Assignment 1: Tabular Reinforcement Learning
Course: Reinforcement Learning 2023

Peter Breslin s3228282

1. Introduction

Reinforcement learning (RL) is a branch of machine learning which involves interacting with an environment to learn the best actions to take to achieve a goal. The entity that directly interacts with the environment is known as the agent, while the environment itself is a simulation of a physical system used to define the task that the agent must learn. In order for the agent to learn how to better reach its target, it is given feedback. In general, this feedback either rewards the agent when desirable actions are taken or punishes it for choosing actions that do not help it achieve the goal. The feedback is received after an action is made in a given state, where the state refers to the current condition of the environment. When an action is completed, the state of the environment is updated with new information in light of that action, and the agent then receives the reward or penalty. Through this procedure of trial and error, the agent is gradually able to better interpret its environment and learn the strategy that is most optimal for achieving the goal.

RL algorithms take on many different forms and the choice of which algorithm to use is often best defined by the problem at hand. In this assignment, we focus on a well known form called tabular, value-based RL. A number of experiments were conducted in which different algorithms were applied to explore and better understand a range of basic principles related to tabular RL. The Python programming language was used for the implementation of this work.

2. Methodology

2.1. Tabular Reinforcement Learning

In tabular RL, the agent maintains a table containing estimates of the value of each possible action in each possible state. The table is updated with new estimates as the agent interacts with the environment and receives feedback. Values are updated in order to learn the optimal policy, i.e. the actions or *policy* that maximize the expected cumulative future reward. In this method, the state space and action space are discretized into a finite number of states and actions, stored in a matrix. This discretization simplifies the learning process and reduces the computational complexity of the problem at hand. The value function and policy are represented as tables with one entry for each state-action pair. The value function table stores the expected cumulative reward of following a particular policy from a given state, while the policy table stores the probability of selecting each action in each state. The way in which the agent updates these tables is of primary investigation in this assignment.

2.2. Environment

The agent interacts with the environment to learn and improve its behaviour until convergence is reached and the optimal policy is learned. The agent interacts with the environment by observing the current state, selecting an action, receiving feedback, and transitioning to the next state according to the transition conditions. In this assignment, an adapted form of the Stochastic Windy Gridworld environment is used. This consists of a 10×7 grid of cells, where each cell represents a state. The agent can take one of four possible actions at each state: move up, down, left, or right. The goal of the agent is to reach a target destination, defined by the cell at location (7,3). The agent is penalised by -1 for each step it takes, whereas it is awarded +40 if it reaches the goal. A special feature of this environment is that there is a randomly generated vertical wind that affects the movement of the agent; in certain columns of the grid, the agent is pushed either one or two additional steps up. The random nature of this winds ensures it is present on 80% of the occasions, thereby making the environment stochastic.

When the goal is reached, the episode, i.e. the sequence of interactions between the agent and environment, is terminated. The state in which this happens is known as the terminal state and when this is reached, the environment is reset to its initial state before the process begins again. Episodes provide a way to structure the interactions between the agent and environment, helping the agent learn an effective policy over time. By maximizing its cumulative reward over multiple episodes, the agent can improve its performance to achieve the goal. It is important to note that the agent can use its previous experience to inform its actions during the next episode; this is why estimates of the state and action values are updated based on the rewards the agent received during the previous episode. A number of different updating procedures are examined in further sections.

2.3. Dynamic Programming

We first focus on dynamic programming (DP) which, in the context of RL, refers to the family of algorithms used to solve Markov Decision Processes (MDPs). An MDP is an extension of a Markov chain, a stochastic process consisting of a sequence of states for which the probability of transitioning depends on the current state only. MDPs are used for modeling decision making in situations that depend on both the actions taken and the stochastic nature of the environment, hence why RL is typically modeled as an MDP. With MDPs, the state of the environment provides all the information needed to make a decision about what action to take next.

DP algorithms for MDPs work by iteratively improving an estimate of the optimal value function or policy. The value function represents the expected cumulative reward that an agent can achieve by following a specific policy in a given state, while the policy specifies the agent's actions in each state. It is important to note that DP assumes full access to a model of the environment, meaning that the agent has complete knowledge of the transition probabilities $p(s'|s,a)$ and rewards $r(s,a,s')$ associated with each state (s) and action (a) pair, where s' indicates the *next* state.

