Exploring deep reinforcement learning

Benny Collins Maeve Hutchinson

Basic

I. Define an environment and the problem to be solved

The environment to be used is a custom grid world environment of dimensions 25 by 25 units, with obstacles placed to form a maze. There are penalties placed at each dead end in the maze, and the door to exit the maze is placed at (1, 12). The start position of the agent at the beginning of each episode is randomised. Figure 1 shows a visualisation of the environment.

A state in the environment is defined by the agent position. The states are numbered from 0 to 624, starting in the top left corner of the grid, moving horizontally until the bottom right corner.

The objective is for the agent to reach the door in the fewest steps possible, whilst avoiding colliding with the walls or the penalties.

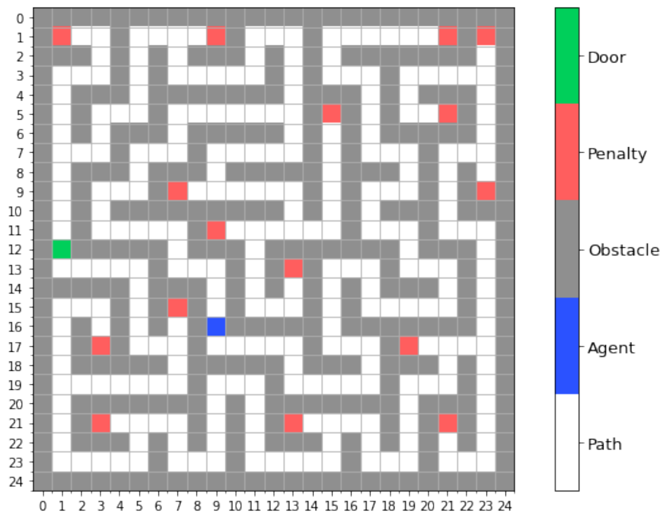

Fig. 1. A visualisation of the grid world environment to be solved.

II. Define a state transition function and the reward function

In each state, there are four actions available to the agent: up, down, left and right. All actions are available in each state, but not all actions will result in a state transition; if the agent chooses an action that would move it into an obstacle, then the agent remains in the same state. All other state transitions — into path cells, penalties, or the door — are allowed. The environment has one terminal state, which is when the agent reaches the door. If the agent does not reach the door, an episode terminates after 625 steps.

For each action taken, the agent receives a reward of -1. This is to encourage the agent to reach the door as quickly as possible. The penalty cells incur a negative reward of -312, and if the agent attempts to enter an obstacle the reward is -156. The only state with a positive reward is the terminal state, in which the agent receives a reward of 625 for having solved the maze.

III. Set up the Q-learning parameters (gamma, alpha) and policy

A. Q-learning Parameters

The Q-matrix for this problem is initialised as an array, full of zeros, of the same dimensions as the environment. At each time step, the agent chooses an action, a, based on a certain policy, π, and then the Q-matrix is updated using the following algorithm:

$$Q_{\text{new}}(s,a) \leftarrow Q(s,a) + \alpha \cdot \left[r + \gamma \cdot \max Q\left(s', a\right) - Q(s,a)\right], \tag{1}$$

where Q_{new} is the new q-matrix and Q is the old q-matrix at the current state s, or new state s' resulting from chosen action a, and r is the reward for choosing that action. α is the learning rate and γ is the discount rate, the two key parameters in Q-learning [3].

α defines how quickly the agent learns, and can take a value of between 0 and 1. An α value of 0 means that the Q-values are never updated, so the agent never learns. As α gets higher, the faster the agent learns, as the Q-values are updated by a larger amount at each step [4]. Initially, α will be set to 0.5, but different values will be explored.

γ defines the importance that the agent places on future rewards. A γ value of 0 means that the agent only focuses on immediate rewards, whereas higher values cause it to focus more on long-term rewards.Initially γ will also be set to 0.5, with further exploration intended.

B. Epsilon-Greedy Policy

The first policy to be implemented is an epsilon greedy policy, which has one parameter, ϵ, which takes a value of between 0 and 1. In a given state, this policy chooses the greediest action — the action with the highest Q-value for that state — with probability $1 - \epsilon$, otherwise it selects an action randomly [3]. So, when ϵ is 1, the policy chooses completely randomly, whereas when it is 0 the choice is always greedy. The value of ϵ can be decayed after each episode so that the actions of the agent become more greedy over time as it approaches the optimal policy.

We will use an initial value of $\epsilon = 1$, and a decay value of 0.999, which ϵ will be multiplied by after completing each episode, until it reaches a value of 0.001 when it will be decreased no further.

C. *Softmax Policy*

The second choice of policy, a softmax policy, also has one parameter, τ, known as the temperature. In a given state the probability, p of choosing a certain action, a, is described by the Boltzmann distribution:

$$p(a) = \frac{\exp\left(\frac{Q(a)}{\tau}\right)}{\sum_{i=1}^{n} \exp\left(\frac{Q(a_i)}{\tau}\right)}, \tag{2}$$

where $Q(a)$ is the Q-value of action a in the current state, and n is the number of available actions in that state [3].

As τ approaches 0, the action choice becomes more greedy. So, τ can also be decayed in a similar manner to ϵ. We will use an initial value of $\tau = 5$, and a decay value of 0.99, until it reaches a value of 0.001 when it will be decreased no further.

IV. Run the Q-learning Algorithm and Represent its Performance

Initially, the Q-learning algorithm was run twice, once with an epsilon greedy policy, and once with a softmax policy, using the parameters outlined above.

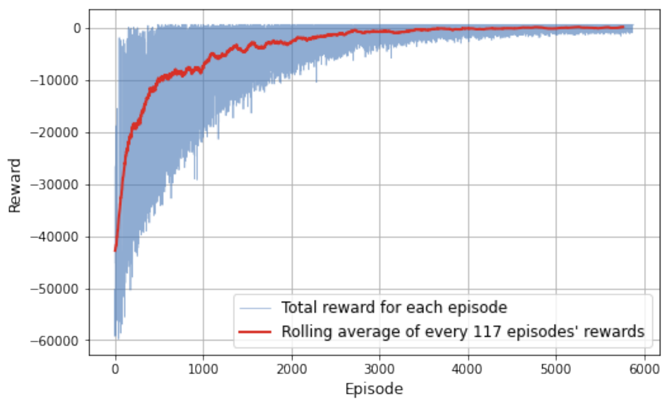

Fig. 2. The total reward for each episode run using an epsilon greedy policy with $\epsilon = 1$, decay $= 0.999$, and $\alpha = \gamma = 0.5$.

Additionally, we implemented early stopping criteria, so that the algorithm stops running when the agent has learnt to solve the maze sufficiently: the rewards of the previous 100 episodes are examined, and when both their mean is above 0, and their standard deviation is less than 625, the learning is considered to have converged, and learning is ceased. These criteria were chosen because a high mean and low standard deviation in the episode rewards means that the agent is frequently solving the maze. If these criteria are not met, the algorithm stops after 10,000 episodes.

The results have been shown by plotting the total reward for each episode, with a rolling average plotted over the top. Figure 2 shows the results of running the algorithm with an epsilon greedy policy. Learning converged at episode 5881, with the final 100 episodes having a mean reward of 200.0, and a standard deviation of 621.7.

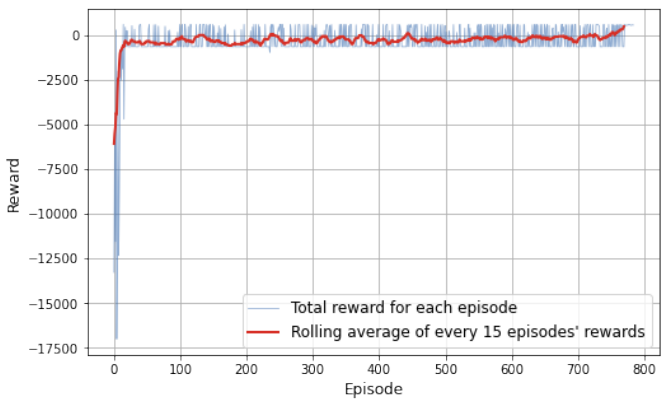

Fig. 3. The total reward for each episode run using a softmax policy with $\tau = 5$, decay $= 0.99$, and $\alpha = \gamma = 0.5$.

Figure 3 shows the results of running the algorithm with a softmax policy. The results have been shown by plotting the total reward for each episode, with a rolling average plotted over the top. Learning converged at episode 781, with the final 100 episodes having a mean reward of 0.93, and a standard deviation of 597.5.

Whilst the softmax policy converged much faster than the epsilon greedy policy, it converged at a much lower reward value. This is probably because, initially, when the value of epsilon is quite high, the epsilon greedy policy chooses randomly more often than greedily, and it is equally likely to pick any action, including actions that have a highly negative reward. Conversely, the softmax policy only chooses randomly when the q-matrix for a given state is empty, so as soon as one action in a state is taken, it uses that to inform future action choices, meaning it learns to avoid penalty cells faster than an epsilon greedy policy, increasing the mean reward rapidly.

However, upon further examination of the low mean reward value, we found that it was caused by the agent rarely actually completing the maze, and thus not receiving the high final reward. Thus, for the rest of the analysis, we will continue with the epsilon greedy policy only as this policy seems to be allowing the agent to actually learn the optimal policy to solve the maze.

V. Repeat the Experiment with Different Parameter Values, and Policies

To find the optimal values of γ and α, a grid search was implemented over γ values from 0 to 1 in steps of 0.2, and α values from 0.1 to 0.9 in steps of 0.2. The metric used to compare these values is the number of episodes it takes for the learning to converge, where a lower number is better. The results of the first run can be seen in figure 4. There is a clear minimum at $\gamma = 0.8$, but the impact of alpha is unclear, especially as γ gets higher.

