AI in Games, *Reinforcement Learning*
Assignment 2, Report

Malo Hamon (230804946)
Nimesh Bansal (230404399)
Pranav Narendra Gopalkrishna (231052045)

January 7, 2024

Definitions

Policy and deterministic policy

A policy is a function that maps each state-action pair to the probability of taking the given action from the given state. In a deterministic policy, for any given state, only one action is mapped the probability of 1, with all others being mapped to 0. This effectively makes a deterministic policy a mapping from each state to the action to be taken from the state, since in a deterministic policy, the agent is certain to take only a particular action from a particular state.

Discounted return

Given that an agent takes a series of steps after time stamp t with the observed rewards $r_{t+1}, r_{t+2}, r_{t+3}...$, the discounted return after time stamp t is given by:

$$u_t = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}$$

State-value function

A state value function *defined with respect to a policy* π maps each state to the expected discounted return of starting from the given state and following the given policy π.

Action-value function

An action value function *defined with respect to a policy* π maps each state-action pair to the expected discounted return of first taking the given action from the given state, then following the given policy π.

Question 1

To obtain the number of iterations taken to return an optimal policy, both policy iteration and value iteration were run with no maximum iteration cutoff, and a threshold of 0 when comparing the state-value delta, which causes them to converge completely, returning optimal policies. The results were as follows:

Method	Iterations
Policy iteration	133
Value iteration	26

Table 1: Iterations to converge to an optimal policy

Policy iteration vs. value iteration

Policy iteration performs alternating steps of policy evaluation and policy improvement. Policy evaluation obtains the state-values for a given policy, and policy improvement obtains the action-values using the state-values and chooses the actions that maximise the action-values for each given state.

On the other hand, value iteration estimates the state-value function itself to approach the optimal state values. It does this by obtaining the action-values for each state-action pair, then picking the maximum of these as the state-value for the given state. Practically, this achieves the same goal as policy improvement, except that instead of applying policy improvement after every policy evaluation, we first obtain the optimal state values then run policy improvement. This leads to far fewer policy evaluation steps, since unlike policy iteration, value iteration evaluates only one set of state-values and does not evaluate a series of slightly better policies. After obtaining the optimal state-values, we apply policy improvement once to obtain the optimal policy from the optimal state values.

Question 2

NOTE: *Discounted return has been defined in the "Definitions" section*

An episode is a set of iterations for which the agent interacts with the environment until it reaches an absorbing state (which could be due to reaching the goal, a whole or simply reaching the maximum number of steps allowed to the agent by the environment). In the course of model-free algorithms, the agent interacts with the environment over multiple episodes. If the agent has taken a series of steps at time stamps $0, 1, 2...$ with the respective observed rewards $r_1, r_2, r_3...$ for a given episode n, then the discounted return for episode n is given by:

$$E_n = \sum_{k=1}^{\infty} \gamma^{k-1} r_k$$

In practice, the number of steps taken as well as the summation are not infinite since the absorbing state is reached after a certain time stamp. Note that even if we take further steps from the absorbing state, the summation will not be affected since the observed rewards will be zero.

To obtain the following plots, we have taken the moving averages of per-episode discounted returns, with the window size as 20. Moving averages (or rolling averages) with window size N of a sequence of per-episode discounted returns $E_1, E_2...E_k$ are simply the following sequence of averages:

$$\frac{1}{N}\sum_{i=1}^{N} E_i, \frac{1}{N}\sum_{i=2}^{N+1} E_i, \frac{1}{N}\sum_{i=3}^{N+2} E_i ... \frac{1}{N}\sum_{i=k-N}^{k} E_i$$