2.3.1. Value Iteration

Value iteration is a well known example of a DP algorithm used to compute the optimal value function for an MDP. This involves iteratively estimating the value function of each state based on the expected reward of the next state. This includes a discount factor, a value between 0 and 1 that determines the importance of future rewards and allows the agent to take long-term consequences of its actions into account so as to prioritize actions that lead to greater long-term rewards. The Bellman equations relate the value of a state or state-action pair to the values of the possible states to follow. For value iteration, the value function is estimated using the Bellman equation for the discounted future reward:

$$V^{\pi}(s) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s') \qquad (1)$$

where $V^{\pi}(s)$ is the value of state s (i.e. the expected cumulative reward of a state for following the policy from s), $r(s,a)$ is the reward from state s, γ is the discount factor, $p(s'|s,a)$ is the probability of transitioning to state s' when taking action a in state s, and the summation is over all possible next states s'. This formulation assumes a discrete policy $(\pi(s) \rightarrow a)$.

2.3.2. Q-value Iteration

The Q-value iteration algorithm is a form of value iteration that tends to converge faster. The Q-value represents the expected total reward obtained by taking action a in state s, and then following the optimal policy for the remaining steps. This can be thought of as a measure of how good it is to take a particular action in a particular state, with respect to achieving the goal of maximizing the total reward received. In Q-value iteration, the agent iteratively updates its estimate of the optimal Q-values for each state-action pair. This is based on the Bellman equation, which relates the Q-value of a state-action pair to the Q-values of the possible states to follow:

$$Q(s,a) = \sum_{s'} \left[p(s'|s,a) \cdot \left(r(s,a,s') + \gamma \cdot \max_{s'} Q(s',a') \right) \right] \qquad (2)$$

where $\max_{s'} Q(s',a')$ is the maximum Q-value over all possible next state-action pairs. The method of selecting the action with the maximum Q-value for a given state is known as a greedy policy, an approach used to exploit the current knowledge of the Q-values to maximize the immediate reward. The algorithm continues to update the Q-values until convergence to the optimal values is achieved. The optimal policy can then be found by selecting the action with the highest Q-value for each state. To study DP, the Q-value iteration algorithm was implemented and different parameters were tested, the details of which are outlined in Section 3.

2.4. Model-free Reinforcement Learning

While DP algorithms have access to a model of the environment, problems in which the model is not accessible can be encountered often. In this case, the transition probabilities are known by the environment only. These model-free algorithms require the agent to learn directly from experience, meaning that it is the interaction between the agent and environment which determines the value function or policy (the transition function and reward function are unknown). In practice, the agent receives a state and a reward from interacting with the environment, and then takes an action based on the current policy. Feedback from the environment is then given in the form of a new state and reward, and the process repeats itself n steps until convergence is reached (provided n is sufficiently large).

2.4.1. Exploration

In model-based algorithms like DP, you are able to sweep through all state-action pairs, guaranteeing all states under a greedy policy to be visited and thereby providing the best actions to take. However, you cannot sweep through the entire state space in model-free algorithms, meaning that it is possible to miss states under a greedy policy. This means that the agent may choose an action that appears to be the best but, in reality, might not be so and a better action leading to a higher reward may exist elsewhere. In other words, the agent is at risk of getting stuck in a local mini-

mum while the globally optimal action could be somewhere completely different. To account for this, an exploration parameter can be introduced into the action selection to encourage the agent to randomly explore more actions or visit new states. This parameter, which is typically called ϵ, is a value between 0 and 1 that determines the probability of exploration, with higher values leading to more exploration. It is important to maintain a random chance of exploration as excessive exploration can lead to a large amount of time being spent in sub-optimal states or taking sub-optimal actions. Including this value ϵ into our action selection gives the ϵ-greedy policy:

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon \cdot \frac{|\mathcal{A}|-1}{|\mathcal{A}|}, & \text{if } a = \text{argmax}_{b \in \mathcal{A}} \hat{Q}(s, b) \\ \epsilon/(|\mathcal{A}|), & \text{otherwise} \end{cases} \tag{3}$$

where $\mathcal{A}$ represents the set of all possible actions that can be taken by an agent in a given state. With this strategy, the agent will select a random action with probability ϵ to encourage exploration. The policy subtracts ϵ multiplied by the fraction of non-optimal actions $|\mathcal{A}| - 1$ over the total number of actions $|\mathcal{A}|$ so as to ensure that the policy does not get stuck in a sub-optimal solution or become too deterministic. Otherwise, if a is not the action with the maximum estimated reward, then the policy selects the greedy action with a small probability of $\epsilon/|\mathcal{A}|$. This ensures that all actions have a non-zero probability of being selected, which is important for finding better actions. This implies that selecting the greedy action will have a probability greater than $1 - \epsilon$. Recall that, in the context of Q-learning, the greedy action is simply the maximum Q-value (i.e. the action with the maximum estimated reward).