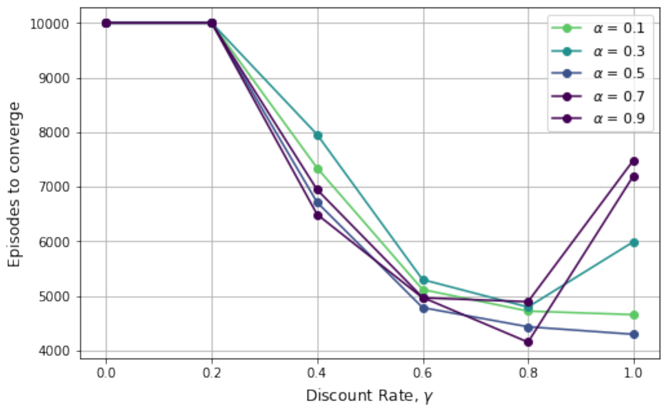

Fig. 4. A grid search over different values of α and γ to find the optimal parameters for the Q-learning algorithm. The policy used was an epsilon greedy policy with $\epsilon = 1$ and decay $= 0.999$.

Thus, we decided to do another run over a smaller interval of γ from 0.6 to 0.9 in steps of 0.1, and α of 0.5 to 0.9. The results of this grid search can be seen in figure 5. The minimum is slightly less clear in this graph, but it is still around $\gamma = 0.8$. Again, the impact of α is still unclear, so we decided to remain with $\alpha = 0.5$, as this value had the lowest number of episodes for each value of γ.

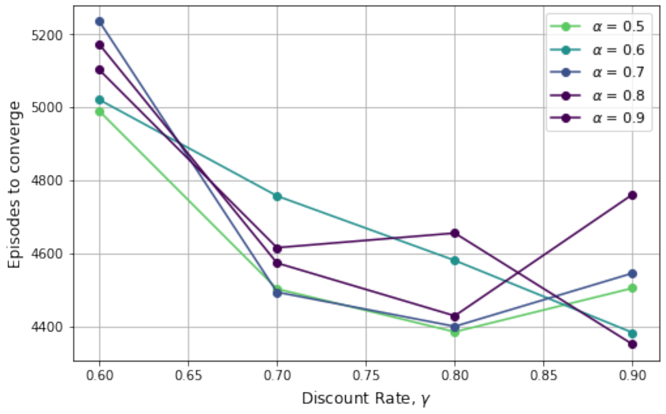

Fig. 5. A grid search over a smaller range of values of α and γ to find the optimal parameters for the Q-learning algorithm. The policy used was an epsilon greedy policy with $\epsilon = 1$ and decay $= 0.999$.

VI. Analyse the Results Quantitatively and Qualitatively

Figure 6 shows the results using the optimised values of $\gamma = 0.8$ and $\alpha = 0.5$ with an epsilon greedy policy with an initial value of $\epsilon = 1$ and a decay value of 0.999. The learning converged after 4551 episodes, and the rewards of the final 100 episodes have a mean of 331.17 and a standard deviation of 620.22.

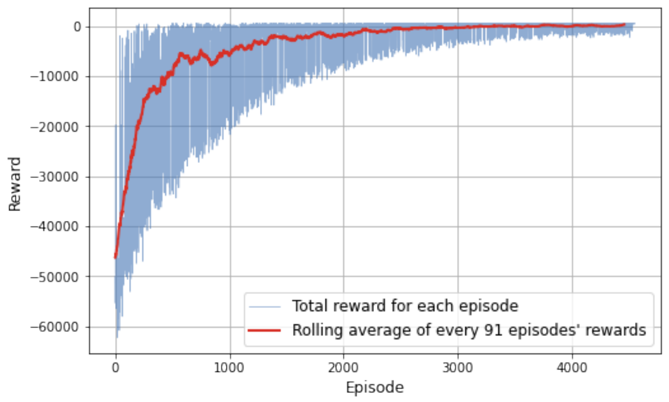

Fig. 6. The total reward for each episode run using an epsilon greedy policy with $\epsilon = 1$, decay $= 0.999$, $\alpha = 0.5$, and $\gamma = 0.8$.

To test these results further, the Q-matrix obtained from this policy was then used as the policy for 1000 episodes. Figure 7 shows the agent using this policy from the starting position in figure 1. In this example, the agent does take the optimal route of 58 steps. Figure 8 shows the results of all 1000 runs. It can be seen that very often the agent receives a total episode reward that is over 600, indicating that it reached the door in an optimal or near-optimal number of steps, however, sometimes it receives a much lower reward near 0. Out of the 1000 episodes, the agent reached the door 78.8% of the time, indicating that the policy learnt is nearly an optimal policy, and definitely much better than choosing actions randomly.

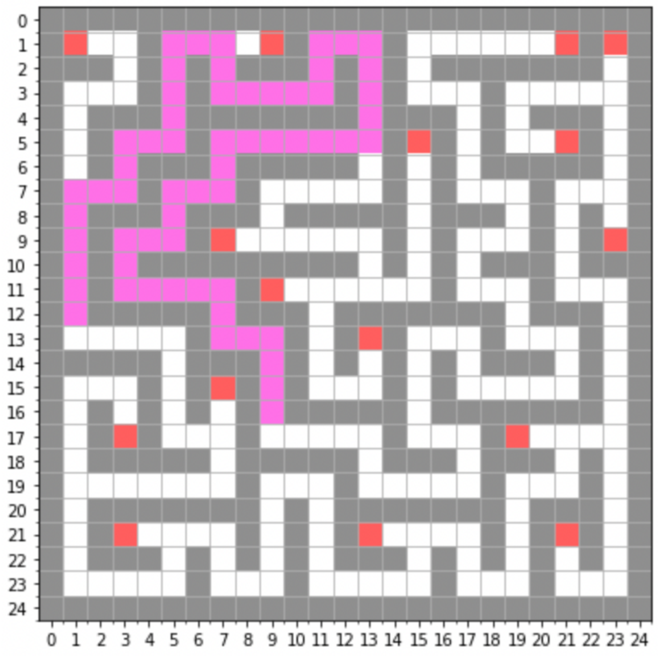

Fig. 7. A visualisation of the path taken by the agent in the environment from figure 1, when using the policy learnt from the tuned epsilon greedy policy.

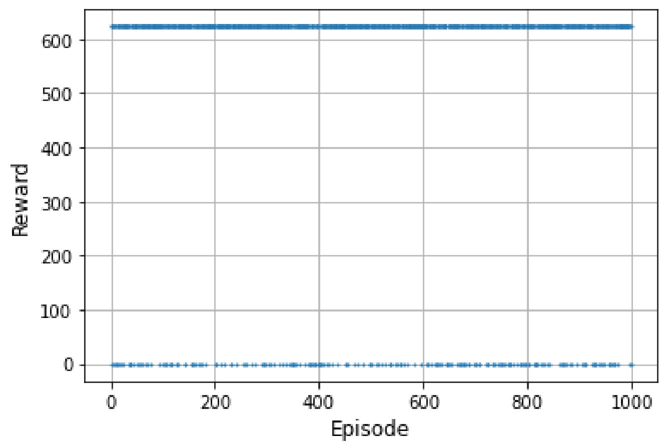

Fig. 8. The total reward for each episode when using the policy learnt from the tuned epsilon greedy policy.

ADVANCED

VII. IMPLEMENT DQN WITH TWO IMPROVEMENTS

Q-learning shows good performance for a grid-world maze environment, however, the agent is trained on a singular maze, which resets to its original state at the end of each episode. This results in an agent that can find an optimal path for this single maze but can't adapt to states and environments it hasn't already encountered. Deep Q-networks are able to adapt to unseen states, and are often employed to tackle environments with increased complexity. Through approximation of the action-value function, using neural networks, deep Q-networks eliminate the Q-table used in Q-learning, allowing the agent to learn more efficient paths for maze environments that are randomised at the start of each episode.

The states returned by the Maze class are adapted to work as input for the Deep Q-networks. Rather than returning the entire environment for a given state, the coordinates of the door relative to the agent is calculated and the surrounding states, in a flattened 3x3 grid, are concatenated to the relative position. This provides all requisite information for a given state through a positional element and an element depicting all potential next states, whilst also decreasing the number of values needed to define a state from the size of the full environment, 625 by default, to 11 values. Deep Q-networks make use of Experience Replay - a technique that defines a Replay Memory buffer, with size limit N, containing experiences stored in the form:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})\,, \tag{3}$$

where s_t is the state and a_t is the action taken at time t, r_{t+1} is the reward for action a_t and s_{t+1} is the next state.

An experience, from a particular state, is acquired through action-selection by the policy network, with respect to the state and policy, before executing the given action in an emulator to obtain the reward and next state. The buffer is filled prior to training the network and updated with a new experience, replacing the oldest experience, at each timepoint during training [1].

Each update of the Q-network parameters is calculated using a batch of random samples of experiences from the buffer, breaking the correlation between consecutive sequential experiences that can lead to inefficient learning. When training the Q-network, a batch is randomly sampled from the Replay Memory. Within each experience in the batch, the state is passed to the Q-network, which acquires Q-values for each action that can be taken from that state through forward propagation. The Q-value corresponding to the action taken in the experience is taken as the current Q-value. The next state in the experience is then passed into the Q-network, with the maximum Q-value output being used to calculate the target Q-value:

$$Q_*(s,a) = r_{t+1} + \gamma \max_{a'} Q_*(s', a')\,, \tag{4}$$

where $Q_*(s,a)$ is the target Q-value for state s and action a, r_{t+1} is the reward for action a, γ is the discount rate, s' is the next state from s, and a' is the action taken from the state s' [3].

The loss is calculated - the Mean Squared Error of the target Q-value and the current Q-value - and then minimised through backpropagation and stochastic gradient descent on the weights and biases of the Q-network.

The DQN architecture begins with 11 input neurons, where a preprocessed state acts as input. This input is passed through a number of fully-connected linear hidden layers, each containing a specified number of neurons. The input is fed into the first hidden layer, where batch normalisation is applied, before being fed through two more hidden layers with the same structure. The final layer has a linear connection to the outputs layer, which has a number of neurons equal to the number of actions available, and outputs Q-values for each action from the input state.

Using a single network for both action selection and action evaluation often results in a bias towards overestimating the target Q-values, known as maximisation bias. This is because minimising the loss involves performing gradient descent on the weights of the Q-network. If only one network is used to calculate both sets of Q-values, then updating the parameters causes the next set of Q-values to be shifted in the same direction. This can lead to the Q-network learning sub-optimal policies

The first improvement made to the algorithm was implementing Double DQN, involving using two separate Q-networks. The policy network is used in the process of calculating the current Q-values and updating model parameters, as well as experience acquisition when filling the buffer. The second network, the target network, is initialised as a duplicate of the policy network, only updating it's parameters to match that of the policy every specified interval of episodes. This improvement should enhance convergence time, as well as the optimality of the policy that is learned [3]. The second improvement applied to the Deep Q-network was Duelling.

