CITY UNIVERSITY
LONDON

INM707:
Reinforcement Learning

Víctor Abia Alonso & Julian Jimenez Nimmo
student IDs: 230061642 & 230066319

May 12, 2024

Repository

Contents

1 Environment and Problem 1

2 Transition and reward functions. 1

3 Q-learning parameters. 2

4 Different policies. 3

5 Different reward structures. 5

6 Conclusion of Q-learning 6

7 Vanilla DQN and improvements. 6

7.1 Environment and problem. 6
7.2 Vanilla DQN 6
7.3 Double DQN 6
7.4 Prioritized Experience Replay 7
7.5 Implementation 7

8 Analyse the results quantitatively and qualitatively. 8

9 Individual Component: Atlantis 9

10 Analysis of the results. 9

11 Contributions 11

A Appendix: Q values map 13

B Appendix: Task 2 Hyperparamaters 14

C Appendix: Breakout strategy 14

D Code 15

D.1 Task 1 15
D.1.1 Maze_env.py 15
D.1.2 agent.py 17
D.2 Task 2 code 21
D.2.1 buffer.py 21
D.2.2 dqn.py 22
D.2.3 dqn_inference.py 27
D.2.4 Individual Part 29

BASIC TASKS

1 Environment and Problem

The environment to be solved by the Q-learning agent [1] is a variation of a maze layout. In a traditional maze environment, there are only walls and paths. In our environment, walls now represent fire, there exists a special state granting an additional reward and an exit state granting a final reward which terminates the game. We wish to model a situation where the goal of a robot is to save a person and reach the exit during a house fire in the shortest time possible.

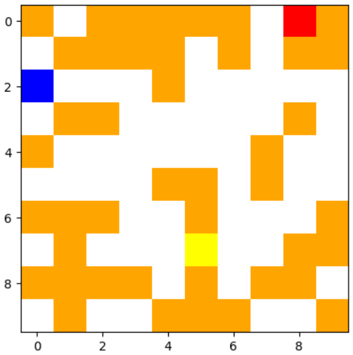


Figure 1: Fire maze simulation: the starting state is the blue cell, the person is located in the yellow cell,, and the exit is the red state, with fires represented by orange cells.

The maze is represented as a 10x10 grid. The starting state is cell (2,0), the person is located at cell (5,5) and the exit is located at cell (8,0). Other variations of the maze can be constructed, both through randomization or manually changing what is present at each cell. Our aim is to construct an agent which prioritizes retrieving the person and then reaching the exit rather than only heading towards the exit.

2 Transition and reward functions.

In reinforcement learning, an agent learns through interactions with an environment. The environment defines which actions are allowed for each state and provides a reward signal for each action, which the agent uses to learn. For our problem, we encapsulate rewards in a 3 dimensional matrix of shape (10, 10, 2) where the first two dimensions represent the position of the robot within the grid (xy coordinates) and the third dimension is a binary variable representing whether the person has been collected or not. Therefore, there are 200 possible states [1] the agent can find themselves in. For each state, in general, there are four available actions: [up, down, left, right], which make the agent transition from its current cell to the corresponding adjacent one. For edge cells and corners, out-of-bound actions make the agent stay in the grid. Also, if the person hasn't been collected yet, the actions that lead to cell (7, 5) change the last dimension of the matrix from 0 to 1. These descriptions are implemented through conditional statements in the **transition function** of the environment (`.transition_R()` method in the `maze_env.py` file). This environment is deterministic as every action unequivocally leads to a specific state at any state and time.

Additionally, the environment also provides a reward for every state and action which, as the transition function is deterministic, is in fact equivalent to receiving a reward for every next state. These rewards are encapsulated in the reward matrix R, which is a 4 dimensional array of size (10, 10, 2, 4) providing a reward for every action (4 possibilities) taken in each state (10x10x2). The unacceptable actions are conceived as NaN -Not a Number- values on the matrix. The Q matrix the agents uses to value a state-action pair has the same dimensions as this R matrix.

In order to explore different environment structures that would lead the agent to learn to collect the person and then reach the exit, we consider two different environments each with different reward and transition structures. Both structures summarize the objective we defined for the agent, and vary in the the implications of getting into a fire cell.

[1] In fact, there are just 199 because the robot cannot be in the location of the person unless it has been collected.

Action	Limited movement	Terminal movement
Getting to the exit.	1*	1*
Collecting the person	10	10
Taking a step to a non-fire cell	-0.005	-0.005
Taking a step into a fire cell	None	-1*

Table 1: Description of rewards in the two types of environments.