The Boltzmann policy is another exploration approach for action selection. Actions with a higher current value estimate are given a greater probability, while ensuring actions other than the greedy one are explored. The choice of action is given by the softmax function which assigns a probability to each action based on its Q-value relative to the other actions:

$$\pi(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\Sigma_{b \in \mathcal{A}} e^{\hat{Q}(s,b)/\tau}} \tag{4}$$

where $\tau \in (0, \infty)$ denotes the temperature parameter which controls the degree of exploration, the summation is over all possible actions b in the set $\mathcal{A}$, and all other symbols have their usual meaning. It can be noted that when τ is large, the exponential term becomes small for all actions, meaning that all actions are given a similar probability. This causes the policy to become more uniform/random since the agent is encouraged to explore more uniformly among all possible actions due to there being a higher chance of selecting sub-optimal actions. On the other hand, when τ is small, the exponential term becomes much larger for actions with higher Q-values, and much smaller for actions with lower Q-values. This results in the greedy action being given a much higher probability compared to the other actions, meaning that it is exploitation that is now encouraged.

To better balance exploration and exploitation, the technique of annealing can be used. This refers to the process of gradually reducing the magnitude of the exploration rate over time. When this reduction is done so linearly, the process is called linear annealing. The idea behind this is to initially encourage the agent to explore when little knowledge is known about the environment. As the agent explores it will learn more about the rewards associated with each action. By progressively decreasing the exploration rate as the agent's knowledge about the environment grows, we now encourage the agent to rely more on the learned policy instead of exploring any further. This reduces the risk of over-exploring and making sub-optimal decisions.

2.4.2. Update Rule

After an action is selected, an update rule must be applied to the algorithm. The choice of action will change the environment and the agent will receive a reward and the next state. Hence, after a timestep t the agent will have observed $\langle s_t, a_t, r_t, s_{t+1} \rangle$. The update rule updates the estimates of the values (i.e. Q-values if in the context of Q-learning) of the state-action pairs based on this data (specifically on the observed reward and next state). This helps to improve the agent's policy by gradually adjusting its value estimates for different state-action pairs. The 1-step Q-learning update is a basic approach for the update rule. This begins by first computing the new estimate for the back-up G_t which approximates the total expected return:

$$G_t = r_t + \gamma \cdot \max_{a'} \hat{Q}(s_{t+1}, a') \tag{5}$$

We see that this is composed of the immediate reward received after taking an action, and the discounted estimate of the future expected rewards represented by the product of the discount factor γ and the maximum Q-value of all possible actions a' in the next state s_{t+1}. It is also referred to as the "target" value because it represents the desired value that the agent is trying to estimate and optimize. The maximum Q-value of the next state is estimated using the current Q-value function $\hat{Q}(s_{t+1}, a')$, which gives the expected return if the agent takes action a' in state s_{t+1}. The update rule then takes on the form:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot \left[G_t - \hat{Q}(s_t, a_t) \right] \tag{6}$$

where $\alpha \in (0, 1]$ is the learning rate, a hyper-parameter used to control the pace at which the algorithm learns. We see from this update rule that the difference is found between the target value G_t and the current estimate of the Q-value function for the state-action pair $\hat{Q}(s_t, a_t)$. This is known as the temporal difference error, and it represents the difference

between the actual reward received by the agent and the expected reward predicted by the current estimate of the Q-value function (i.e. it represents the error in the current estimate of the Q-value). By including this in the update and repeatedly updating, the agent can refine its estimate of the optimal Q-value function and hence improve its policy until convergence is reached.

2.5. On-policy versus off-policy target

Q-learning uses an off-policy approach to the back-up method. The term "off-policy" refers to the fact that the value function is learned based on a policy that is different from the policy that was used to obtain the data. In other words, the policy being evaluated differs from the policy used for learning.