This involves altering the Q-network architecture to split into two streams - the value stream, for approximating state values $V(s)$, and the advantage stream, for approximating the advantage $A(s,a)$ for a particular action - which are combined to form Q-values, using the equation:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a) \right), \quad (5)$$

where $\mathcal{A}$ is the advantage space for all actions that can be taken from a given state [2].

This allows the network to better differentiate between actions as well as not having to learn the value of each action from each state in order to learn state values, which should improve the learned policy.

VIII. Analyse the DQN results quantitatively and qualitatively

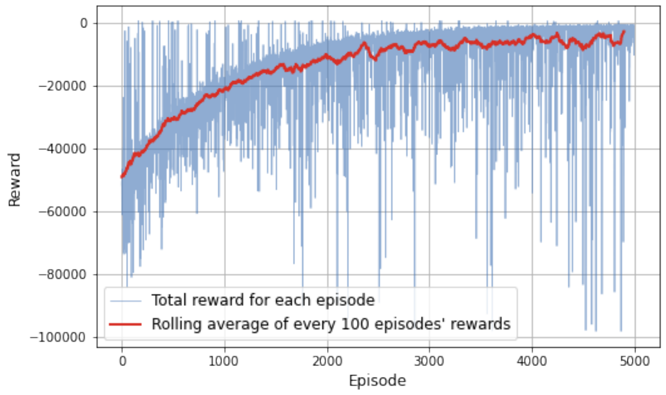

Fig. 9. Vanilla DQN.

Figure 7 displays the reward over 5000 episodes of training our vanilla DQN, with $\gamma = 0.5$ and $\alpha = 0.01$, using an epsilon greedy policy with an initial value of $\epsilon = 1$ and a decay value of 0.999. Replay buffer size is set to $N = 10000$ and batch size is set to 256, as it is better for this to be a power of 2, and the number neurons in the hidden layer is 128. We can see that the reward improves from a 100-episode average reward of about -45000, to a 100 episode average fluctuating between approximately -4000 and -8000.

Figure 8 displays the reward over 5000 episodes of training our Double DQN, with all the same hyperparameters as the vanilla DQN. The rewards are very similar to those of the vanilla DQN, with similar improvements in the 100-episode average. The lack of improvement in results could be caused by either an implementation problem, the number of episodes being too low, or the environment's use of a large number of negative rewards with only one positive reward, meaning almost all states would have very negative state values.

The rewards of our DDDQN, trained over 5000 episodes, are shown in Figure 9. The model was trained using the same hyperparameters as the other two models, except for the size of the hidden layer in the network. To reduce runtime, this

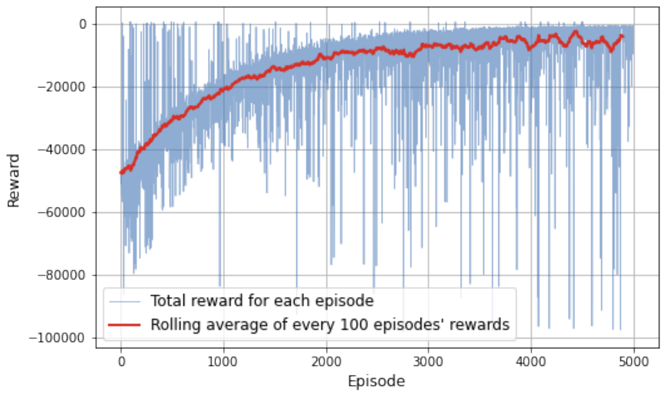

Fig. 10. Double DQN.

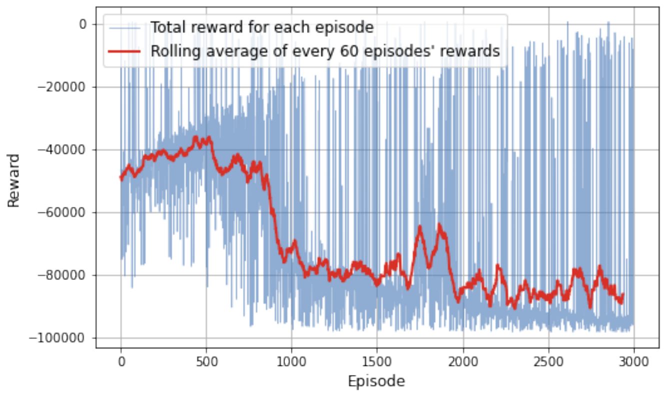

Fig. 11. DDDQN.

hyperparameter was decreased to 64, decreasing the size of the network architecture, as the size of a network is positively correlated with convergence time. Unfortunately, we can see from these results that the DDDQN did not converge over the 5000 episodes, with the mean 100-episode reward initially slightly increasing before decreasing to a fluctuation between approximately -70000 and -90000. These results could be due to an implementation error or an unstable convergence that needs a very large number of episodes in order to reach a satisfactory 100-episode mean reward.

To improve the results for the different implementations of DQN, hyperparameter tuning could be applied thoroughly. It seems it would also be worth training the DQNs with environments that have been altered to include a higher number of positive rewards, in order to help guide the agent. Training on a larger number of episodes may also produce interesting results.

It must also be noted that the results for these models are expected to be worse than the results for the Q-learning, as the Q-learning is only implemented on one maze, whereas the various DQN models are implemented on maze environments that are randomised each episode. This randomisation thoroughly increases fluctuations in the rewards as well as increasing the number of episodes required in order to converge to a policy.

IX. Apply an RL algorithm to one of the Atari Learning Environments

The A3C algorithm was selected to train an agent to play Pong from the OpenAI Atari environments. The algorithm selection was dictated by its adaptability to different environments, as well as it not requiring as much memory, as it does not store samples in the memory. A3C's adaptability is a result of it being an Actor-Critic method - a set of methods that combine value-based methods and policy-based methods.

Value-based methods learn a value that will map each state-action pair to a value - such as the Q-values in Q-learning and DQN - where the value function is updated to optimise the policy. The action taken from the current state is determined by the maximum value mapped from the state-action pair. These types of methods work well on discrete action spaces, but encounter issues when they are implemented on continuous, or very large, action spaces.

Policy-based methods directly optimise a policy as opposed to using a value function. In contrast to value-based methods, policy-based methods are applicable to continuous action spaces. The policy is evaluated through Monte Carlo methods, which calculate a score function over the course of an entire episode, requiring a full episode to run before any updates are made to the policy. Due to the number of actions taken in an episode, and the number of rewards accumulated from this, the score function is often subject to high variance, which results in fluctuations and slow learning.

Actor-Critic methods combine policy-based and value-based learning through the use of 2 networks. This includes an Actor network, which picks actions in a policy-based fashion, and a Critic network, which evaluates the action returned by the Actor network using a value-based approach. The Critic network is approximating the value function, which is used to replace the score function for the policy-based Actor network. This results in updates being made after each step, as opposed to each episode, reducing the variance of the score function, speeding up learning. In the A3C algorithm, multiple Actors are employed to train in parallel, obtaining large amounts of training data, allowing for policy gradients to be implemented without the risk of highly-correlated training data.

X. Analyse the results quantitatively and qualitatively

The model is trained using 8 CPUs and 0 GPUs with, otherwise, largely default parameters in the config. The episode reward mean increases, with fluctuation, from approximately -20 to 0 over the course of almost 160 million steps. The mean reward has not yet fully converged, as the run was taking a very long time, however, we can plainly see that the average reward is improving over training. Higher mean reward performance and convergence time could both be improved via grid search hyperparameter tuning, which was not completed due to time constraints.

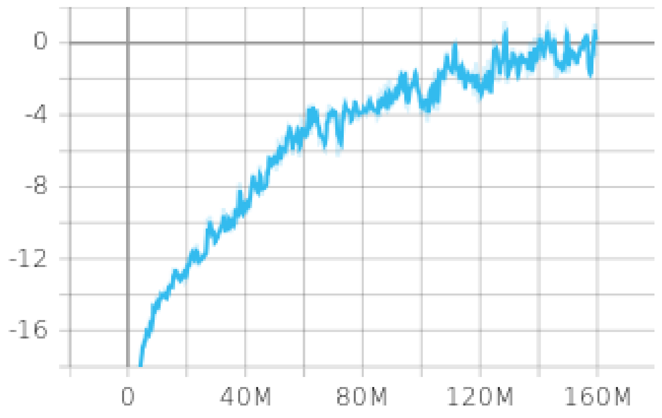

Fig. 12. The mean reward for each episode plot against the number of steps taken in training

There are also other algorithms that could have been employed to train the agent. One of these is the PPO algorithm, which converges over a much shorter period of time and is considered, despite its simplicity, to be state of the art for training an agent in the Pong environment. The other algorithm that could have been used is the IMPALA algorithm, which is also an Actor-Critic method, and has been shown, with the requisite tuning, to achieve satisfactory rewards in as low as 3 minutes of training time.

XI. Contribution

Most of the code was written in partnership, face-to-face. What little that was written separately was written in parallel, face-to-face, with constant input and discussion on methodology and approach to understanding of the algorithms and problems within the code.

References

[1] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. arXiv: 1811.12560.

[2] Nimish Sanghi. Deep Q-Learning. In Nimish Sanghi, editor, *Deep Reinforcement Learning with Python: With PyTorch, TensorFlow and OpenAI Gym*, pages 155–206. Apress, Berkeley, CA, 2021.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 1998.

[4] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

Team code is available at:
https://github.com/maevehutch/INM707-Collins-Hutchinson
Individual code is available at:
https://github.com/BennyCollins/INM707-Collins

A Basic Task - Q Learning

This file covers the code for the basic tasks of the coursework. We define a custom grid world enviroment, with a maze to be solved in as few steps as possible. We also define two policies: epsilon greedy with decay, and softmax. We run the Q-learning algorithm on the environment using both of these policies, and varying values of gamma and alpha.