The asterisk (*) on the table means that the action terminates the episode. The choice of rewards for collecting the person and reaching the exit prioritizes collecting the person and provides extra reward for the exit. Safe steps are still characterized by some small negative reward in order to encourage the robot to finish the episodes quicker, which implicitly incentives them to get to the exit despite its relatively small reward. For actions that lead to the fire, the *limited* environment doesn't allow them at all -NaN value in the R matrix- while in the *terminal* environment it yields a significant negative reward and terminates the episode. These two ways of conceptualizing problematic states in environments are common in games, for example, either by giving negative reward (subtracting life points) or by directly prohibiting the dangerous actions in the first place. Therefore, both environments encourage the desired task of collecting the person and then exiting.

3 Q-learning parameters.

Q learning formula: $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$

The Bellman equation has two parameters alpha (α) and gamma (γ), which are not defined by the environment, and the choice of these parameters can thus influence the performance and effectiveness of the agent. While finding the optimal parameters, it is important to show the evolution of rewards over time, as it showcases how significant each parameter was for convergence. The number of steps per episode is not an appropriate metric as terminating early, either by getting to the target without the coin or by dying in the fire (in the case of terminal environments) does not reflect the optimal performance.

The γ parameter is the discount factor and reflects how much the agent values next step rewards compared to immediate rewards; it takes values between 0 and 1. It applies a geometric accumulation to future rewards; for instance, $\gamma = 0.1$ diminishes the value of rewards two steps away to just 0.01. Low γ values render the agent myopic, especially in environments with rapid state changes where the value of future rewards decreases exponentially. Conversely, setting γ to 1 is typically avoided because it theoretically eliminates the urgency for the agent to acquire rewards promptly. In order to find the optimal γ for our environment we try different values and analyse convergence and rewards acrosss episodes.

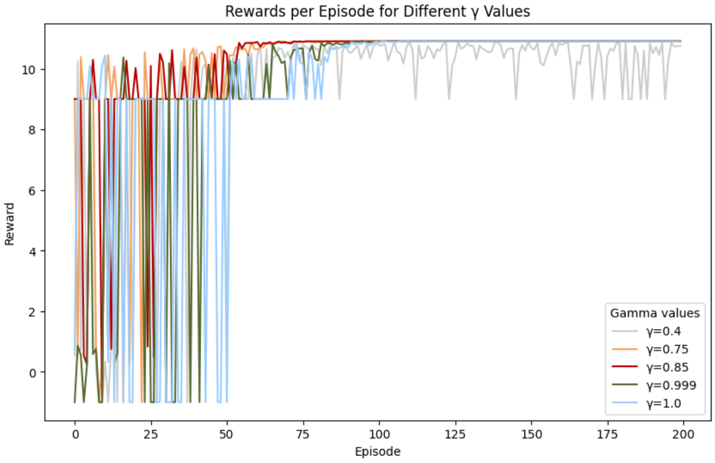


Figure 2: Comparison of different gamma values. 200 episodes with softmax policy with $T_{min} = 0.001$, $T_0 = 50$ and $\lambda = 0.6$ in a limited reward environment. $\alpha = 1$.

Low values of γ, like 0.4, does not allow the agent to retrieve the person, and thus not obtain the optimal reward. It considers this strategy too time consuming, preferring the immediate reward of the exit. Higher values of γ seem to all converge to the optimal path, with 0.85 converging the fastest in around 60 episodes.

The α parameter is the learning rate of the update of the Q matrix at each step taking values between 0 and 1. An α value close to 1 means that the agent relies mostly on recent information (i.e. the last updates of that state-action pair) which may lead to unstable learning by underutilizing past information, although training can be quicker. An α close to 0 means Q values are more conservative, leading to more stable, but slower training as it incorporates information more gradually. In order to find the optimal value of α, different values should be tried to analyse convergence.

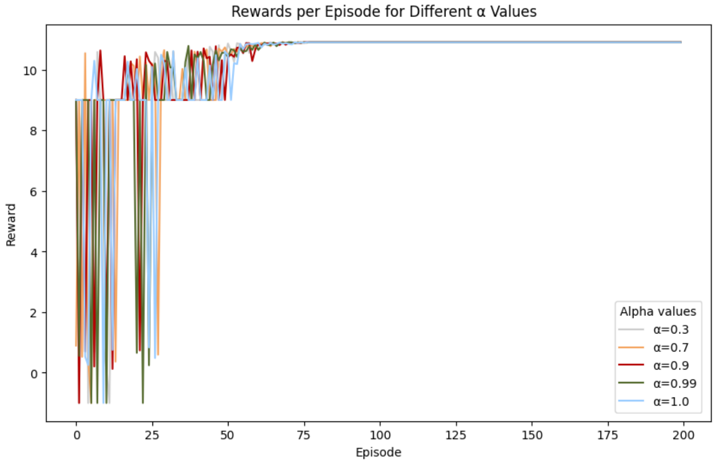


Figure 3: Comparison of different alpha values. 200 episodes with softmax policy with $T_{min} = 0.001$, $T_0 = 50$ and $\lambda = 0.6$ in a limited reward environment. $\gamma = 0.85$.