However, there is also the on-policy approach which learns the value function and the policy based on the same policy that is used to obtain the data. The term "on-policy" refers to the fact that the value function is learned based on the current policy that is being followed. In other words, the policy being evaluated is the same as the policy used for learning. SARSA is a well-studied example of an on-policy method. The name stands for State-Action-Reward-State-Action, which refers to the sequence of steps taken by the algorithm. For each step, the agent observes the current state, takes an action based on the current policy, receives a reward, observes the next state, and takes another action based on the new state and policy. The algorithm repeats this process until the episode terminates. Hence, the back-up equation for SARSA, given observations $\langle s_t, a_t, r_t, s_{t+1} \rangle$, can be expressed as:

$$G_t = r_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1}) \tag{7}$$

while the update rule follows the standard tabular update as given in Eq. 6. The main difference between the Q-learning and SARSA back-up equations pertains to the value they bootstrap at the next state. Bootstrapping is a process for estimating the expected future rewards and values, based on the current estimates of the value function. By using the estimates of future rewards and values to improve the estimates of the current state, this allows the agent to update its estimates iteratively. In Q-learning, the optimal policy is learned by backing-up the value of the best possible action in the next state, while with SARSA, the value of the action taken, whether it be the best possible one or some explored action, is backed-up. Hence, SARSA works to learn the value function of the policy we actually execute.

2.6. Depth of Target

The depth of the target is a further consideration to be made when implementing the back-up algorithm. The depth refers to the number of steps taken into the future when computing the expected return for a given state. So far, we have explored the 1-step method, which directly bootstraps a value estimate after a single transition. Alternatively, one can sum up multiple awards in an episode before bootstrapping such that we look n steps into the future when computing the expected return. This type of approach where the depth of the target is some number greater than one is referred to as n-step methods. Since this only relies on how the algorithm bootstraps, the Q-learning and SARSA algorithms can be simply extended to so-called n-step Q-learning or n-step SARSA. In this assignment, we choose to focus on n-step Q-learning, which computes the following target:

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \max_a Q(s_{t+n}, a) \tag{8}$$

where the first term $\sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i}$ is the sum of the discounted rewards obtained over the number of timesteps for which the agent accumulates rewards to compute the target Q-value estimate (this is also known as the n-step horizon), and all other symbols take on their usual meanings. We can see that the target balances the immediate rewards and the estimated future rewards in a trade-off that depends on the value of n and γ. Put simply, this algorithm works to update the Q-value estimates using all rewards obtained over n timesteps. As with the 1-step back-up equations, the update rule follows the standard tabular update as given in Eq. 6.

Instead of looking n-steps ahead, we can also choose to omit bootstrapping altogether and simply sum up all rewards obtained over an episode. This gives a Monte Carlo update, which follows the form:

$$G_t = \sum_{i=0}^{\infty} (\gamma)^i \cdot r_{t+i} \tag{9}$$

where the infinite sum represents the discounted future rewards that are expected beyond the current timestep (with the agent following the current policy until the end of the episode). As seen by the equation, the Monte Carlo method does not bootstrap but instead estimates the value directly from the observed returns by acting over the entire episode, not just over a fixed number of timesteps.

3. Experimental Set-up

The Stochastic Windy Gridworld environment was used for each experiment conducted in this assignment, the code for which having been provided. Passing a Q-value table in as input to the environment gives a display of the Q-value estimates for each action in each state, including indications of the optimal policy. This gives us a visual representation for the operation of the algorithm and how it progresses with time. Please refer to the Python files included with this report for the code written throughout this assignment.

3.1. Dynamic Programming

In the first experiment, the Q-value iteration algorithm as given in Eq. 2 was fully completed in the `DynamicProgramming.py` file. This involved implementing both the greedy policy for the action selection procedure and the update rule, and then iterating the update for all state-action pairs. For this experiment, the discount factor was set to $\gamma = 1.0$ (this was the case for all subsequent experiments) and a threshold value of 0.001 was used. The threshold is a small positive value that defines the convergence criterion. Our implementation of the algorithm is written such that it will terminate when the maximum change in the Q-values is less than this threshold. This maximum change is simply the error Δ as given by the maximum absolute difference between the updated Q-values and the previous ones for each state-action pair:

$$\Delta = \max(|Q(s_{t-1}, a_{t-1}) - Q(s_t, a_t)|) \tag{10}$$

This continues until Δ is less than the given threshold, indicating that the Q-values have converged to their optimal values.