[2]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
from matplotlib import colors
from matplotlib.ticker import AutoMinorLocator
```

A.0.1 The Enviroment

Inspiration for this dungeon environment was gained from the code presented in Labs.

[3]:
```
class Maze:

    def __init__(self, N=25):
        """
        This class defines a maze environment in a square 2-dimensional numpy␣
↪array.
        N: dimensions of array, integer, default 25

        There are penalties placed at every dead end in the maze.
        The agent and door are at randomised positions in the maze.
        In the array, these are represented by:
        0: part of the maze path, 1: the agent, 2: maze wall/obstacle, 3:␣
↪penalty, 4: door
        """
        self.N = N

        # Generating the maze.
        self.grid = np.full((N,N),2)
        self.generate_maze()

        # Placing penalties at every dead end.
        self.penalties_pos = self.generate_penalties()

        for cell in self.penalties_pos:
            self.grid[cell] = 3

        # Adding door and agent at random positions, and getting the current␣
↪state of the agent.
        self.door_pos = self.get_rand_empty_cells(1)
        self.grid[self.door_pos] = 4
        self.agent_pos = self.get_rand_empty_cells(1)
        self.grid[self.agent_pos] = 1

        self.state = self.get_state(self.agent_pos)

        # Setting a step counter, and the maximum number of steps per episode,␣
↪which is N squared.
        self.time_elapsed = 0
        self.time_limit = self.N*self.N

        # Defining actions available to the agent.
        self.actions = [('up',-1,0), ('down', 1, 0), ('left', 0, -1), ('right',␣
↪0, 1)]

        # Defining the reward and transition matrices for the environment.
        self.reward_matrix, self.transition_matrix = self.get_matrices()

    def generate_maze(self):
        """
        Generate a maze using an iterative depth-first search algorithm.
        """

        # Making every other cell in the grid a 1, denoting a cell that will be␣
↪part of the path but hasn't been visuted by the algorithm yet.
            # This means every other cell is currently a wall, and leaves a␣
↪border of walls around the grid.
        self.grid[1::2, 1::2] = 1

        # Picking a random unvisited cell and making it the current cell
        rand_x = random.randrange(1,self.N,2)
        rand_y = random.randrange(1,self.N,2)
        current_cell = (rand_x, rand_y)

        self.grid[current_cell] = 0 # Adding the current cell to the maze path.

        cell_stack = [current_cell] # Creating a stack to store cells to be␣
↪visited by the algorithm.

        # Algorithm runs while there are still cells in the stack to check.
        while len(cell_stack) > 0:
            # Retrieving most recent cell in stack, removing it from stack, and␣
↪obtaining its unvisited neighbouring path cells.
            current_cell = cell_stack[-1]
            cell_stack.pop(-1)
            borders = self.get_nearby_cells(current_cell, dist=2)
            unvisited_borders = [border for border in borders if self.
↪grid[border] == 1]
            # If there are unvisited borders:
            if len(unvisited_borders) > 0:
                cell_stack.append(current_cell) # Add current cell back to stack.
                # Pick bordering cell at random and remove the wall between␣
↪current cell and that cell, add that cell to stack.
                chosen_cell = random.choice(unvisited_borders)
                self.grid[(chosen_cell[0] + current_cell[0])//2, (chosen_cell[1]␣
↪+ current_cell[1])//2] = 0
                self.grid[chosen_cell] = 0
                cell_stack.append(chosen_cell)

    def generate_penalties(self):
        """
        Return a list of the coordinates of every dead end in the maze.
        """
        penalties_pos = []
        # Iterating through all path cells to check if they are dead ends.
        for i in list(np.argwhere(self.grid == 0)):
            path_cell = tuple(i)
            # Getting bordering cells, and checking which ones are also path␣
↪cells.
            borders = self.get_nearby_cells(path_cell)
            path_borders = [border for border in borders if self.grid[border] ==␣
↪0]
            # If there is only one bordering path cell, then it is dead end.
            if len(path_borders) == 1:
                penalties_pos.append(path_cell)
        return penalties_pos

    def get_matrices(self):
        """
        Generate the reward and transition matrices for the environment.

        In the reward matrix, a given value represents the reward that the agent␣
↪would recieve for taking a given action.
        Reward matrix indices are [current_state, action_index].

        In the transition matrix, a value of 0 means a given action is not␣
↪allowed, and a 1 means that it is.
        Transition matrix indices are [current_state, action_index, next_state].

        Action indices - 0: up 1: down 2: left 3: right
        """
        # Creating arrays to hold matrices - reward is by default -1 for a step,␣
↪and actions are by default disallowed.
        reward_matrix = -1 * np.ones((self.N*self.N, 4))
        transition_matrix = np.zeros((self.N*self.N, 4, self.N*self.N))

        # Iterating through entire grid.
        for i in range(self.N):

            for j in range(self.N):
                current_cell = (i,j)
                current_cell_type = self.grid[i, j]
                current_state = self.get_state(current_cell)

                for action_index, action_tuple in enumerate(self.actions):
                    next_cell = (i + action_tuple[1], j + action_tuple[2])

                    if self.out_of_bounds(next_cell):
                        next_cell_type = current_cell_type
                        next_state = current_state
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//4 # Negative reward for going out of bounds.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1 # Transition from one state to itself is allowed.
                    else:
                        next_cell_type = self.grid[next_cell]
                        next_state = self.get_state(next_cell)

                    # Transitions to empty path cells, penalties, and the door␣
↪are allowed.
                    if next_cell_type == 0:
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1
                    elif next_cell_type == 2:
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//4 # Trying to transition into a wall incurs a negative reward.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 0
                    elif next_cell_type == 3:
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//2 # Transitioning into a penality incurs a negative reward.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1
                    elif next_cell_type == 4:
                        reward_matrix[current_state, action_index] += self.
↪N*self.N # Transitioning into the door gives a large positive reward.
```

```
                            transition_matrix[current_state, action_index,␣
↪next_state] = 1

        return reward_matrix, transition_matrix

    def print_reward_matrices(self):
        """
        Display the reward matrices nicely by action.
        """
        for action_index, action_tuple in enumerate(self.actions):
            print(action_tuple[0])
            print(self.reward_matrix[:, action_index].reshape(self.N, self.N))

    def get_nearby_cells(self, cell, dist=1):
        """
        For a given cell, get the coordinates of all orthogonal cells that are a␣
↪given distance away.
        dist - distance of desired cells, integer, default 1

        Distance of 1 means returning neighbouring cells.
        """
        cell_up = (cell[0]-dist, cell[1])
        cell_down = (cell[0]+dist, cell[1])
        cell_left = (cell[0], cell[1]-dist)
        cell_right = (cell[0], cell[1]+dist)

        cell_list = [cell_up, cell_down, cell_left, cell_right]

        return [c for c in cell_list if not self.out_of_bounds(c)]

    def out_of_bounds(self, cell):
        """
        Check whether given cell coordinates are out of the bounds of the grid.
        """
        if ((cell[0] < 0) or (cell[0] > self.N-1)) or ((cell[1] < 0) or (cell[1]␣
↪> self.N-1)):
            return True
        return False

    def get_rand_empty_cells(self, n=1):
        """
        Retreive a random selection of n empty cells from the grid.
        n - integer, default 1

        Returns a list of tuples specifying the coordinates of the selected␣
↪cells.
        """
        if n == 1:
            zero_cells = np.argwhere(self.grid == 0) # retrieving empty cells
            rand_cell = random.choice(zero_cells)
            rand_cells = tuple(rand_cell)
        else:
            zero_cells = np.argwhere(self.grid == 0)
            rand_cells = random.choices(zero_cells, k=n)
            rand_cells = [tuple(i) for i in rand_cells]

        return rand_cells

    def get_state(self, cell):
        """
        Return the state number of a given cell.
        States are numbered from 0 to (N*N)-1 (N is the size of the grid),
        starting in the top left and moving left to right along each row to the␣
↪bottom right.
        """
        state_grid = np.arange(0, self.N*self.N).reshape(self.N, self.N)
        state = state_grid[cell[0], cell[1]]
        return state

    def get_cell(self, state):
        """
        Return the cell coordinates of a given state.
        """
        state_grid = np.arange(0, 25*25).reshape(25, 25)
        cell = (state_grid == state).nonzero()
        return cell

    def display(self):
        """
        Display the grid as an image.
        """
        fig, ax = plt.subplots(figsize=(7*1.61,7))
        cmap = colors.ListedColormap(['white', '#2b52ff','#8f8f8f', '#ff5e5e',␣
↪'#00cf5a'])

        boundaries = [-0.5, 0.5, 1.5, 2.5, 3.5, 4.5]
        norm = colors.BoundaryNorm(boundaries, cmap.N)
        plt.imshow(maze.grid, cmap=cmap, norm=norm)
        cell_names = ["Path", "Agent", "Obstacle", "Penalty", "Door"]
        formatter = plt.FuncFormatter(lambda val, loc: cell_names[val])
        cbar = plt.colorbar(ticks=[0, 1, 2, 3, 4], format=formatter)
        cbar.ax.tick_params(labelsize=13)

        ax.xaxis.set_minor_locator(AutoMinorLocator(2))
        ax.yaxis.set_minor_locator(AutoMinorLocator(2))

        plt.xticks(np.arange(0,25,1))
        plt.yticks(np.arange(0,25,1))
        plt.grid(visible=True, which = 'minor')

        plt.show()


    def step(self, action_index):
        """
        One step in the grid.
        action_index - the index of the action to be taken, 0: up 1: down 2:␣
↪left 3: right

        Returns:
        end - boolean, whether or not the episode has ended
        reward - integer, the reward for the taken step
        self.state - integer, the new state of the environment after the step
        """
        # Retreive reward value for action from reward matrix.
        reward = self.reward_matrix[self.state, action_index]

        # Work out new position and state if action were to be taken.
        new_pos = (self.agent_pos[0]+self.actions[action_index][1],
                    self.agent_pos[1]+self.actions[action_index][2])
        new_state = self.get_state(new_pos)

        self.time_elapsed += 1
        end = False
        door = False

        # If transition matrix says action is allowed, do action.
        if self.transition_matrix[self.state, action_index, new_state] == 1:
            self.grid[self.agent_pos] = 0
            self.agent_pos = new_pos
            self.grid[self.agent_pos] = 1

        # If agent has reached the door, terminate episode.
        if self.agent_pos == self.door_pos:
            end = True
            door = True

        # If number of steps is over the time limit, end episode.
        if self.time_elapsed > self.time_limit:
            end = True
        self.state = self.get_state(self.agent_pos)

        return end, reward, self.state, door

    def reset(self):
        """
        Reset the grid to its original state, but with the agent in a new random␣
↪position.
        Maze and door remain unchanged.