In this case, it seems like every value of α greater than 0.3 converges quite similarly, meaning that the dynamics of the environment in this specific setup seem to be indifferent to the update rate as learning will happen naturally over the epochs. However, this is not a guarantee that training is indifferent to α in every setup and its performance should be analysed case by case. We will choose $\alpha = 0.9$ from now on.

Note that almost from the beginning the agent is already getting high rewards, implying that the agent got the person. This is because it is the limited movement environment in which an agent never dies with the fire and it has 200 steps per episode to learn where the person and the exit are, which allows fast training (see more on the reward section). These results encouraged us to try the terminal reward structure, in which stepping into a fire kills you, and to tune the parameters of policies and compare them in this harder variant of the environment.

4 Different policies.

In reinforcement learning, agents choose which actions to take using a policy, which is a function mapping states to actions. Agents already have an internal map of the environment, in this case the Q matrix, which gives to every action in each state a value which aligns with the expected reward. As such, policies are a way of incorporating this knowledge from the Q-learning function into the actions the agents take. A key aspect is that the Q matrix is suboptimal at the beginning and requires exploration of the environment to be refined. Following the greedy policy of taking the action that maximizes rewards at the beginning may not give the best reward overall. This motivates the pursuit of an optimal exploration-exploitation ratio in which policies allow for sufficient exploration of the environment before focusing on choosing actions that maximize the expected reward. The two policies considered, epsilon-greedy and softmax, will encapsulate this trade-off with an exponential decay parameter λ for which different values are considered.

The epsilon greedy policy chooses a random action with probability epsilon (ϵ), and follows the greedy policy (choosing the action with the highest expected reward according to the Q function) with probability 1 - ϵ. Thus, ϵ encapsulates the exploration and should be higher at the beginning and lower at the end. We choose $\epsilon = 1$ at the beginning. For the minimum ϵ (at the end) we first consider $\epsilon = 0.05$. However, this lead to degrading performance as even when the optimal path was identified, it ”cannot” follow it as sometimes it chooses actions randomly. In this case, the optimal path is 20 steps long, meaning that the agent has 64% chance ($= 1 - 0.95^{20}$) of taking a suboptimal step at some point in the path, which in the terminal environment frequently means death. Given the simple nature of our environment, and after monitoring training, we chose the minimum epsilon to be 0. The exponential decay rate describes the value of ϵ in each epoch, which decays from $\epsilon_{max} = 1$ in the first epoch to $\epsilon_{min} = 0$ after infinite epochs following the formula: $\epsilon_n = \epsilon_{min} - (\epsilon_{max} - \epsilon_{min}) \times e^{-\lambda n}$ where n indicates the epoch.

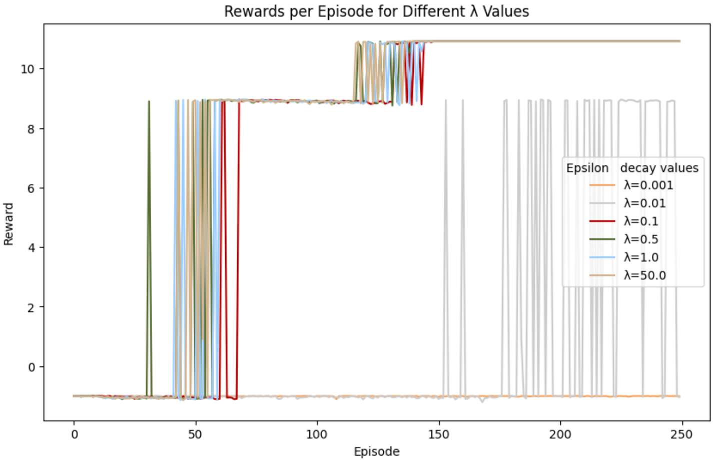


Figure 4: Comparison of different epsilon decay values on the epsilon greedy policy performance. 250 episodes. Terminal rewards. $\gamma = 0.85$. $\alpha = 1$. $\epsilon_{min} = 0$. $\epsilon_{max} = 1$.