3.2. Q-learning

Following DP, the effect of exploration with Q-learning was investigated in the `Q_learning.py` file. Instead of only following the greedy policy in the action selection, both the $\epsilon-$greedy (Eq. 3) and Boltzmann (Eq. 4) policies were implemented and each one was tested separately. The softmax function was given in the provided `Helper.py` file to define the Boltzmann policy. Following this, the 1-step Q-learning update rule was implemented using the back-up estimate as given in Eq. 5, and the tabular learning update as outlined in Eq. 6. The algorithm operates over a fixed number of timesteps where at each step, an action is selected, the environment is simulated, the update rule is applied, and the rewards are collected. This iterative process will break when the agent can no longer take any further actions i.e. when the terminal state is reached.

Experiments were run separately using the $\epsilon-$greedy and Boltzmann policies, for which values of $\epsilon = [0.02, 0.1, 0.3]$ and $\tau = [0.01, 0.1, 1.0]$ were used for the exploration and temperature parameters, respectively, and a learning rate of $\alpha = 0.1$ was taken. For each setting, 50000 timesteps were run and the results were averaged over 50 repetitions.

3.2.1. Annealing

Experiments were run to test annealing ϵ and τ during training using the linear annealing function given in `Helper.py`. In practice, annealing involves defining an exploration rate schedule, which specifies how the exploration rate will be decreased over time. A common schedule is to start with a high exploration rate and then decrease it linearly over a fixed number of steps until it reaches a minimum exploration rate. The function provided returns the current value of the exploration parameter being annealed based on the current timestep and the annealing schedule. The annealing schedule decreases the exploration value linearly from a starting value to a final value over a specified number of timesteps. Using the Q-value iteration algorithm, a further experiment was run to compare the ϵ-greedy and Boltzmann policies with and without annealing.

3.3. SARSA

To explore an on-policy back-up method, an experiment was conducted using SARSA in the `SARSA.py` file. The implementation of this is similar to that of Q-learning, except that the back-up algorithm now followed the target definition for SARSA given in 7. This meant that the algorithm for iterating the update had to be altered slightly to include select a policy to use during learning that is different to the one that is evaluated. To compare Q-learning to SARSA, experiments were run with both methods using learning rate values of $\alpha = [0.02, 0.1, 0.4]$. In the case of Q-learning, the $\epsilon-$greedy policy was used with $\epsilon = 0.1$. For each setting, 50000 timesteps were run and the results were averaged over 50 repetitions.

3.4. Target Depth

To investigate how changing the depth of the target affects the learning, the n-step Q-learning method was implemented in the `Nstep.py` file. This followed that of the 1-step Q-learning algorithm, except that the back-up expression in the update was changed to follow the form of Eq. 8. The sum of discounted rewards obtained over the number of timesteps could not be simply computed because the sum must be over all rewards for each timestep. Hence, the target algorithm was written such that it aggregates for each reward, meaning that the number of rewards left to sum (which we call m) needed to be tracked. It is possible that the algorithm cannot always go n-steps ahead (e.g. we cannot look two steps ahead at the last iteration). Hence, we must ensure that the n-step return is computed only for the remaining timesteps in the episode if it is not long enough to cover n timesteps. We resolve this by defining m to be the minimum between n and the remaining number of timesteps in the episode from the current timestep.

If the final state in the sequence is a terminal state, the algorithm should terminate before reaching the end of the episode. In this case, the n-step return is computed as the first term of Eq. 8, i.e. $G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i}$, which is just n-step without bootstrapping. This is because the future rewards beyond the last observed state are assumed to be zero, hence the second term of Eq. 8 is not included. However, the way in which we loop means that it is possible

that the last state in the sequence may not be a terminal state. If the loop were to continue beyond the last observed state and reach a non-terminal state, the n-step return would no longer be well-defined, as there would be no estimated value for the state-action pair beyond the end of the episode. To prevent this, we add a further condition to the termination criteria that checks if $t + m = T_{ep}$, i.e. it checks if the end of the episode is reached. This ensures that the loop will be terminated at the end of the episode even if the last state is not terminal. Finally, the update rule then follows the standard tabular form as given in Eq. 6.