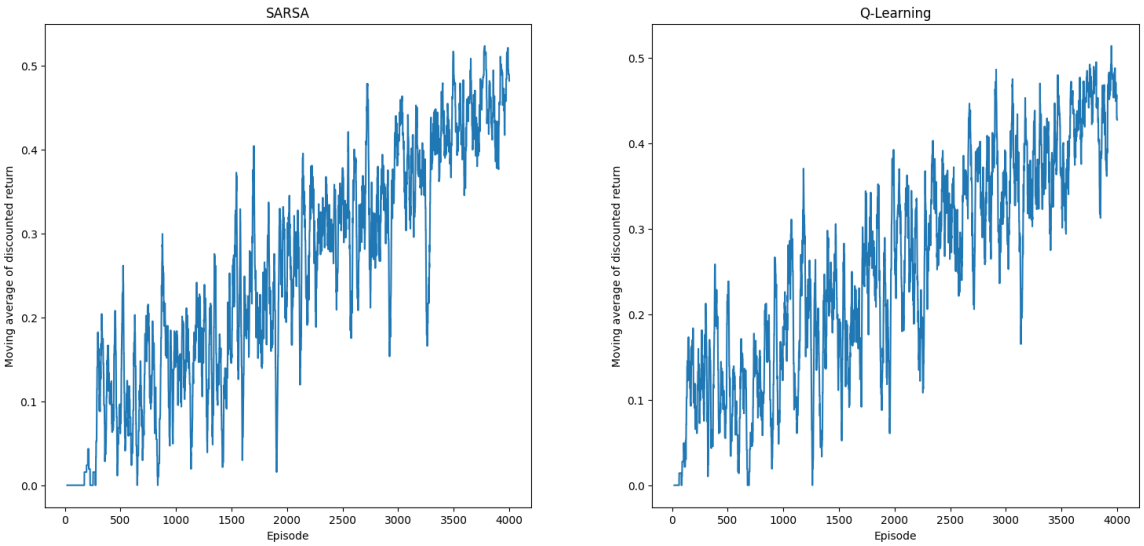

Figure 1: Moving averages for tabular model-free methods

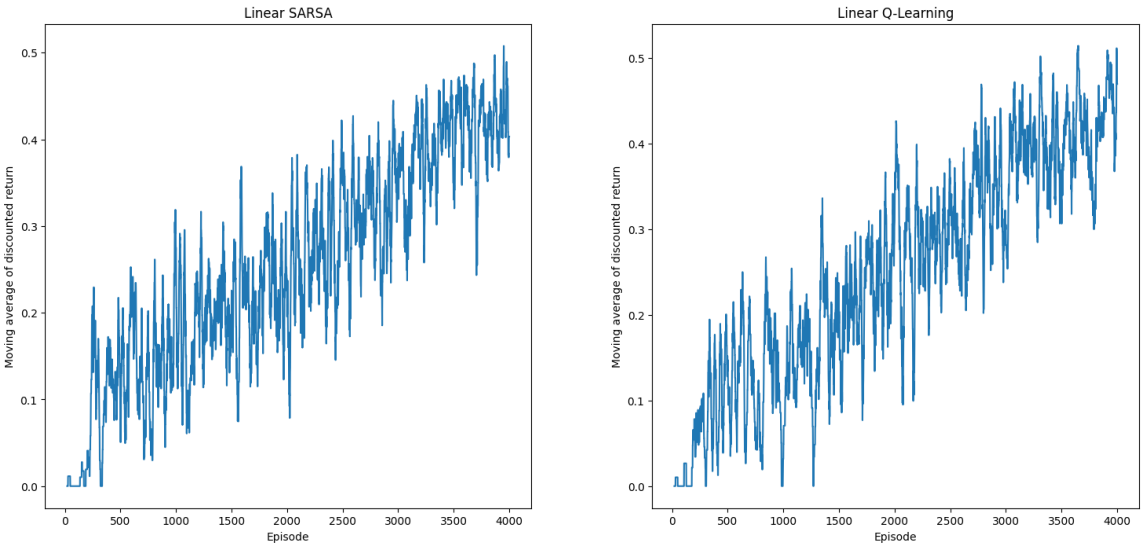

Figure 2: Moving averages for non-tabular model-free (linear function approximation) methods