The parameter λ control the rate of decay, and different values of λ are evaluated. The exponential decay rate λ represents this exploration-exploitation tradeoff with a higher λ meaning the agent explores less than with a lower λ. Figure 4 shows that values of λ smaller than 0.01 cannot get the exit after the person in 250 episodes, and values smaller than 0.001 do not even reach the exit, making very small values of λ quite inefficient. Bigger values of λ all successfully converge to the optimal strategy roughly in less than 150 episodes, with the very large value of $\lambda = 50$ showing slightly better performance. This is interesting as if $\lambda = 50$, after one episode, the value of epsilon is $\epsilon_1 = 0 + (1 - 0) \times e^{-50 \times 1} = e^{-50} \approx 1.9 \times 10^{-19} \approx 0$. This means that after an initial random step, the agent is always following a pure greedy policy with no exploration but still performing optimally. The reason for this lies in the simple nature of the environment which, without intrinsic exploration in the policy, allows the agent to learn solely by trial and error. With this result we discovered that, for some parameters of α and γ, the environment is designed in a way that always guides the agent naturally towards the optimal path.

The softmax policy is an intrinsically stochastic policy which gives a probability for each action weighing their expected reward, giving non-zero chance for all actions to occur. The formula is

$$\pi(a) = \frac{e^{\frac{Q(a)}{T}}}{\sum_b e^{\frac{Q(b)}{T}}}$$

where $T \in [0, \infty)$ is the temperature which controls the action selection sampling, a is the action considered and b are all the possible actions. A high temperature means all actions are nearly equally likely to be chosen, thus prioritizing exploration. A low temperature prioritizes sharply the actions with highest expected reward and the agent is closer to greedy, thus prioritizing exploitation.

The decay of temperature T is calculated in the same manner as the decay of epsilon. As before, it follows an exponential decay from $T_{max} = 5$ in the first epoch to $T_{min} = 0$ after infinite epochs. We tried different values of the decay parameter λ for temperature. The results align with the analysis done for the epsilon decay parameter above, where λ=0.01 does not reach the exit, $\lambda = 0.1$ reaches the person but not the exit, and higher values all converge in less than 200 episodes with the highest value $\lambda = 50$ showing the best performance.

In this environment, we can use high decay rates without problem as exploration is not necessary. We set decay parameters for both policy types to 1. The parameters should be finetuned for any specific environment to get optimal performance.

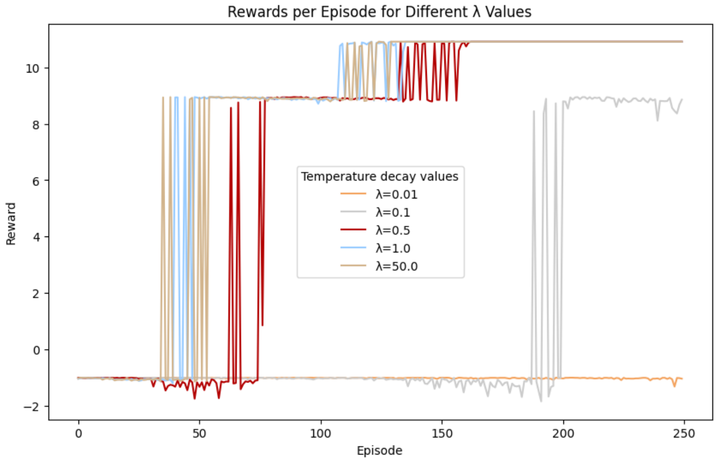
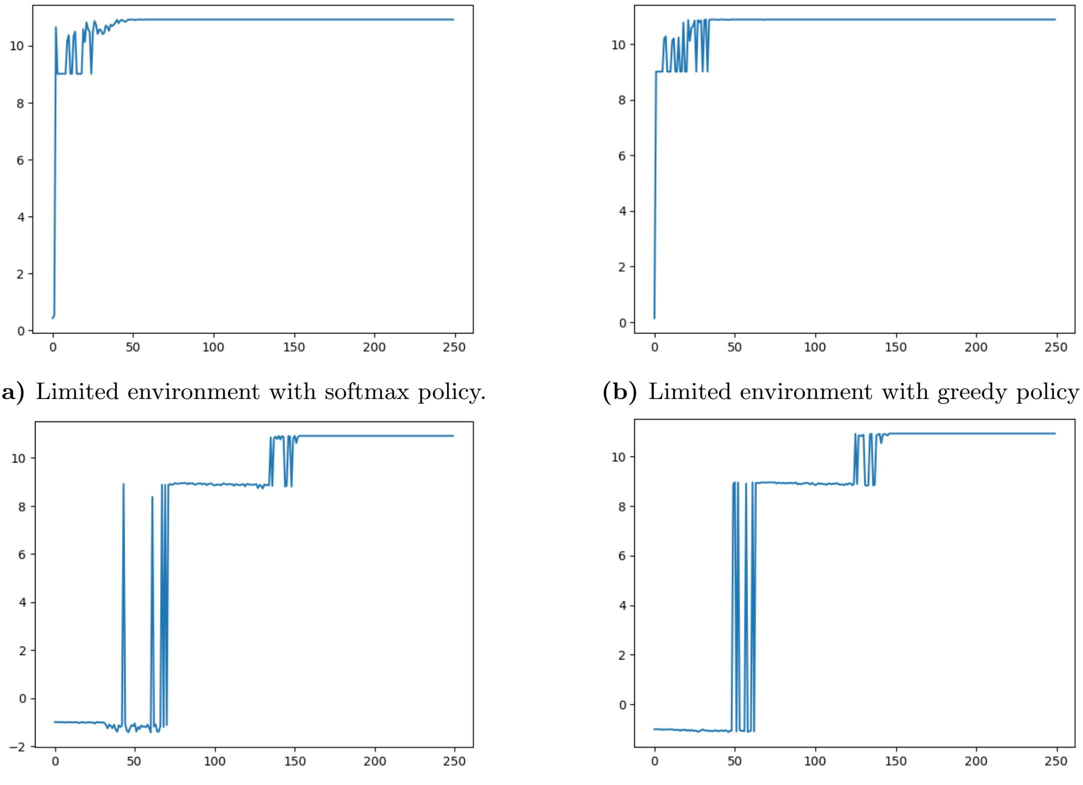