        Returns the new state of the environment.
        """
        for cell in self.penalties_pos:
            self.grid[cell] = 3

        self.grid[self.door_pos] = 4

        self.grid[self.agent_pos] = 0
        self.agent_pos = self.get_rand_empty_cells(1)
        self.grid[self.agent_pos] = 1

        self.time_elapsed = 0

        self.state = self.get_state(self.agent_pos)

        return self.state
```

A.0.2 The Policies

```
[4]: class eGreedyDecayPolicy:

    def __init__(self, epsilon = 1, decay = 0.999, min_epsilon = 0.001):
        """
        This class defines an epsilon greedy policy with decay.
        epsilon - float, default 1
        decay - float, default 0.999
        min_epsilon - float, default 0.01

        In the case where decay = 1, epsilon will remain constant.
        In this case:
        If epsilon = 1, the policy will always choose an action randomly.
        If epsilon = 0, the policy will always choose greedily.
        """
        self.eps_current = epsilon
        self.eps_initial = epsilon
        self.min_eps = min_epsilon
        self.decay = decay
    def __call__(self, state, q_matrix):

        greedy = random.random() > self.eps_current # action is greedy with␣
↪probability 1 - epsilon

        if greedy:
            action_index = np.argmax(q_matrix[state])

        else:
            action_index = random.randint(0, 3)

        return action_index

    def update(self):
        """
        Decay epsilon
        """
        self.eps_current = self.eps_current*self.decay

        if self.eps_current < self.min_eps:
            self.eps_current = self.min_eps

    def reset(self):
        self.eps_current = self.eps_initial
```

[5]:
```
class softmaxPolicy:
    """
        This class defines a softmax policy with decay.
        temp - float, default 5
        decay - float, default 0.99
        min_epsilon - float, default 0.01

        In the case where decay = 1, temperature will remain constant.
        """
    def __init__(self, temp=5, decay=0.99, min_temp = 0.001):
        self.temp_current = temp
        self.temp_initial = temp
        self.min_temp = min_temp
        self.decay = decay

    def __call__(self, state, q_matrix):

        e = np.exp(q_matrix[state]/self.temp_current)
        prob_actions = e / np.sum(e) # calculating probabilities from boltzmann␣
↪distribution
```

```
        if not np.isnan(prob_actions).any(): # checking all probs have been␣
↪calculated
            action_index = np.random.choice(range(4),p = prob_actions)
        else:
            action_index = np.argmax(q_matrix[state]) # greediest action

        return action_index

    def update(self):
        """
        Decay temperature
        """
        self.temp_current = self.temp_current*self.decay

        if self.temp_current < self.min_temp:
            self.temp_current = self.min_temp

    def reset(self):
        self.temp_current = self.temp_initial
```

A.0.3 The Q-learning Algorithm

```
[6]: def runQLearning(env, policy, alpha = 0.5, gamma = 0.5, min_eps=100, max_eps =␣
↪10000):
    """
    Run the Q-learning algorithm for a given environment using a given policy.
    alpha - float, default 0.5
    gamma - float, default 0.5
    min_eps - minimum number of episodes, integer, default 100
    max_eps - maximum number of episodes, integer, default 10000
    """

    q_matrix = np.zeros( (env.N*env.N, 4) ) # Initialise empty Q-matrox
    converged = False
    eps = 0
    rew = []
    doors = []

    while not converged: # run algorithm until convergence

        # checking what policy was used so values can be printed
        if isinstance(policy, eGreedyDecayPolicy):
            param = policy.eps_current
        elif isinstance(policy, softmaxPolicy):
            param = policy.temp_current

        eps += 1
        policy.update()
        state = env.reset()
        end = False
        step_rewards = []

        while not end: # running one episode
            # selecting action and making step
            action_index = policy(state, q_matrix)
            end, reward, new_state, door = env.step(action_index)

            # the q-learning algorithm!
            q_matrix[state, action_index] = q_matrix[state, action_index] + \
                alpha * (reward + gamma * np.max(q_matrix[new_state, :]) -␣
↪q_matrix[state, action_index])

            state = new_state
            step_rewards.append(reward)

        ep_reward = np.array(step_rewards).sum()
        rew.append(ep_reward)
        doors.append(door)

        val_matrix = np.zeros( (env.N, env.N) ) # initialising empty value␣
↪matrix to hold highest q value for each state
        for i in range(env.N):
            for j in range(env.N):
                state = env.get_state([i,j])
                val_matrix[i,j] = max(q_matrix[state])
                val_matrix[i,j] = max(q_matrix[state])


        if eps > min_eps:
            prev_100 = rew[-100:] # using previous 100 episodes' rewards
            std = np.std(prev_100)
            mean = np.mean(prev_100)
            print(env.N*env.N)
            print(f"std:{std}, mean: {mean}")

            if std < env.N*env.N and mean > 0: # checking for convergence using␣
↪reward mean and std
                converged = True

        if eps == max_eps: # stop running if we reach the maximum number of␣
↪episodes
            converged = True
            print("Did not converge")
        print(f"Episode {eps} finished. reward: {ep_reward}. param: {param}.␣
↪door: {door}")

    return eps, rew, val_matrix, q_matrix, doors
```

[7]:
```
def calculate_rolling_av(x, n):
    """
    Calculate a rolling average with n datapoints for some data x.
    """
    cumsum = np.cumsum(np.insert(x, 0, 0))
    rolling_av = (cumsum[n:] - cumsum[:-n]) / float(n)
    return rolling_av

def plot_results(rewards):
    """
    Plot the reward for each episode with a rolling average containing 50␣
↪datapoints.
    """
    fig, ax = plt.subplots(figsize=(5*1.61, 5))

    ax.plot(rewards, lw=0.75, alpha=0.6, color="#4575b4", label = "Total reward␣
↪for each episode")

    ax.set_xlabel("Episode", fontsize=12)
    ax.set_ylabel("Reward", fontsize=12)

    ax.grid()

    n = len(rewards)//50
    rolling_av = calculate_rolling_av(rewards, n)

    ax.plot(rolling_av, lw=2, color = "#d73027", label = f"Rolling average of␣
↪every {len(rewards)//50} episodes' rewards")
    ax.legend(fontsize=12)
```

A.0.4 Running the algorithm and representing its performance

[8]:
```
# Initialising the environment
random.seed(420)
maze = Maze(25)
maze.display()
```

```
[ ]: # Running epsilon greedy policy with default parameters
     random.seed(420)
     eps_policy = eGreedyDecayPolicy()
     eps_eps, rew_eps, val_matrix_eps, q_matrix_eps, doors_eps =␣
      ↪runQLearning(env=maze, policy=eps_policy)
```

```
[145]: plot_results(rew_eps)
```

```
[ ]: # Running softmax policy with default parameters
     random.seed(420)
     softmax_policy = softmaxPolicy(temp=5, decay=0.99)
     eps, rew, val_matrix, q_matrix, doors = runQLearning(env=maze,␣
      ↪policy=softmax_policy)
```

```
[146]: plot_results(rew)
```

A.0.5 Parameter Tuning

```
[ ]: # Running grid search over different values of alpha and gamma
     eps_decay_policy = eGreedyDecayPolicy()
     alpha = np.arange(0.1,1.0,0.2)
     gamma = np.arange(0,1.1,0.2)

     data = []

     for a in alpha:
         for g in gamma:
             eps_decay_policy.reset()
             eps, rew, val_matrix, doors = runQLearning(maze, eps_decay_policy,␣
      ↪gamma=g, alpha=a)
             data.append(eps)
```

```
[85]: # Plotting results
      cmap=plt.cm.get_cmap('viridis_r', 5)

      gamma = np.arange(0,1.1,0.2)
      fig, ax = plt.subplots(figsize=(5*1.61, 5))
      for i,d in enumerate(data):
          ax.plot(gamma, d, '-o', color = cmap(i+1), label = fr"$\alpha$ = {alpha[i]}")

      ax.set_xlabel(r"Discount Rate, $\gamma$", fontsize=12)
      ax.set_ylabel("Episodes to converge", fontsize=12)
      ax.legend(fontsize=11)
      ax.grid()
      plt.show()
```

```
[ ]: # Running grid search over smaller window

alpha = np.arange(0.5,0.8,0.1)
gamma = np.arange(0.6,0.9,0.1)

data = []

for a in alpha:
    for g in gamma:
        eps_decay_policy.reset()
        eps, rew, val_matrix, doors = runQLearning(maze, eps_decay_policy,␣
 ↪gamma=g, alpha=a)
        data.append(eps)
```

A.0.6 Running with tuned parameters

```
[ ]: random.seed(420)
eps_policy = eGreedyDecayPolicy()
eps_eps, rew_eps, val_matrix_eps, q_matrix_eps, doors_eps =␣
 ↪runQLearning(env=maze, policy=eps_policy, gamma = 0.8, alpha =0.5)
```

```
[43]: plot_results(rew_eps)
```

```
[44]: def runOptimalPolicy(env, q_matrix):
    """
    Run greedy policy based on given Q-matrix in a given environment
    """
    state = env.get_state(env.agent_pos)
    state_list = [state]
    end = False
    steps = 0
    while not end:
        action_index = np.argmax(q_matrix[state])
        end, reward, new_state, door = env.step(action_index)
        steps += 1
        state = new_state
        state_list.append(state)
    return steps, state_list, reward
```

