γ)	0.99	Our prior analysis showed that the agent performs well with a high discount rate.
Epsilon (ϵ)	0.5	
Epsilon Decay	0.002 per episode	
Minimum Epsilon (ϵ_{min})	0.01	This minimum will be reached after 245 episodes.
Actor Network Architecture	{1x40, 1x40, ELU activations}	2 hidden layers both with 40 neurons.
Critic Network Architecture	{1x400, 1x400, ELU activations}	2 hidden layers both with 400 neurons.
Actor Learning rate	{0.0001,0.00005,0.00001}	We experiment with each of these values.
Critic Learning rate	{0.005,0.0008,0.00056}	As above.
Number of episodes	1,000	
Max number of steps per episode	1,000	We only increased this to 1,000 for DQN as we suspect this will be sufficient for agent to converge.

Figure 8: A2C Results

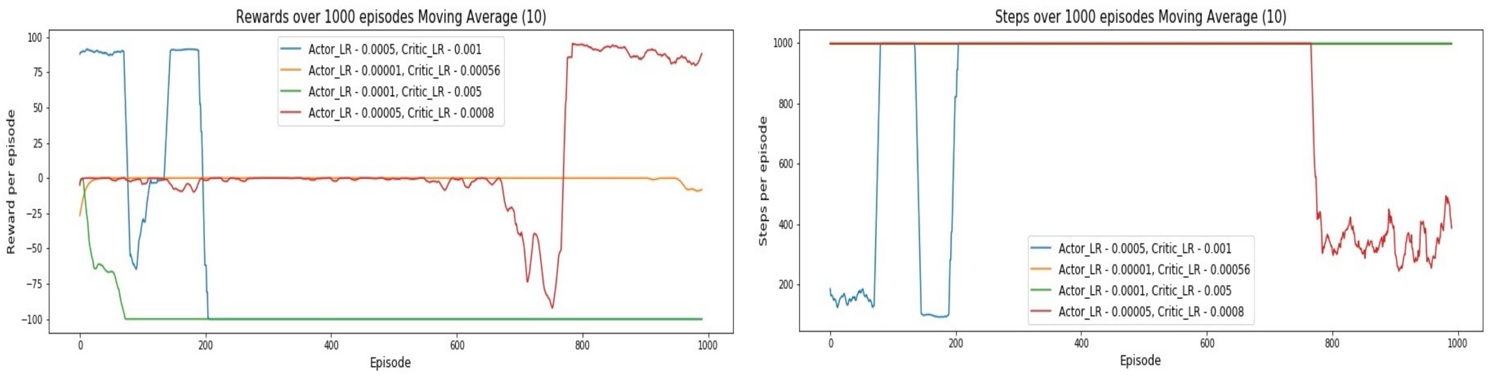


(a) (b)

The above graphs in figure 8 represent the performance (a) in terms of reward accumulated for each episode and time steps (b) to obtain those rewards for varying learning rate values in both the actor and critic networks. From the given performance plots we see that in each case the agent is able to reach the goal state early on and continues to improve its performance over the episodes. Although we use an Adam optimiser which varies the learning rate with every time step, we observe that using higher values for the initial learning rate exhibits more variant performance but also achieves higher rewards (mean >94) towards the later episodes. Looking at the steps taken to reach the goal we observe that generally the steps taken is inversely proportional to the reward obtained, signifying more steps taken leads to accumulating more negative reward. However it is interesting to see that during episodes 600 onwards the best performing model (with Actor_LR = 0.0001 & Critic_LR = 0.005) takes more steps than the other models but contrarily achieves a higher mean reward > 94. Given that our reward structure is based on accumulating negative rewards for more energy spent, it seems in this case at the expense of taking more steps by moving up and down the hills the agent learns to expend less energy towards reaching its goal state. This allows the agent to accumulate a higher reward.

Non-linear approximators allow us to scale the environment and accommodate continuous state spaces and action spaces using a parameterised probability distribution to represent policy in the A2C, however we're still restricted to a discrete action space in the DQN as we rely on the $max_a Q(s_{t+1}, a)$ for the Q-value approximation. Despite this restriction we see that the use of deep neural networks achieves higher reward in the given environment and potentially provides better overall performance as compared to Q-learning. A similarity with Q-learning is we find the average performance over episodes for the DQN and the A2C tend to exhibit variance. This is understandable given the delayed reward structure of the environment which we rely on to update the Q-values aiming for an optimal policy in Q-learning, as well as for function approximation in both the A2C and DQN.

Figure 9: A2C Results from unsuccessful runs



(a) (b)

Looking at figure 9, we observe the consequences of a delayed reward structure for the A2C where in the initial episodes the agent might get stuck in sub-optimal minima(Actor_LR - 0.0001) on account of not witnessing the goal state reward. In some cases the delayed reward leads the agent to achieve objective function maximisation by accumulating lesser negative reward and does so by updating its parameters to expend less energy by not taking any action (action 0; case of Actor_LR's - 0.00001 0.00005). Hence we rely on the agents exploration to reach goal state which sets the trend for later gradient updates towards objective maximisation. We also see the consequence of using a high initial Learning rate (Actor_LR - 0.005, Critic_LR - 0.001) which does not allow stable convergence.

8 Conclusion

In this paper, we used reinforcement learning to solve the Mountain Car problem, using three different algorithms. For each environment and algorithm we experimented with varying the parameters to see how these affected the performance of the agent. In order to use basic Q-learning we had to discretise the state space, and although the agent was able to reach the goal state this way, we hypothesised that this step limits the performance of the agent as this prevents the agent from exploring the full state space. We used DQN to overcome this and indeed the performance improved by some 50%, at the cost of greater computation time. We then looked at the environment represented by a continuous state-action space and used A2C in this case. The goal here was the same but the reward structure was altered. Here we observed the possible consequences of learning with a delayed reward in the Mountain car environment. This highlights in general the important role of the reward structure in the agent's learning process.

In the future we could explore how altering the network architecture affects the agent's performance when using DRL methods. We could also try incorporating prior knowledge to augment the agent's learning through reward shaping [5] or action biasing.

Appendix - References

[1] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016). URL: https://gym.openai.com/.

[2] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning*. The MIT Press, Nov. 13, 2018. 552 pp. ISBN: 0262039249. URL: https://www.ebook.de/de/product/32966850/richard_s_sutton_andrew_g_barto_reinforcement_learning.html.

[3] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[4] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[5] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–287. ISBN: 1558606122.

[6] Vincent François-Lavet et al. “An Introduction to Deep Reinforcement Learning”. In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. ISSN: 1935-8245. DOI: 10.1561/2200000071. URL: http://dx.doi.org/10.1561/2200000071.

[7] Vijay R Konda and John N Tsitsiklis. “Actor-critic algorithms”. In: *Advances in neural information processing systems*. 2000, pp. 1008–1014.

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]