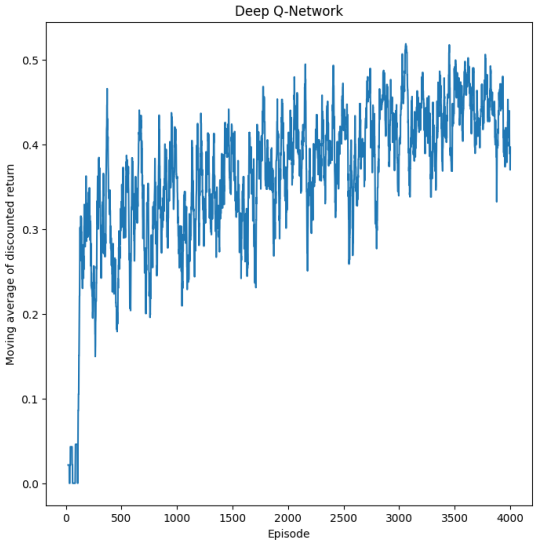

Figure 3: Moving averages for deep Q-network learning

NOTE: *In the following discussion, "returns" refers to "per-episode discounted returns".*

From the above graphs, we observe that the returns for SARSA and Q-learning (*both the tabular and the linear function approximation variations*) increase at a similar rate, with similar levels of fluctuations. We also observe that Deep Q-network learning (DQN) shows sparse returns up to 100 iterations, after which returns shoot higher than they do for the other methods at the same number of episodes. Furthermore, DQN shows greater returns on average, but reaches a similar level of returns as the other methods after 4000 iterations due to a slower rate of increase in returns across episodes.

In each method, the steps are taken by the agent in the environment based on the *epsilon*-greedy policy, wherein ϵ (epsilon) is the exploration factor. Furthermore, in each method, the exploration factor is made to decrease linearly with the number of episodes. Hence, we can see that the discounted return for an episode tends to increase on average across episodes due to two factors. Firstly, the exploration factor eventually decreases enough to make the agent more likely to take the most promising action rather than a random action. Secondly, the action-values have been updated enough to more accurately represent the expected (i.e. future) rewards of taking actions from each state. These factors work hand-in-hand; a higher exploration factor initially (when the environment is relatively unknown) allows the agent to gather enough experience to update its action-values to a higher accuracy, and a lower exploration factor as the action-values become more accurate allows the agent to pursue actions within the episode that are more rewarding in the long-term. These two factors together contribute to increasing returns (i.e. discounted returns) across episodes.

Question 3

Approach

To minimise the number of episodes required to find an optimal policy, a grid search can be performed with different combinations of the learning rate and exploration factor. To determine when the algorithm has converged on an optimal policy, the return (sum of discounted rewards) of each episode can be compared to the return of an optimal policy. Due to the noisy nature of returns obtained during each episode, the rolling average of the returns is compared to the optimal return instead of the actual return of each episode, to ensure that the policy found is actually optimal. However due to the ϵ-greedy policy, it is unlikely that the rolling average can actually reach the optimal return, so the target optimal return is multiplied by a coefficient (0.95).

To reach the optimal returns, we need to obtain the discounted return of one episode following the optimal policy. This optimal policy was obtained using a tabular model-based method (policy iteration, in this case).

Algorithm 1 Optimise learning rate and exploration factor

Input env, LR, EF, algorithm

```
policy* ← policy_iteration(env)
return* ← get_episode_return(env, policy*)
best ← ∞
best_params ← null
for lr ∈ LR do
    for ef ∈ EF do
        episode_returns ← algorithm(env, lr, ef)
        rolling_average ← rolling_average(episode_returns)
        i ← rolling_average ≥ return * ×0.95
        if i ≤ best then
            best ← i
            best_params ← lr, ef
        end if
    end for
end for
```

Search space

The following values were included in the grid search:
Learning rate: 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2, 0.1
Exploration factor: 1, 0.8, 0.6, 0.4, 0.2, 0.1, 0.01

Results for Small lake

The exploration factor had the largest impact on rate of convergence, in fact the smallest exploration factor reached an optimal policy much faster than the larger ones. This is likely due to larger exploration factors causing the algorithm to waste time taking random actions and exploring the search space. Because the environment is so simple, there seems to be little need for exploration. A follow up experiment was run with an exploration factor of 0 which did not converge at all, indicating that some exploration is needed, but not much.