Figure 5: Comparison of different temperature decay values on the epsilon softmax policy performance. 250 episodes. Terminal rewards. $\gamma = 0.85$. $\alpha = 1$. $T_{min} = 0.001$. $T_{max} = 5$.

5 Different reward structures.



(a) Limited environment with softmax policy.

(b) Limited environment with greedy policy

(c) Terminal environment with softmax policy.

(d) Terminal environment with greedy policy.

Figure 6: Comparative evolution rewards over 250 episodes of the two policies in the two environments. $\gamma = 0.85$. $\alpha = 1$. $\epsilon_{min} = 0$. $\epsilon_{max} = 1$. $T_{min} = 0.001$. $T_{max} = 5$. $\lambda_T = \lambda_\epsilon = 1$.

After evaluating different parameters of the Q-learning agent and two different policies, we now explore its behaviour in the two different reward-transition structures of the environment. In the terminal environment -where the agent dies and is penalized every time it goes to a fire state-, as well as in the limited environment -where the agent cannot attempt to go towards the fire- the agent successfully learns to get the person and then go to the exit. If we look at the rewards per episode (Figure 12), the performance of the agent in a limited environment converges earlier to the optimal path (around 50 episodes) compared to the terminal environment where it takes three times longer (around 160 episodes).

This is a consequence of each specific design. At the beginning, the episodes on the terminal environment are very short with the agent dying often. In contrast, the limited environment episodes only can end when the exit is reached. They have a maximum timestep of 200, which allows the agent to get significant exploration from the beginning, thus learning "faster" in terms of rewards per episode. In terms of timesteps to solve the environment they are much closer with limited environment still outperforming, likely because the agent does not have reset the environment whenever a fire is encountered.

6 Conclusion of Q-learning

The results from the experiments show the design of these two environments successfully lead the agent to learn the intended objective, retrieve the target and each the exit. Both the small negative reward for each step by default and the discount factor gamma, pushed the agent to actually perform this task faster in the least amount of timesteps possible. Both policies where successful in making the agent learn. The finetuning of the decay rate -which showed that a purely greedy policy was enough to find the optimal path under certain settings- highlights the efficiency of the Bellman Equation for this environment. By updating the values associated to each state-action pair, the agent creates a meaningful representation of the rewards in the environment, that enables them to follow and find the optimal solution. There is a detailed map of the Q values the agent learned for each state in Appendix A.

7 Vanilla DQN and improvements.

7.1 Environment and problem.

A challenging environment which cannot be solved feasibly using tabular linear methods such as Q-Learning was chosen for this task. Atari environments with large state spaces are ideal, with extensive research applied to solve this environment. Tabular methods would crash from the state-space memory requirements, while deep learning approaches require a representative subsample to solve it.

Breakout is an Atari game [2] in which the agent with 5 lives must destroy blocks using a bouncing ball and moving a paddle. There are 6 different rows to destroy, with each pair of rows (1-2, 3-4, 5-6) formed by blocks with points 1, 4 and 7 respectively. There are 4 actions available: NOOP (no movement), FIRE, LEFT, RIGHT. A state in the environment is represented as an RGB frame with dimensions 210x160x3. A terminal state occurs when a player has lost all their lives. A combination of lives, ball position, paddle position and brick configuration leads to state space of over 10^{12} states [3].

7.2 Vanilla DQN

DQN [4] is an extension of the Q-learning algorithm , using Neural Networks (NN) to approximate the state-value function. The network learns by minimizing the loss between the estimated Q-values predicted by the NN and the ground truth Q-values calculated using the Bellman equation and the NN. This loss is propagated through the network and its weights are updated iteratively.