Following this, the Monte Carlo method was used in the `MonteCarlo.py` file to explore the other extreme, when bootstrapping is omitted altogether. Once again, the way in which the target was defined had to be changed to instead follow Eq. 9. While the standard tabular form was used for the update procedure, the updating was looped in the reverse order (i.e. starting from the end of the episode and moving backwards in time towards the beginning). This is because the Monte Carlo update requires the total discounted reward from each timestep until the end of the episode, hence the Q-values need to be updated in reverse order.

Finally, both the n-step Q-learning and Monte Carlo methods were compared to each other. For the n-step Q-learning, four different back-up depths were tested (1, 3, 10, and 30). A learning rate of $\alpha = 0.25$ was taken, and the ϵ-greed policy was used with $\epsilon = 0.1$. For each setting, 50000 timesteps were run and the results were averaged over 50 repetitions.

4. Results

4.1. Dynamic Programming

The DP experiment ran successfully, with convergence being reached after the 18th iteration of the algorithm. When the max error is printed during execution, we observe the value slowly decreasing at the beginning before steeply decreasing to a value close to zero after a couple of iterations. This tells us that the agent is learning and that the learned knowledge about the environment builds up quickly towards convergence. The progression of the Q-value iteration during execution was tracked, the plots for which can be viewed in the supplementary material document submitted along with this report (as was instructed to do in the guidelines).

In each cell, the arrows indicate the greedy action for each state according to the Q-values, and the values plotted indicate the Q-value for each state-action pair in the environment. We see that the greedy action points towards the action with the maximum expected award. At convergence (Figure A3), the values in each cell near the goal are quite large because of this close proximity. The agent has the greatest level of knowledge about the environment at this stage, and this is reflected by the arrows and values given in each cell. At midway (Figure A2), the agent begins to get a better understanding of the environment and has a reasonable idea of the optimal actions to take to reach the goal. Finally, after just one iteration (Figure A1), we see that there is not a clear action to take and the agent has a very limited understanding of the environment simply due to the fact that it hasn't had a chance to properly learn yet.

Following this, the converged optimal value at the start was computed i.e. $V^*(s = 3)$ at the start state. This is simply the optimal value of being in state $s = 3$, which can be determined by the maximum Q-value for $s = 3$ after convergence is reached. This was found to be $V^*(s = 3) \approx 23.8$. This conveys the maximum expected cumulative reward that the agent can receive by following the optimal policy starting from state 3 based on the estimated Q-values. After convergence, the Q-learning algorithm has determined the optimal policy that maximizes the expected cumulative reward for each state (which includes state 3). In other words, it indicates the long-term value of being in state $s = 3$ under the optimal policy. Using this value, the magnitude of the final reward, and the magnitude of the reward on every other step, the average reward per timestep under the optimal policy was then computed as ≈ 1.4. However, a value of 1.3 was used for the remainder of the experiment, keeping in line with the assignment guidelines.

The goal state in the environment ($s = 52$) is terminal. However, convergence is still reached because there is a condition in `Environment.py` that makes the transition and rewards function a self-loop without rewards if the terminal state is reached. Another way to solve this issue could be to set a flag in the environment indicating that the goal state has been reached, and then block any actions from being taken in the goal state. This would ensure that the agent does not revisit the goal state repeatedly and prevent the environment from getting stuck. A further way could be to remove the goal state altogether and terminate the episode when the agent reaches the goal location (i.e. without making it a state).

Finally, the goal state was changed from location [7,3] to [6,2] and the experiment was run again. In this set-up, the algorithm took longer to converge. This is simply because the new goal state is farther away from the starting state compared to the previous goal state of [7,3]. When the goal state is farther away, the agent needs to take more steps to reach it, which means that the Q-values for many state-action pairs need to be updated more times before convergence is achieved. In addition, since the wind in the environment is stochastic, there is a greater chance that the agent will be pushed away from the goal state and need to take even more steps to reach it. Therefore, it takes longer for the algorithm to converge to the optimal policy when the

goal state is farther away.

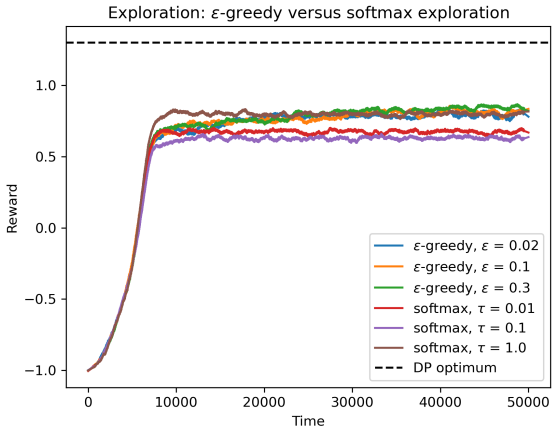

Figure 1. The results of the Q-learning iteration algorithm averaged over 50 repetitions for both the ϵ-greedy and Boltzmann policies with varying exploration parameter values.