The learning rate had a smaller effect. However, a learning rate between 0.6 and 0.8 seems to be optimal.

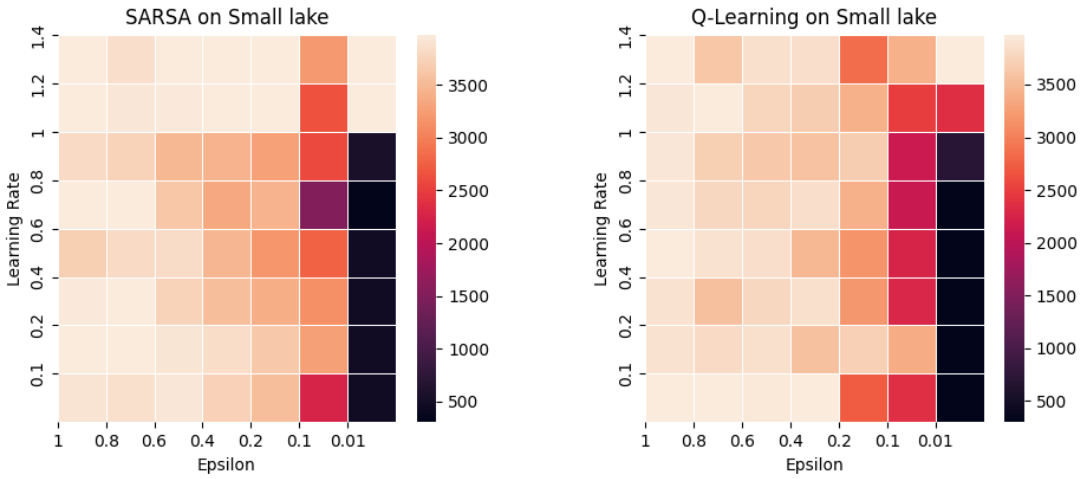

Figure 4: Number of episodes to reach optimal return

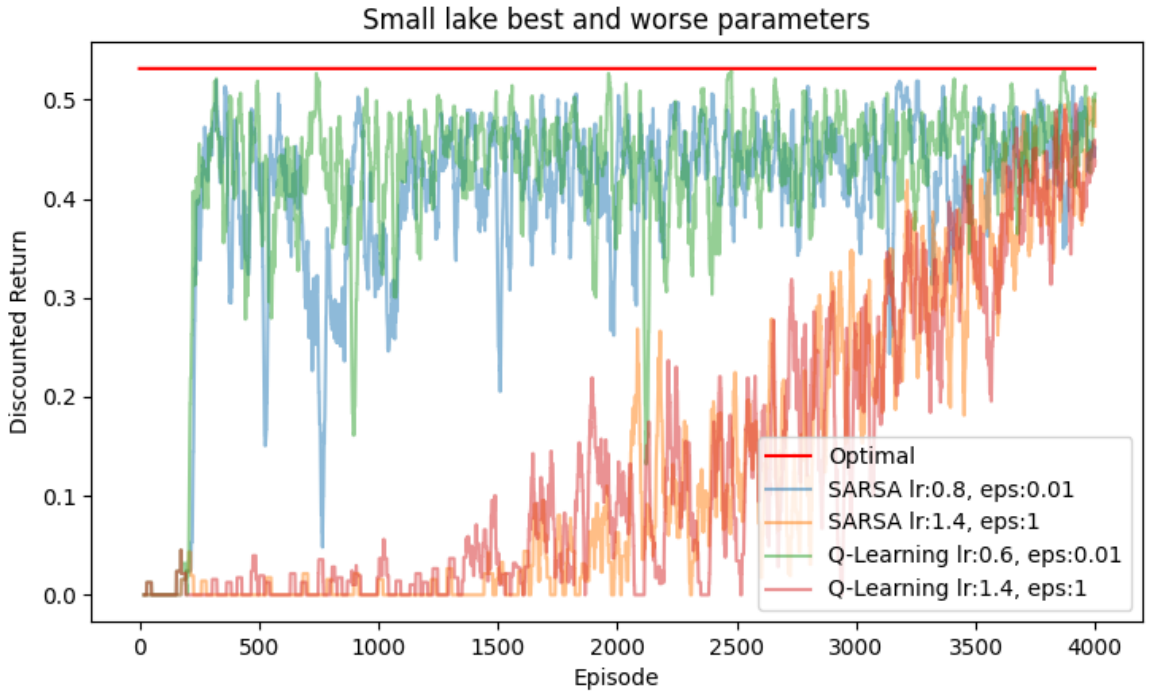

Figure 5: Rolling average return for best and worse parameters

Algorithm	Learning rate	Exploration factor
SARSA	0.8	0.01
Q-Learning	0.6	0.01

Table 2: Optimal parameters for Small lake

Results for Big lake

Unlike Small lake, Big lake has much more open space for the agent to move, which means the possible paths an agent may take within an episode are far greater in number. This means that although the agent can reach the goal in 16 steps, it is both viable and desirable to allow a much larger number of steps for the agent to interact with the environment per episode. If this number is too small, the agent to very rarely (if ever) achieves a positive reward, and thus, the action-values never converge to the

optimum. For this reason, the maximum steps per episode was increased to 100 for the search on Big lake.

With this change, the same result can be observed for Big lake, where a very small value for the exploration factor seems to reach an optimal return faster, and a learning rate of 0.8 is optimal. When comparing the rolling discounted return plots of the best parameters combinations for both Small lake and Big lake, it can be seen that Big lake takes longer to reach the optimal return (around episode 1000 instead of episode 300 and this is with big lake having a maximum of 100 steps in the environment instead of 16), as it is more complex and requires more exploration.