Like Q-Learning algorithm, DQN is an off-policy algorithm, allowing the algorithm to use samples belonging to different policies. Similarly to Q-Learning, exploration-exploitation is important to learn the environment and how to solve the objective. Improvements over this naive implementation of DQN were added [5], improving performance drastically. These improvements include:

1. Experience replay buffer which trains the agent using representative samples of recent past experiences, to decorrelate the agent's experiences and stabilize training.
2. Target network which get temporal difference target values in order for updates to move towards stationary target values, stabilizing training and reducing divergence of the policy.

The pseudocode for the DQN implementation can be seen in Algorithm 1, where γ is the discount factor and L the loss function (Mean Square Error):

7.3 Double DQN

An issue from DQN is the overestimation of the target Q-values. This occurs as bias may be introduced in the value calculations if the target network produces an error in its estimations of $\max_{a'} \hat{q}_2(s', a', \theta_2)$.

Algorithm 1 DQN algorithm

```
Initialise Experience replay memory $M$ to capacity $N$
Initialise policy network $\hat{q}_1$ with parameters $\theta_1 \in \mathbb{R}^d$ arbitrarily
Initialise target action-value network $\hat{q}_2$ with parameters $\theta_2 = \theta_1$
for each episode do
    Initialise $S$
    Choose action $A$ in state $S$ using policy derived from $\hat{q}_1(S, \cdot, \theta_1)$
    Take action $A$, observe reward $R$ and next-state $S'$
    Store transition $(S, A, R, S')$ in $M$
    for transition $(S_j, A_j, R_j, S'_j)$ in minibatch sampled from $D$ do
        $y = \begin{cases} R_j(s,a) & \text{if } s' \text{ is terminal} \\ R_j(s,a) + \gamma \hat{q}_2(s', \max_{a'} \hat{q}_2(s', a', \theta_2)) & \text{otherwise} \end{cases}$
        $\hat{y} \leftarrow \hat{q}_1(S_j, A_j, \theta_1)$
        Perform gradient descent step $\nabla_{\theta_1} L(y, \hat{y})$
    end for
    Every $C$ time-steps, update $\theta_2 = \theta_1$
end for
```

As we are maximizing over Q-values, an overestimated choice will always be preferred.

Double-DQN (DDQN) [6] mitigates this issue by introducing the policy network for action selection, removing positive bias and decoupling action selection from value estimation. This should lead to more stable convergence and improved policy learning.

$$\text{Target value } y = \begin{cases} R(s,a) & \text{if } s' \text{ is terminal} \\ R(s,a) + \gamma \hat{q}_2(s', \arg\max_{a'} \hat{q}_1(s', a')) & \text{otherwise} \end{cases}$$

7.4 Prioritized Experience Replay

Random sampling from a list of experiences is not the most optimal way of learning, as all experiences have equal probability of being sampled, regardless of their contribution to learning. Some experiences are more important than others, and should be sampled more frequently. Prioritized experience replay (PER) [7] is an improvement over regular experience buffers, sampling more often valuable experiences. The value of an experience is measured using the temporal difference error, in our case, between target and estimated Q-values.

This non-uniform sampling can add bias if not regulated, as samples do not belong to the same distribution as the expectation. Weights added to the backpropagated loss are calculated from the probabilities $P(i)$ to correct this bias, with an additional hyperparameter β controlling the compensation against bias. These weights are normalized for stability purposes, as well as ensuring weights can only decrease the update downwards. The probabilities $P(i)$ are calculated from the stored priorities p_i and hyperparameter α controls the prioritization used. Probabilities are added a small $\epsilon = 0.01$ constant to avoid probabilities with value 0. N corresponds to the sample batch size.

$$Weights(i) = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta} \quad \text{and} \quad Probabilities(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}}$$

As additional computation is required for sampling and weight calculations, an efficient data structure is necessary. To reduce sampling and update complexity, priorities are stores in a sum-tree data structure, offering a complexity of $O(logN)$, unlike lists with a complexity of $O(N)$.

7.5 Implementation

To reduce the state space complexity without sacrificing learning, the observations were converted to grayscale, as the colours of the environment do not contribute to learning the optimal policy. Furthermore, the image resolution was reduced to 84x84, reducing the state space without sacrificing functionality.

Lastly, frame stacking was introduced to allow the network to understand direction and velocity of the ball, joining 4 frames into a single frame. Additionally, to improve learning, all rewards were considered of equal value to encourage the model to remove all blocks rather than maximize rewards. To increase training speed, auto-shoot was implemented when starting a new episode or when starting a new life.