4.2. Exploration

The ϵ-greedy and Boltzmann policies were compared to each other with different settings for the exploration parameters. Figure 1 shows the smoothed learning curves obtained after running each setting for 50000 timesteps and averaging the results over 50 repetitions. The learning curves show how the agent's performance improves over time as it gains more data about the environment. Specifically, the learning curve shows the average reward obtained by the agent over a certain number of episodes as a function of the number of episodes completed. Averaging the results over multiple repetitions allows us to obtain a more robust estimate of the agent's performance than would be possible with a single run of the algorithm. Since the Stochastic Windy Gridworld environment involves random factors that can affect the agent's performance, averaging over multiple runs can help to account for this variability and provide a more accurate estimate of the agent's true performance.

We see that the performance of the runs with the ϵ-greedy policy are very similar to each other, despite different values of ϵ being used. This is likely due to the value of ϵ not being high enough in each case, and perhaps exploration is not being encouraged aggressively enough as a result. It can be noted that these runs outperform those using the Boltzmann policy, except for the run with $\tau = 1.0$. Annealing was also experimented with, the results of which can be seen in Figure A4 of the supplementary appendix. Different values for the percentage and the start and final values for the exploration parameters were tested until settling on a percentage of 0.5 and start / final values of $[\epsilon_{\text{start}} = 1.0, \epsilon_{\text{final}} = 0.02]$ and $[\tau_{\text{start}} = 1.0, \tau_{\text{final}} = 0.01]$. From Figure A4, there is not much change to be observed from the inclusion of annealing. This could be due to the fact that the results are averaged over 50 repetitions, since a more appreciable improvement was noticed when comparisons were made for single runs. However, the curve for the ϵ-greedy policy did perform slightly better when annealed, and that for the Boltzmann policy did initially learn more quickly compared to the curve that wasn't annealed.

4.3. On-policy versus off-policy target

The Q-learning and SARSA algorithms were tested for learning rates of $\alpha = [0.02, 0.1, 0.4]$ where each setting was run for 50000 timesteps and the results being averaged over 50 repetitions, the learning curves for which are shown in Figure 2. For both Q-learning and SARSA a value of $\alpha = 0.02$ is too low and results in the learning process becoming very slow, making it difficult for the agent to converge to an optimal policy. We see that, for Q-learning, the performance of the algorithm gets better as α increases. However, the runs with $\alpha = 0.1$ reach similar results to those for $\alpha = 0.4$, meaning that it probably wouldn't be the best idea to increase α any further. While a higher learning rate can lead to faster convergence, it can also make the algorithm more susceptible to noise in the environment. This can be seen in the plot, where the Q-learning curves get progressively noisier as α increases. The difference in the learning curves is starker in the case of SARSA. Although the curve with $\alpha = 0.1$ initially learns quicker, the curve with the higher learning rate of $\alpha = 0.4$ reaches a better performance in the long run.

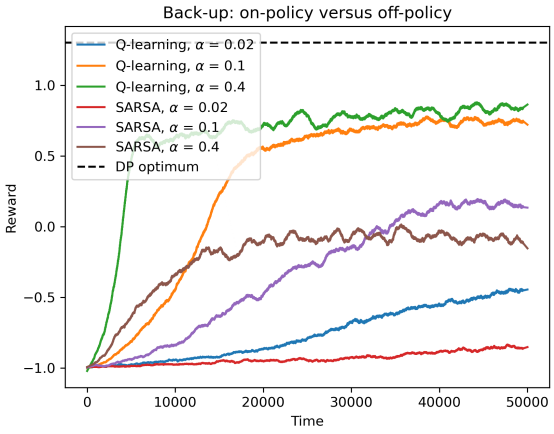

Figure 2. The learning curves for the SARSA (on-policy) and Q-learning (off-policy) algorithms for different values of the learning rate α.