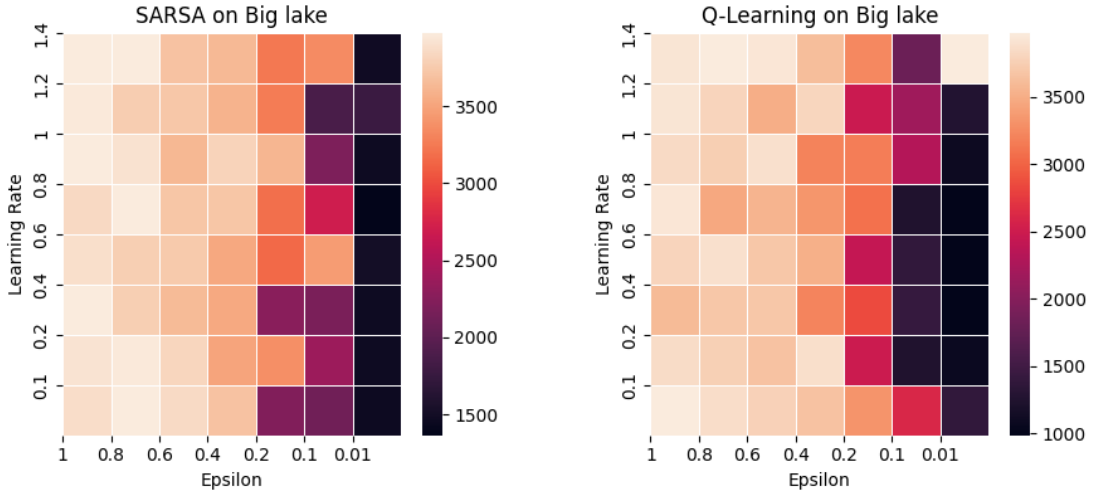

Figure 6: Number of episodes to reach optimal return

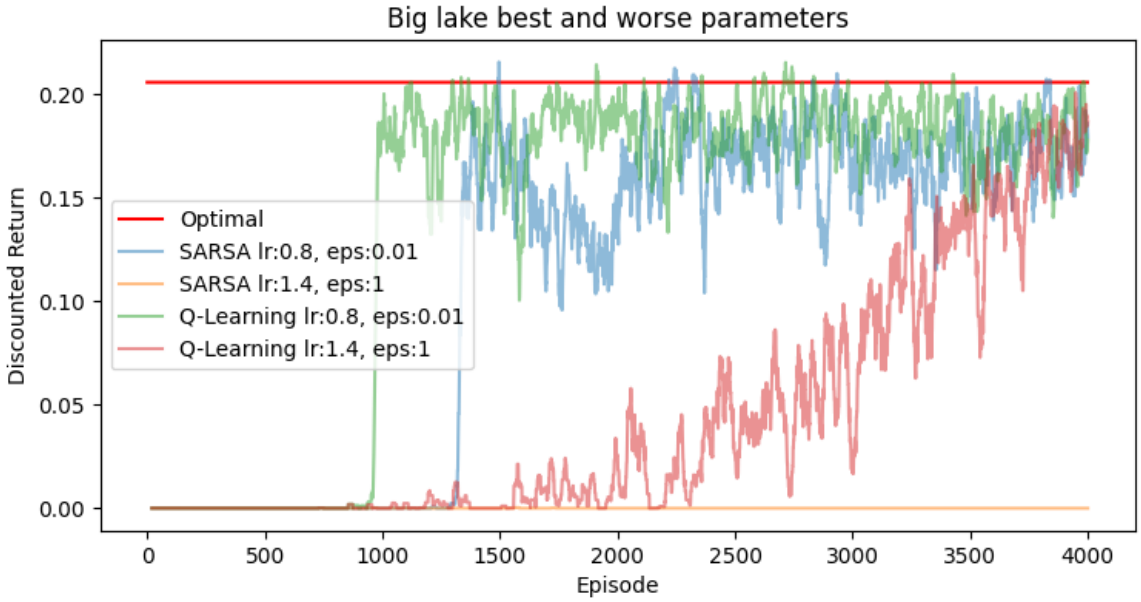

Figure 7: Rolling average return for best and worse parameters

Algorithm	Learning rate	Exploration factor
SARSA	0.8	0.01
Q-Learning	0.8	0.01

Table 3: Optimal parameters for Small lake

Question 4

For the following discussion, we define:

- S as the set of all possible states
- A as the set of all possible actions

In linear action-value function approximation, if the feature vector $\phi(s, a)$ (for some $s \in S$ and $a \in A$) is such that the element at the index corresponding to the state-action pair (s, a) is 1, while every other element is 0, then the parameter vector θ can be taken to approximate the action-value function, wherein each element is the action-value of a particular action-value pair.

To elaborate, the vector θ and $\phi(s, a)$ here are both of size $|A||S|$, wherein each index corresponds to a particular state-action pair. Hence, the dot product of the feature vector $\phi(s, a)$ and parameter vector θ produces the action-value for the particular state-action pair (s, a).

This is identical to a tabular model-free algorithm, as the entire action-value function is being learnt with no linear approximation taking place. Therefore, tabular model-free methods are a special case of linear approximation methods when the following conditions are met:

- The feature vector $\phi(s, a)$ has size $|A||S|$
- Every element in the feature vector $\phi(s, a)$ is 0 except for 1
- Every state-action pair has a different feature vector

Question 5

An *epsilon*-greedy policy is a policy that acts according to a greedy policy with probability $1 - \epsilon$ and acts randomly with probability ϵ. This is necessary during training to ensure exploration of the state space. If the algorithm just acted greedily with respect to Q, the neural network would risk getting stuck at a local optimum, i.e. a policy that seems optimal with respect to the agent's very limited knowledge, but which may be observed to be sub-optimal upon knowing the environment further. To avoid this issue, it is necessary not only to expand the most promising moves (as per the agent's knowledge) but also to explore the environment for moves that may turn out to be better, especially in the long-run. This problem of balancing experience-based knowledge-gathering and experience-based reward-seeking is the exploration-exploitation dilemma, and an ϵ-greedy policy is a simple and efficient way to overcome it.