The architecture for the neural network consisted of 3 convolutional layers and 2 fully connected linear layers, with ReLu activation functions between each layer, based on [5]. An epsilon-greedy policy was chosen, linearly decaying ϵ during a number of frames. The sum-tree data structure was constructed following the code provided in [8]. Our expectation based on previous literature, is to observe the best performance in DQN rather than DDQN, with prioritized experience replay alongside DDQN between both results. DDQN can mitigate the overestimation of some states, while PER should select samples which are more informative to the learning process. Interestingly, DQN with this implementation of PER was not added to benchmarks, and could provide an improvement over DQN.

8 Analyse the results quantitatively and qualitatively.

The vast number of hyperparameters can greatly affect the performance of the algorithms. As a competitive starting point, the hyperparameters based on the DQN paper were used on most experiments, with the specific hyperparameters in PER based on values suggested by the PER paper. An additional experiment using slightly tuned hyperparameters for DDQN based on the DDQN paper was also evaluated. The different set of hyperparameters are detailed in Appendix B. All experiments were trained up to 4 million frames for fair comparisons.

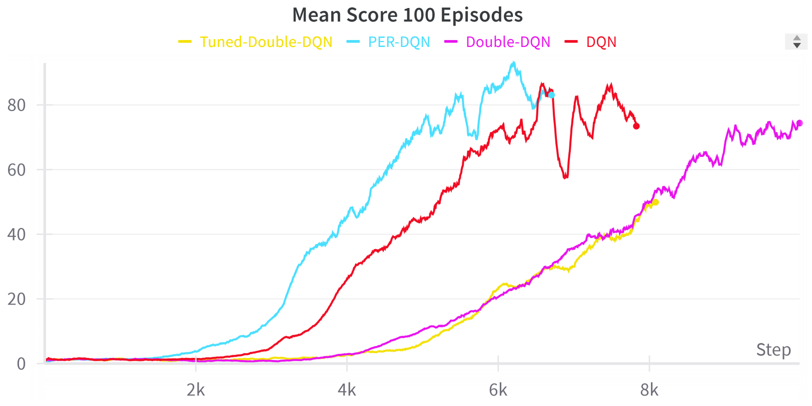


Figure 7: Mean rewards over 100 episodes for DQN, DDQN, Tuned DDQN and DQN with PER. Note the different lengths correspond to the number of episodes performed by each configuration (except Tuned-DDQN which was stopped early as no improvements were noted)

To evaluate the performance during training, the mean reward over the most recent 100 episodes and the scores per episodes were plotted. Note this evaluation considers all rewards equally. A checkpoint for a model is saved when a best mean reward is achieved. These checkpoints are then evaluated, choosing only greedy actions during inference, and a video of the agent interacting with environment is stored.

DQN with PER achieves the best results, reaching higher rewards in less episodes. DQN followed these results, achieving similar results but requiring more episodes. DDQN and its tuned equivalent achieved similar results by the end but at much slower rate of learning. Curiously, no difference was noted when training DDQN on tuned parameters and DQN hyperparameters, as seen in Figure 7. In terms of time complexity, DQN with PER encumbered a significant cost, almost doubling training time compared to other models (6 hours vs 3-4 hours on Hyperion).

Overall, all models achieved similar rewards, although higher rewards in certain episodes are achieved only in DQN and DQN with PER, as seen in Figure 13. During inference (rewards now have different values), DDQN achieves a best score of 356, DQN a score of 366 and DQN with PER a score of 407. During inference some models struggled to achieve improved results as they got stuck in a loop, with the ball bouncing between a gap of bricks and back to the paddle, repeated continuously in a cycle.

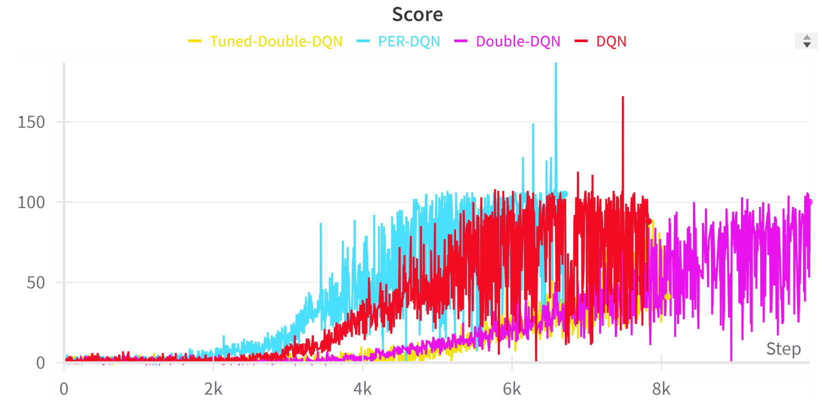


Figure 8: Rewards per episode for DQN, Double-DQN, Tuned Double-DQN and DQN with PER.