Interestingly, Q-learning rather significantly outperforms SARSA. This is likely due to the environment being used. Q-learning is an off-policy algorithm, and since there is a large degree of randomness due to the windy conditions in the Stochastic Windy Gridworld environment, the agent

can be caused to deviate from the optimal policy during training. Q-learning may be better equipped to handle this randomness by learning from the maximum Q-value of the next state, regardless of which action is taken in that state. In the Stochastic Windy Gridworld environment, if the policy being followed is suboptimal due to the windy conditions, SARSA may have a harder time learning the optimal policy, because it is learning from the suboptimal policy it follows during training since it is an on-policy method. While Q-learning is preferred in our problem case, one could imagine SARSA being preferred in problems where the rewards are sparse and delayed, and where the agent needs to explore the environment to discover the best actions to take (e.g. problems where the agent needs to explore a very complex environment to find a target location).

4.4. Depth of Target

Unfortunately, I could not get my experiments for the n-step Q-learning and Monte Carlo methods to run correctly. I had thought they were correct since they were indeed converging when run separately, but no learning was happening whatsoever when it came to running the experiment file. They are no longer working in the individual files either now and I do not understand why, I followed the pseudo-code correctly to the best of my knowledge. Unfortunately, I didn't realize this issue until it cam time to include my plots in the report over the weekend that the assignment was due, hence I didn't reach out to the TA's for help. In future, I will ensure all aspects of my code is working before leaving for the weekend.

If this was done correctly, I would expect the n-step Q-learning algorithm to learn faster initially because it updates its Q-values after every n steps, allowing it to learn more quickly from new data. In contrast, the Monte Carlo method only updates the Q-value of a state-action pair once the agent has completed an entire episode, so it may take longer to learn from new data. Furthermore, the Monte Carlo method typically requires more episodes to converge to an optimal policy because it must wait for the end of each episode before updating its estimates. This can result in slower learning and worse final performance. Hence, I would also expect the Q-learning method to have the better final performance.

Additionally, I would expect the performance of the n-step Q-learning algorithm to be worse as n increases. Since the Stochastic Windy Gridworld is a relatively simple environment with a small state space, it is unlikely that the 30-step Q-learning algorithm would perform better than the 1-step or 3-step Q-learning algorithms. The 30-step Q-learning algorithm would be better suited to environments where episodes are long and the agent must take many steps to reach a terminal state. Furthermore, due to the stochastic nature of the environment, the 3-step Q-learning algorithm may find it more difficult to estimate accurate Q-values, since the rewards and transitions are less predictable over longer timestep sequences. In contrast, the 1-step Q-learning algorithm only relies on immediate rewards and transitions, which may be more predictable in a stochastic environment. Hence, I would expect the 1-step Q-learning algorithm to perform best.

5. Discussion & Conclusions

In this assignment, a range of experiments were conducted to explore the basic principles of tabular RL. Firstly, DP was implemented allowing for comparisons between it and RL to now be drawn. A strength of DP compared to RL is that DP algorithms typically converge to the optimal policy using a relatively small number of iterations. Since RL algorithms must explore the environment, this can be a slow and inefficient process. However, a potential weakness to DP algorithms is that they are model-based, meaning that they require full knowledge of the environment. Model-free RL algorithms are more suitable for real-world applications where a model of the environment may not be available or may be difficult to obtain.

As was previously outlined, the off-policy method of n-step Q-learning dealt with the stochastic nature of the environment better than the on-policy SARSA method. n-step Q-learning is indeed an off-policy method since it updates the Q-values based on the maximum Q-value of the next state-action pair, meaning it can learn the optimal policy regardless of the policy being followed by the agent. Based on our experiments, we can confidently say that tabular RL algorithms perform well problems with stochastic dynamics. These algorithms are also relatively simple and easy to understand, making them a good starting point to learning RL. However, tabular RL algorithms can run into trouble when dealing with high-dimensional state or action spaces. This is known as the curse of dimensionality, which refers to the exponential increase in the number of states or actions as the dimensionality of the state or action space increases. To overcome this, ML methods like deep learning can be used to approximate the value function or policy by automatically learning to extract relevant features from high-dimensional inputs and generalizing well to unseen states.

In final conclusion, the Q-learning algorithm with the ϵ-greedy policy was the most consistently best performing method explored. Hyperparameters such as the learning rate and exploration parameters were varied, the results of which helping to better define their affects. Furthermore, knowledge of on- and off-policy methods and how they relate to the back-up depth and the concept of bootstrapping was gained, leading to an overall more complete understanding of tabular RL methods.