Question 6

The authors explain that estimating the action-value function with a non-linear function approximation is known to have a tendency to change (across the learning process) in an unstable way or even to diverge, so as to never converging to an estimate that lets us obtain an optimal or near-optimal policy.

Including a target Q-network in addition to an online Q-network is a way to combat this tendency toward instability by keeping the target Q-values more stable, i.e. less frequently updated than the currently estimated Q-values. A target value in this context is a value considered as a benchmark against which the network should update the values of the action-rewards stored in Q.

In other words, by keeping the target values more stable, we have a greater chance of ignoring those fluctuations in the estimated Q-values that would lead away from a convergence to local or global optimum.

Code structure

All of the environment classes and algorithms are structured into two python packages:

- **agents:** contains all the RL agent algorithms and the classes necessary
  - *tabular_model_rl.py:* Policy Iteration and Value iteration algorithms
  - *tabular_model_free_rl.py:* SARSA Control and Q-Learning Control algorithms
  - *non_tabular_model_free_rl.py* Linear SARSA Control, Linear Q-Learning Control and Deep Q Learning algorithms
  - *DeepQNetwork.py:* The Q-Network Neural Network class
  - *ReplayBuffer.py:* The replay buffer for DQL
- **environment:** contains all the files related to the environment.
  - *EnvironmentModel.py:* Base class for environments
  - *Environment.py:* Abstract class for an environment
  - *FrozenLake.py:* Class for the Frozen Lake environment
  - *LinearWrapper.py:* A Wrapper for the Frozen lake environment used for linear sarsa and q learning
  - *FrozenLakeImageWrapper.py:A* Wrapper for the Frozen lake environment used for Deep Q Learning
  - *EpisodeRewardsWrapper.py:* A Wrapper for an environment that keeps track of all rewards obtained during each episode of training

The scripts used to run the algorithms and produce the data to answer the questions in this report are found in the root folder of the code base:

- *tests.py:* runs tests for the environment implementation
- *main.py:* main function that runs all agents and prints resulting policy and value
- *count_tabular_model_iterations.py:* script used for question 1 of the report, logs the number of iterations taken for policy and value iteration to converge
- *train_agent_and_plot_returns:* script used for question 2 of the report, plots returns of each algorirthm
- *optimise_tabular_model_free.py:* script used for question 3 of the report, runs a grid search for the learning rate and epsilon values
- *utils.py:* utility functions used throughout the code base

Implementation notes

Policy representation

For all the methods in this project, we consider the policy to be deterministic, meaning that it maps each state to a certain action, rather than each state-action pair to a probability. Hence, 'policy' is a 1D array with each index corresponding to a state and the value at a given index i corresponding to the action to be taken from state i. This is equivalent to the policy wherein each state-action pair is related to either 0 or 1 so that each state is mapped to 1 only when paired with a particular action.

State and action value function representation

The state-value function is implemented with an array of $|S|$ values (where S is the set of all possible states), wherein each index corresponds to a state. Hence, mapping values to any state is done by simply storing the state's value in the corresponding index.

The action-value function is implemented in two ways. The first way (used in tabular methods) is a $|S| \times |A|$ matrix, wherein the row index corresponds the a state and the column index corresponds to a value. The second way (used in non-tabular methods) is essentially the parameter vector. A given state's action-values are obtained by taking the dot product of the feature vector and the parameter vector.