```
[45]: def runOptimalPolicyLoads(env, q_matrix, episodes = 1000):
    """
    Run greedy policy based on given Q-matrix in a given environment over␣
 ↪several episodes
    """
    steps_list = []
    reward_list = []
    for i in range(episodes):
        env.reset()
        steps, state_list, reward = runOptimalPolicy(env, q_matrix)
        steps_list.append(steps)
        reward_list.append(reward)
    return steps_list, reward_list
```

[46]:
```
# Running optimal policy on exact starting position from earlier
random.seed(420)
maze = Maze(25)
steps, state_list, reward = runOptimalPolicy(maze, q_matrix_eps)
cells_visited = []
for state in state_list:
    cell = maze.get_cell(state)
    maze.grid[cell[0], cell[1]] = 5
```

[47]:
```
steps
```

[47]:
```
58
```

[48]:
```
# Running optimal policy from 1000 randomised start positions
steps, rewards = runOptimalPolicyLoads(maze, q_matrix_eps)
```

[49]:
```
# Visualising results
plt.plot(rewards, 'o', ms ='1')
plt.grid()
plt.xlabel("Episode", fontsize = 12)
plt.ylabel("Reward", fontsize = 12)
plt.show()
```

[50]:
```
np.sum(np.array(rewards)>600)
```

[50]:
```
788
```

B Advanced Task - DQN

This file covers the code for the joint advanced tasks of the coursework. We define the same custom grid world enviroment as the basic task, with a maze to be solved in as few steps as possible, but with a few modifications. We implement DQN and two improvements: double and dueling.

[ ]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms as T
```

```
from collections import namedtuple, deque
```

B.0.1 The Environment

```
[1]: class Maze:

         def __init__(self, N=25):
             """
             This class defines a maze environment in a square 2-dimensional numpy␣
     ↪array.
             N: dimensions of array, integer, default 25

             There are penalties placed at every dead end in the maze.
             The agent and door are at randomised positions in the maze.
             In the array, these are represented by:
             0: part of the maze path, 1: the agent, 2: maze wall/obstacle, 4: door
             """
             self.N = N

             # Generating the maze.
             self.grid = np.full((N,N),2)
             self.generate_maze()

             # Placing penalties at every dead end.
             self.penalties_pos = self.generate_penalties()

             for cell in self.penalties_pos:
                 self.grid[cell] = 3

             # Adding door and agent at random positions, and getting the current␣
     ↪state of the agent.
             self.door_pos = self.get_rand_empty_cells(1)
             self.grid[self.door_pos] = 4

             self.agent_pos = self.get_rand_empty_cells(1)
             self.grid[self.agent_pos] = 1

             self.state = self.get_state(self.agent_pos)

             # Setting a step counter, and the maximum number of steps per episode,␣
     ↪which is N squared.
             self.time_elapsed = 0
             self.time_limit = self.N*self.N

             # Defining actions available to the agent.
             self.actions = [('up',-1,0), ('down', 1, 0), ('left', 0, -1), ('right',␣
     ↪0, 1)]
```

```
        # Defining the reward and transition matrices for the environment.
        self.reward_matrix, self.transition_matrix = self.get_matrices()

    def generate_maze(self):
        """
        Generate a maze using an iterative depth-first search algorithm.
        """

        # Making every other cell in the grid a 1, denoting a cell that will be␣
↪part of the path but hasn't been visuted by the algorithm yet.
            # This means every other cell is currently a wall, and leaves a␣
↪border of walls around the grid.
        self.grid[1::2, 1::2] = 1

        # Picking a random unvisited cell and making it the current cell
        rand_x = random.randrange(1,self.N,2)
        rand_y = random.randrange(1,self.N,2)
        current_cell = (rand_x, rand_y)

        self.grid[current_cell] = 0 # Adding the current cell to the maze path.

        cell_stack = [current_cell] # Creating a stack to store cells to be␣
↪visited by the algorithm.

        # Algorithm runs while there are still cells in the stack to check.
        while len(cell_stack) > 0:
            # Retrieving most recent cell in stack, removing it from stack, and␣
↪obtaining its unvisited neighbouring path cells.
            current_cell = cell_stack[-1]
            cell_stack.pop(-1)
            borders = self.get_nearby_cells(current_cell, dist=2)
            unvisited_borders = [border for border in borders if self.
↪grid[border] == 1]
            # If there are unvisited borders:
            if len(unvisited_borders) > 0:
                cell_stack.append(current_cell) # Add current cell back to stack.
                # Pick bordering cell at random and remove the wall between␣
↪current cell and that cell, add that cell to stack.
                chosen_cell = random.choice(unvisited_borders)
                self.grid[(chosen_cell[0] + current_cell[0])//2, (chosen_cell[1]␣
↪+ current_cell[1])//2] = 0
                self.grid[chosen_cell] = 0
                cell_stack.append(chosen_cell)

    def generate_penalties(self):
        """
        Return a list of the coordinates of every dead end in the maze.
        """
        penalties_pos = []
        # Iterating through all path cells to check if they are dead ends.
        for i in list(np.argwhere(self.grid == 0)):
            path_cell = tuple(i)
            # Getting bordering cells, and checking which ones are also path␣
↪cells.
            borders = self.get_nearby_cells(path_cell)
            path_borders = [border for border in borders if self.grid[border] ==␣
↪0]
            # If there is only one bordering path cell, then it is dead end.
            if len(path_borders) == 1:
                penalties_pos.append(path_cell)
        return penalties_pos

    def get_matrices(self):
        """
        Generate the reward and transition matrices for the environment.

        In the reward matrix, a given value represents the reward that the agent␣
↪would recieve for taking a given action.
        Reward matrix indices are [current_state, action_index].

        In the transition matrix, a value of 0 means a given action is not␣
↪allowed, and a 1 means that it is.
        Transition matrix indices are [current_state, action_index, next_state].

        Action indices - 0: up 1: down 2: left 3: right
        """
        # Creating arrays to hold matrices - reward is by default -1 for a step,␣
↪and actions are by default disallowed.
        reward_matrix = -1 * np.ones((self.N*self.N, 4))
        transition_matrix = np.zeros((self.N*self.N, 4, self.N*self.N))

        # Iterating through entire grid.
        for i in range(self.N):

            for j in range(self.N):
                current_cell = (i,j)
                current_cell_type = self.grid[i, j]
                current_state = self.get_state(current_cell)

                for action_index, action_tuple in enumerate(self.actions):
                    next_cell = (i + action_tuple[1], j + action_tuple[2])
                    if self.out_of_bounds(next_cell):
                        next_cell_type = current_cell_type
                        next_state = current_state
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//4 # Negative reward for going out of bounds.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1 # Transition from one state to itself is allowed.
                    else:
                        next_cell_type = self.grid[next_cell]
                        next_state = self.get_state(next_cell)

                    # Transitions to empty path cells, penalties, and the door␣
↪are allowed.
                    if next_cell_type == 0:
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1
                    elif next_cell_type == 2:
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//4
                        transition_matrix[current_state, action_index,␣
↪next_state] = 0
                    elif next_cell_type == 3:
                        reward_matrix[current_state, action_index] -= self.
↪N*self.N//2 # Transitioning into a penality incurs a negative reward.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1
                    elif next_cell_type == 4:
                        reward_matrix[current_state, action_index] += self.
↪N*self.N # Transitioning into the door gives a large positive reward.
                        transition_matrix[current_state, action_index,␣
↪next_state] = 1

        return reward_matrix, transition_matrix

    def print_reward_matrices(self):
        """
        Display the reward matrices nicely by action.
        """
        for action_index, action_tuple in enumerate(self.actions):
            print(action_tuple[0])
            print(self.reward_matrix[:, action_index].reshape(self.N, self.N))

    def get_nearby_cells(self, cell, dist=1):
        """
        For a given cell, get the coordinates of all orthogonal cells that are a␣
↪given distance away.
```

```
        dist - distance of desired cells, integer, default 1

        Distance of 1 means returning neighbouring cells.
        """
        cell_up = (cell[0]-dist, cell[1])
        cell_down = (cell[0]+dist, cell[1])
        cell_left = (cell[0], cell[1]-dist)
        cell_right = (cell[0], cell[1]+dist)

        cell_list = [cell_up, cell_down, cell_left, cell_right]

        return [c for c in cell_list if not self.out_of_bounds(c)]

    def out_of_bounds(self, cell):
        """
        Check whether given cell coordinates are out of the bounds of the grid.
        """
        if ((cell[0] < 0) or (cell[0] > self.N-1)) or ((cell[1] < 0) or (cell[1]␣
↪> self.N-1)):
            return True
        return False

    def get_rand_empty_cells(self, n=1):
        """
        Retreive a random selection of n empty cells from the grid.
        n - integer, default 1

        Returns a list of tuples specifying the coordinates of the selected␣
↪cells.
        """
        if n == 1:
            zero_cells = np.argwhere(self.grid == 0) # retrieving empty cells
            rand_cell = random.choice(zero_cells)
            rand_cells = tuple(rand_cell)
        else:
            zero_cells = np.argwhere(self.grid == 0)
            rand_cells = random.choices(zero_cells, k=n)
            rand_cells = [tuple(i) for i in rand_cells]

        return rand_cells

    def get_state(self, cell):
        """
        Return the state number of a given cell.
        States are numbered from 0 to (N*N)-1 (N is the size of the grid),
        starting in the top left and moving left to right along each row to the␣
↪bottom right.
```

```
        """
        state_grid = np.arange(0, self.N*self.N).reshape(self.N, self.N)
        state = state_grid[cell[0], cell[1]]
        return state

    def display(self):
        """
        Display the grid as an image.
        """
        return plt.imshow(self.grid)

    def step(self, action_index):
        """
        One step in the grid.
        action_index - the index of the action to be taken, 0: up 1: down 2:␣
↪left 3: right

        Returns:
        end - boolean, whether or not the episode has ended
        reward - integer, the reward for the taken step
        self.state - integer, the new state of the environment after the step
        """
        # Retreive reward value for action from reward matrix.
        reward = self.reward_matrix[self.state, action_index]

        # Work out new position and state if action were to be taken.
        new_pos = (self.agent_pos[0]+self.actions[action_index][1],
                   self.agent_pos[1]+self.actions[action_index][2])
        new_state = self.get_state(new_pos)

        self.time_elapsed += 1
        end = False

        # If transition matrix says action is allowed, do action.
        if self.transition_matrix[self.state, action_index, new_state] == 1:
            self.grid[self.agent_pos] = 0
            self.agent_pos = new_pos
            self.grid[self.agent_pos] = 1

        # If agent has reached the door, terminate episode.
        if self.agent_pos == self.door_pos:
            end = True

        # If number of steps is over the time limit, end episode.
        if self.time_elapsed > self.time_limit:
            end = True
        self.state = self.get_state(self.agent_pos)

        return end, reward, self.state

    def reset(self):
        """
        Reset the grid to its original state, but with the agent in a new random␣
↪position.
        Maze and door remain unchanged.

        Returns the new state of the environment.
        """
        for cell in self.penalties_pos:
            self.grid[cell] = 3

        self.grid[self.door_pos] = 4

        self.grid[self.agent_pos] = 0
        self.agent_pos = self.get_rand_empty_cells(1)
        self.grid[self.agent_pos] = 1

        self.time_elapsed = 0

        self.state = self.get_state(self.agent_pos)

        return self.state
```

[ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[ ]:
```
class MazeDQN(Maze):

    def __init__(self):
        super().__init__(N=25)

    def return_start_state(self):
        '''Return initial state without having to reset the environment'''
        return self.preprocess_state()

    def step(self, action_index):
        '''Apply preprocessing to state and reward returned by the step function␣
↪in the Maze class'''
        end, reward, state = super().step(action_index)
        # Return state in appropriate format for DQN
        input_state = self.preprocess_state()
        # Return reward in appropriate format for DQN
        reward_tensor = torch.FloatTensor([float(reward)], device=device).
↪unsqueeze(0)
```

```
        return end, reward_tensor, input_state

    def preprocess_state(self):
        '''Preprocess state to be used as input for DQN'''
        # Calculate coordinates of the agent position relative to the door
        relative_coordinates = np.array([self.door_pos[0] - self.agent_pos[0],␣
↪self.door_pos[1] - self.agent_pos[1]])

        # Pad maze edge with 1s so that taking the surrounding cells of an edge␣
↪cell does not return an index error
        maze_padded = np.ones( (self.N + 2, self.N + 2), dtype = np.int8)
        maze_padded[1:self.N+1, 1:self.N+1] = self.grid[:,:]

        # Take surrounding cells using agent's position
        surroundings = maze_padded[ self.agent_pos[0] - 1: self.agent_pos[0] + 2,
                                    self.agent_pos[1] - 1: self.agent_pos[1] +␣
↪2]
        surroundings = surroundings.flatten()

        # Preprocess to acquire the state's DQN input format
        DQN_input_state = np.concatenate([relative_coordinates, surroundings])
        DQN_input_state = torch.FloatTensor(DQN_input_state, device=device).
↪unsqueeze(0)
        return DQN_input_state
```

B.0.2 Defining required models and functions

```
[ ]: Experience = namedtuple('Experience', ('state', 'action', 'reward',␣
      ↪'next_state'))
```

```
[ ]: # Inspiration for class from Lab 6
     class ExperienceReplayBuffer:

         def __init__(self, N_buffer):
             self.size_lim = N_buffer # Define size limit of buffer
             self.buffer = deque(maxlen=N_buffer) # Initialise buffer as a list

         def store(self, state_tensor, action, reward_tensor, next_state_tensor):
             '''Store an experience'''
             action_tensor = torch.tensor([action], device=device).unsqueeze(0) #␣
     ↪convert action to tensor
             experience = Experience(state_tensor, action_tensor, reward_tensor,␣
     ↪next_state_tensor) # wrap experience in namedTuple
             self.buffer.append(experience) # append to buffer

         def __len__(self):
```

```
        return len(self.buffer)

    def sample(self, BATCH_SIZE):
        '''Sample batch from the replay buffer'''
        return random.sample(self.buffer, BATCH_SIZE)

    def fill_replay_buffer(self):
        '''Fill replay buffer prior to DQN training'''
        buffer_filled = False
        while not buffer_filled:
            maze = MazeDQN() # define a new randomised maze environment each␣
 ↪episode
            state = maze.return_start_state() # return starting state
            end = False #
            while not end:
                # Obtain action from policy using q values acquired from␣
 ↪Q_policy_net
                action_index = policy(state, Q_policy_net)
                # Take the specified action in an emulator to acquire reward and␣
 ↪next state
                end, reward, next_state = maze.step(action_index)
                if end:
                    next_state = None # Set next_state to terminal state label␣
 ↪if the goal has been reached
                self.store(state, action_index, reward, next_state) # Store the␣
 ↪experience in the replay buffer
                state = next_state # Update the state
                buffer_filled = self.size_lim == len(self) # Check if the buffer␣
 ↪is full
        print('Buffer Filled')
```

[ ]:
```
# Class directly taken from Lab 6 code, already being appropriate for our␣
 ↪environment
class DQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):

        super().__init__()

        self.fc1 = nn.Linear(input_size, size_hidden)
        self.bn1 = nn.BatchNorm1d(size_hidden)

        self.fc2 = nn.Linear(size_hidden, size_hidden)
        self.bn2 = nn.BatchNorm1d(size_hidden)

        self.fc3 = nn.Linear(size_hidden, size_hidden)
        self.bn3 = nn.BatchNorm1d(size_hidden)

        self.fc4 = nn.Linear(size_hidden, output_size)


    def forward(self, x):
        h1 = F.relu(self.bn1(self.fc1(x.float())))
        h2 = F.relu(self.bn2(self.fc2(h1)))
        h3 = F.relu(self.bn3(self.fc3(h2)))
        output = self.fc4(h3.view(h3.size(0), -1))
        return output
```

[ ]:
```
# Inspiration from https://towardsdatascience.com/
 ↪dueling-deep-q-networks-81ffab672751
class DuelingDQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):
        super(DuelingDQN, self).__init__()
        self.input_size = input_size
        self.size_hidden = size_hidden
        self.output_size = output_size

        self.hidden_layer = nn.Sequential(
            nn.Linear(self.input_size, self.size_hidden),
            nn.ReLU(),
            nn.Linear(self.size_hidden, self.size_hidden),
            nn.ReLU())

        self.state_value_stream = nn.Sequential(
            nn.Linear(self.size_hidden, self.size_hidden),
            nn.ReLU(),
            nn.Linear(self.size_hidden, 1))

        self.advantage_stream = nn.Sequential(
            nn.Linear(self.size_hidden, self.size_hidden),
            nn.ReLU(),
            nn.Linear(self.size_hidden, self.output_size))


    def forward(self, x):
        hidden_values = self.hidden_layer(x)
        state_values = self.state_value_stream(hidden_values)
        advantages = self.advantage_stream(hidden_values)
        Q_values = state_values + (advantages - advantages.mean(dim=1).
 ↪unsqueeze(1))
        return Q_values
```

```
[ ]: class eGreedyDecayPolicy:

         def __init__(self, epsilon = 1, decay = 0.999, min_epsilon = 0.001):
             """
             This class defines an epsilon greedy policy with decay.
             epsilon - float, default 1
             decay - float, default 0.999

             In the case where decay = 1, epsilon will remain constant.
             In this case:
             If epsilon = 1, the policy will always choose an action randomly.
             If epsilon = 0, the policy will always choose greedily.
             """
             self.eps_current = epsilon
             self.eps_initial = epsilon
             self.min_eps = min_epsilon
             self.decay = decay

         def __call__(self, state, Q_policy_net):

             greedy = random.random() > self.eps_current

             if greedy: # if the policy takes the greedy action
                 with torch.no_grad(): # disable gradient computation
                     Q_policy_net.eval() # switch to evaluation mode

                     # Acquire action index from policy network
                     action_index = Q_policy_net(state).max(1)[1].view(1, 1).
       ↪numpy()[0][0]

                     Q_policy_net.train() # return to training mode

             else:
                 action_index = random.randint(0, 3) # return random action

             return action_index

         def update(self):
             self.eps_current = self.eps_current*self.decay # apply decay to epsilon
             if self.eps_current < self.min_eps:
                 self.eps_current = self.min_eps # if epsilon has gone below the min␣
       ↪epsilon value, set epsilon to the min value

         def reset(self):
             self.eps_current = self.eps_initial # reset epsilon
```

```
[ ]: # function adapted from code in Lab 6
     def optimize_DQN():

         # (state_tensor, action_tensor, reward_tensor, next_state_tensor)

         batch = replay_buffer.sample(BATCH_SIZE) # acquire batch
         batch = Experience(*zip(*batch)) # make batch experience variables available␣
      ↪by variable name
         state_batch = torch.cat(batch.state) # collect states
         action_batch = torch.cat(batch.action) # collect actions
         reward_batch = torch.cat(batch.reward) # collect rewards

         # Collect tensor of boolean values based on next state values: True if the␣
      ↪state is not terminal, False if it is
         non_final_next_states_mask = torch.tensor(tuple(map(lambda state: state is␣
      ↪not None, batch.next_state)), device=device, dtype=torch.bool)

         # Collect tensor of next state values for all next states that are not␣
      ↪terminal
         non_final_next_states = torch.cat([state for state in batch.next_state if␣
      ↪state is not None])

         Q_values = Q_policy_net(state_batch).gather(1, action_batch) # compute␣
      ↪Q-values through policy network

         next_Q_values = torch.zeros(BATCH_SIZE, device=device) # initialise next␣
      ↪state Q-values as tensor of zeros
         # Compute next state Q-values of non-terminal states using Q network, taking␣
      ↪the maximum Q-value for each input
         next_Q_values[non_final_next_states_mask] =␣
      ↪Q_policy_net(non_final_next_states).max(1)[0].detach()
         next_Q_values = next_Q_values.unsqueeze(1)

         # Compute target Q-values
         target_Q_values = reward_batch + (next_Q_values * GAMMA)

         # Compute Mean Squared Error Loss
         loss = F.mse_loss(Q_values, target_Q_values)

         optimizer.zero_grad() # set optimiser gradients to 0
         loss.backward() # calculate gradients through a backwards pass

         # Avoid gradient clipping
         for param in Q_policy_net.parameters():
             param.grad.data.clamp_(-1, 1)
```

```
    optimizer.step() # perform parameter update based on stored gradients

    return loss
```