Breaking ties between reward-maximising actions

When the The ϵ-greedy policy chooses to exploit rather than explore, it may be the case that there exist multiple actions that (within a tolerance level) maximise the action-value function from a given state. In such a case, exploration can be encouraged even during exploitation by making a random selection from the reward-maximising actions. To achieve this, we create a list of actions that maximise the action-value for a given state, wherein the comparison of each action's action value to the maximum action-value is made with a tolerance level. This tolerance level is kept at the default values defined in the function **numpy.allclose**. Then, random selection is done on this list of actions to next action action.

Notes on Deep Q-network learning

What learning means in deep Q-network

Learning is done by performing gradient descent over the minimum squared error (MSE) loss function. Instead of obtaining loss with respect to differences from observed values alone (observed action-values in our case), we obtain the loss as the mean of the sum of squares of the temporal differences for each state-action pair (the implementation details are discussed in the following subsection). This is done because rewards are very sparse in the given environment, which means obtaining accurate estimates of action-values from interaction with the environment alone will take an unfeasibly long time, prompting us to instead use temporal differences (discussed further in the following subsection).

Online network, target network and replay buffer

Online and target networks

- DQN = Online deep Q-Network
- TDQN = Target Deep Q-Network

DQN is meant to be the up-to-date network (i.e. the online network with the updated weights) using which we estimate expected rewards for state-action pairs. TDQN is meant to be the previous network (i.e. before current or recent updates to the weights) using which we obtain the previous estimated expected rewards for state-action pairs. These previous estimates are used as benchmarks to update the current network using temporal difference, i.e. these previous estimates are used as the "observed target" values for training the network. Note that we can set the parameters such that TDQN is the same as DQN. However, keeping TDQN less frequently updated can help avoid (1) getting stuck in a local optimum and (2) preventing (to some extent) the estimated action-values from being affected by fluctuations that move away from convergence.

Replay buffer

The replay buffer stores state-transitions (or simply "transitions"). A transition is a tuple composed of a state, action, reward, next state, and a flag variable that denotes whether the episode ended at the next state. In the Python implementation, the buffer is represented by a Python **deque** object that automatically discards the oldest transitions when it reaches capacity, thus preventing the saturation of data points to draw from. The method **draw** in the replay buffer class (defined by our code to encapsulate the replay buffer's memory and functions) returns a list of n transitions (n denotes the batch size) drawn without replacement from the replay buffer. The replay buffer is vital in utilising previously obtained state transitions and observed rewards.

Theoretical connection

Given the following temporal difference:

$$r_{t+1} + \gamma \max_{\forall a \in A}(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

Here:

- t: Current time stamp

- A: The set of all possible actions
- Actions are chosen based on *epsilon*-greedy policy
- We take action a_t from state s_t
- We take action a_{t+1} from state s_{t+1}
- r_{t+1}: Observed reward of taking a_t from s_t
- $Q(s_t, a_t)$: Estimated reward of taking a_t from s_t
- $\max(Q(s_{t+1}, a))$: Maximum action-reward possible from s_{t+1}
- γ: Discount factor

Connecting the above to the implementation...

- $max(Q(s_{t+1}, a))$ is computed using TDQN
- $Q(s_t, a_t)$ is computed using DQN
- a_t, s_t, r_{t+1} and s_{t+1} are picked from replay buffer

Vanishing ReLU problem in deep Q-network

If the learning rate for the network is set too high, the gradient descent process will tend to overshoot the optimum. The result of this in our case is that the weights become highly negative overall, producing results such that applying ReLU (rectified linear unit) leads to a zero-matrix; this is the vanishing ReLU problem. This zero-matrix leads to every resultant row of the final output (i.e. the action-values for each state) being equal, leading to a situation where:

- The forward model produces the same action-values for each action, no matter the state
- As a result of the above, the same maximum action-value is indicated for each state
- As a result of the above, action-values and thus policy converge to the same value and same action for each state

Hence, we must set the learning rate sufficiently low to prevent such an overshooting gradient descent and the consequent vanishing ReLU problem.