[ ]:
```
# function adapted from code in Lab 6
def train_DQN():
    policy.reset() # reset policy hyperparameters
    replay_buffer.fill_replay_buffer() # fill buffer
    episode_rewards = []
    for episode in range(NUM_EPISODES):
        maze = MazeDQN() # new maze every episode
        state = maze.return_start_state() # acquire first state of the episode
        end = False # reset episode termination variable
        episode_reward = 0 # reset episode reward

        while not end:
            # Acquire action index based on q-values from the policy network,␣
 ↪using specified policy
            action_index = policy(state, Q_policy_net)

            end, reward, next_state = maze.step(action_index) # take the action␣
 ↪in an emulator

            if end: # if the agent has reached the exit, set the next state to␣
 ↪terminal state label
                next_state = None

            replay_buffer.store(state, action_index, reward, next_state) # store␣
 ↪the transition in memory
            state = next_state # update the state
            episode_reward += float(reward) # add step reward to episode reward

            # Run a single batch through Double DQN and update the model
            loss = optimize_DQN()

        episode_rewards.append(float(episode_reward)) # append episode reward to␣
 ↪list of episode rewards
        policy.update() # update policy hyperparameters

        if episode % 100 == 0:

            print('Episode {}: reward : {} epsilon: {} loss: {}'.format(episode,␣
 ↪episode_reward,
                  policy.eps_current, np.round(loss.detach().numpy(), 3)))
            print('Average of reward for last 100 episodes: {}'.
 ↪format(sum(episode_rewards[-100:])/100))
```

```
    print('Model trained')
    return episode_rewards
```

[ ]:
```
# function adapted from code in Lab 6
def optimize_doubleDQN():

    # (state_tensor, action_tensor, reward_tensor, next_state_tensor)

    batch = replay_buffer.sample(BATCH_SIZE) # acquire batch
    batch = Experience(*zip(*batch)) # make batch experience variables available␣
 ↪by variable name
    state_batch = torch.cat(batch.state) # collect states
    action_batch = torch.cat(batch.action) # collect actions
    reward_batch = torch.cat(batch.reward) # collect rewards

    # Collect tensor of boolean values based on next state values: True if the␣
 ↪state is not terminal, False if it is
    non_final_next_states_mask = torch.tensor(tuple(map(lambda state: state is␣
 ↪not None, batch.next_state)), device=device, dtype=torch.bool)

    # Collect tensor of next state values for all next states that are not␣
 ↪terminal
    non_final_next_states = torch.cat([state for state in batch.next_state if␣
 ↪state is not None])

    Q_values = Q_policy_net(state_batch).gather(1, action_batch) # compute␣
 ↪q-values through policy network

    next_Q_values = torch.zeros(BATCH_SIZE, device=device) # initialise next␣
 ↪state q-values as tensor of zeros
    # Compute next state q-values of non-terminal states using target Q network,␣
 ↪taking the maximum q-value for each input
    next_Q_values[non_final_next_states_mask] =␣
 ↪Q_target_net(non_final_next_states).max(1)[0].detach()
    next_Q_values = next_Q_values.unsqueeze(1)

    # Compute target q-values
    target_Q_values = reward_batch + (next_Q_values * GAMMA)

    # Compute Loss
    loss = F.mse_loss(Q_values, target_Q_values)

    optimizer.zero_grad() # set optimiser gradients to 0
    loss.backward() # calculate gradients through a backwards pass

    # Avoid gradient clipping
    for param in Q_policy_net.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step() # perform parameter update based on stored gradients

    return loss
```

```
[ ]: # function adapted from code in Lab 6
def train_doubleDQN():
    policy.reset() # reset policy hyperparameters
    replay_buffer.fill_replay_buffer() # fill buffer
    episode_rewards = []
    for episode in range(NUM_EPISODES):
        maze = MazeDQN() # new maze every episode
        state = maze.return_start_state() # acquire first state of the episode
        end = False # reset episode termination variable
        episode_reward = 0 # reset episode reward

        while not end:
            # Acquire action index based on q-values from the policy network,␣
 ↪using specified policy
            action_index = policy(state, Q_policy_net)

            end, reward, next_state = maze.step(action_index) # take the action␣
 ↪in an emulator

            if end: # if the agent has reached the exit, set the next state to␣
 ↪terminal state label
                next_state = None

            replay_buffer.store(state, action_index, reward, next_state) # store␣
 ↪the transition in memory
            state = next_state # update the state
            episode_reward += float(reward) # add step reward to episode reward

            # Run a single batch through Double DQN and update the model
            loss = optimize_doubleDQN()

        episode_rewards.append(float(episode_reward)) # append episode reward to␣
 ↪list of episode rewards
        policy.update() # update policy hyperparameters

        # Update the target network, copying all weights and biases in DQN
        if episode % TARGET_UPDATE_FREQ == 0:
            Q_target_net.load_state_dict(Q_policy_net.state_dict())

        if episode % 100 == 0:
            print('Episode {}: reward : {} epsilon: {} loss: {}'.format(episode,␣
↪episode_reward,
                  policy.eps_current, np.round(loss.detach().numpy(), 3)))
            print('Average of reward for last 100 episodes: {}'.
↪format(sum(episode_rewards[-100:])/100))
    print('Model trained')
    return episode_rewards
```

B.0.3 Running the models

DQN

[ ]:
```
# Defining Q-network and optimiser parameters
INPUT_SIZE = 3*3 + 2
HIDDEN_SIZE = 128
NUM_ACTIONS = 4
ALPHA = 0.01

# Define policy Q-network and optimiser
Q_policy_net = DQN(INPUT_SIZE, HIDDEN_SIZE, NUM_ACTIONS).to(device)
optimizer = optim.SGD(Q_policy_net.parameters(), lr=ALPHA)
```

[ ]:
```
# Define DQN training hyperparameters
BUFFER_SIZE = 10000
BATCH_SIZE = 256
GAMMA = 0.5 # Discount
MAZE_SIZE = 25
NUM_EPISODES = 5000
```

[ ]:
```
# Define policy
policy = eGreedyDecayPolicy()
# Define buffer
replay_buffer = ExperienceReplayBuffer(BUFFER_SIZE)
# Train DQN
DQN_episode_rewards = train_DQN()
```

[ ]:
```
DQN_episode_rewards = np.asarray(DQN_episode_rewards)
np.savetxt('vanilla_DQN_results.csv', DQN_episode_rewards, delimiter=',')
```

Double DQN

[ ]:
```
# Defining Q-network and optimiser parameters
INPUT_SIZE = 3*3 + 2
HIDDEN_SIZE = 128
NUM_ACTIONS = 4
ALPHA = 0.01

# Define policy Q-network and optimiser
```

```
Q_policy_net = DQN(INPUT_SIZE, HIDDEN_SIZE, NUM_ACTIONS).to(device)
optimizer = optim.SGD(Q_policy_net.parameters(), lr=ALPHA)

# Define target Q-network
Q_target_net = DQN(INPUT_SIZE, HIDDEN_SIZE, NUM_ACTIONS).to(device)
# Copy parameter values from policy Q-network
Q_target_net.load_state_dict(Q_policy_net.state_dict())
```

[ ]:
```
# Define DQN training hyperparameters
BUFFER_SIZE = 10000
BATCH_SIZE = 256
TARGET_UPDATE_FREQ = 100 # Number of episodes per update of the target DQN␣
 ↪parameters
GAMMA = 0.5 # Discount
MAZE_SIZE = 25
NUM_EPISODES = 5000
```

[ ]:
```
# Define policy
policy = eGreedyDecayPolicy()
# Define buffer
replay_buffer = ExperienceReplayBuffer(BUFFER_SIZE)
# Train DQN
DDQN_episode_rewards = train_doubleDQN()
```

[ ]:
```
DDQN_episode_rewards = np.asarray(DDQN_episode_rewards)
np.savetxt('double_DQN_results.csv', DDQN_episode_rewards, delimiter=',')
```

Dueling Double DQN

[ ]:
```
# Defining Q-network and optimiser parameters
INPUT_SIZE = 3*3 + 2
HIDDEN_SIZE = 128
NUM_ACTIONS = 4
ALPHA = 0.01

# Define policy Q-network and optimiser
Q_policy_net = DuelingDQN(INPUT_SIZE, HIDDEN_SIZE, NUM_ACTIONS).to(device)
optimizer = optim.SGD(Q_policy_net.parameters(), lr=ALPHA)

# Define target Q-network
Q_target_net = DuelingDQN(INPUT_SIZE, HIDDEN_SIZE, NUM_ACTIONS).to(device)
# Copy parameter values from policy Q-network
Q_target_net.load_state_dict(Q_policy_net.state_dict())
```

[ ]:
```
# Define DQN training hyperparameters
BUFFER_SIZE = 10000
BATCH_SIZE = 256
```

```
TARGET_UPDATE_FREQ = 100 # Number of episodes per update of the target DQN␣
 ↪parameters
GAMMA = 0.5 # Discount
MAZE_SIZE = 25
NUM_EPISODES = 3000
```

[ ]:
```
# Define policy
policy = eGreedyDecayPolicy()
# Define buffer
replay_buffer = ExperienceReplayBuffer(BUFFER_SIZE)
# Train DQN with prioritised experience replay
DDDQN_episode_rewards = train_doubleDQN()
```

[ ]:
```
DDDQN_episode_rewards = np.asarray(DDDQN_episode_rewards)
np.savetxt('DDDQN_results.csv', DDDQN_episode_rewards, delimiter=',')
```

C RLLib

[ ]:
```
!pip install -U "ray[rllib] == 1.6"
!pip install -U "ray[tune]"
!pip install gym[atari,accept-rom-license]==0.21.0
```

[ ]:
```
import gym
import numpy as np
import ray
from ray import tune
import ray.rllib.agents.a3c as a3c
```

[ ]:
```
config = a3c.DEFAULT_CONFIG.copy()
config['env'] = 'Pong-ram-v0'
config['num_gpus'] = 0
config['num_workers'] = 7
config['num_envs_per_worker'] = 5
config['framework'] = 'tf'
config['rollout_fragment_length'] = 20 # increase rollout batch size
```

[ ]:
```
analysis = tune.run('A3C',
                    config = {
        # Works for both torch and tf.
        'env': 'Pong-ram-v0',
        'num_gpus': 0,
        'framework': 'tf',
        'num_workers': 7,
        'rollout_fragment_length': 20,
        "num_envs_per_worker": 5,
        "lambda": 0.95})
```

[ ]:
```
%load_ext tensorboard
%tensorboard --logdir /root/ray_results/
```