An interesting observation is the development of strategies. All models developed a tendency to create a hole reaching the ceiling, allowing the ball to repeatedly bounce between the ceiling and the top rows, achieving points rapidly. This long term goal is encouraged by the γ value of 0.99. Another strategy to reach the ball on average faster is to move to the middle of the screen, allowing the agent to move to either directions at a constant speed. Examples of the strategies can be found in the videos uploaded alongside this report and the first strategy can be found in Appendix C.

9 Individual Component: Atlantis

For my individual part of the coursework, I decided to implement and analyze the Atari game Atlantis [9]. This is an Atari game featured in the Gymnasium collection where the objective is to defend the underwater city of Atlantis from aerial attacks by shooting to the enemy planes which fly closer to the surface of the water. The player controls three fixed gun turrets located on the center and the sides of the cityscape, firing at enemy aircrafts that pass overhead in varying patterns and speeds. Players can perform 4 actions: NOOP (no movement), FIRE (fire from THE central turret), LEFTFIRE, and RIGHTFIRE. Each destroyed enemy plane -after being shot- increases the player's score, with big planes being worth 600 points and small ones 200. A terminal state is reached when Atlantis is completely destroyed, marked by the loss of all defensive turrets and structures. The Atari game is impossible to win -planes kept coming faster and the city is eventually destroyed-,as such, the RL training stops after 130,000 points are obtained in more than 25 episodes.

This is a complex environment for which each state is represented by the game screen which has dimensions of 210x160x3, presented in RGB. Preprocessing is applied making it a greyscale and 84x84 pixels, which effectively reduces the state space without sacrificing functionality. Also, compression every four frames into one is applied in order to guide the agent with the idea of movement. Also rewards for the agent are divided by 100, which is more manageable for using neural networks. The architecture used is a neural network with 3 convolutional layers to process the visual input of the the current state and 2 fully connected layers for further processing, all using the ReLU activation function. The output of this network is the number of actions.

Three different algorithms where tried: vanilla DQN, Prioritized Experiences Replay (PER) with DQN, and Double DQN (DDQN). They took all less than 8 hours in a NVIDIA 1650 GTX, with PER taking almost twice per episode than vanilla DQN, which make sense given the extra complexity of the samples introduced by the sampling of PER. The networks were trained for 5000 episodes using the hyperparameters on the Appendix B.

10 Analysis of the results.

The three networks converged to an strategy that won the game in less than 1300, effectively breaking the training loop (5000 episodes). All of them showcase unstable learning as measured in per episode rewards

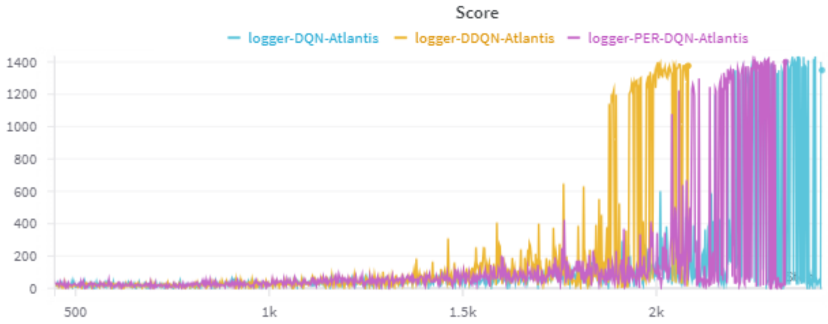


Figure 9: Rewards for DQN with PER, vanilla DQN, and DDQN. *

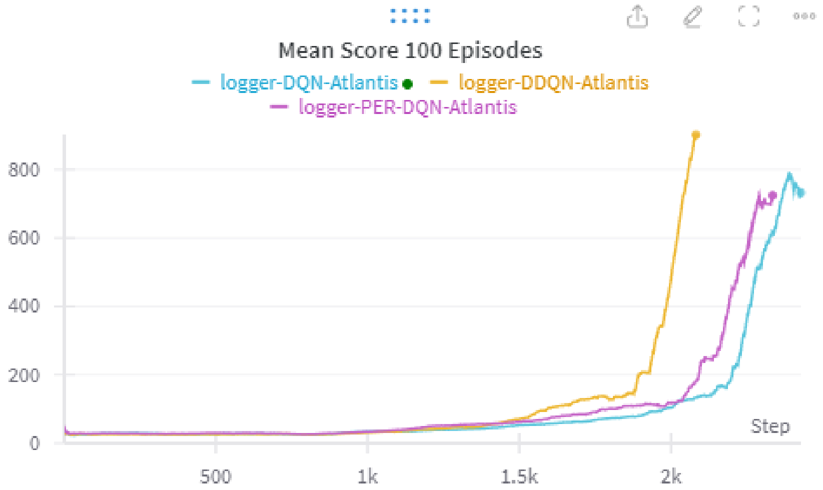


Figure 10: Mean rewards over 100 episodes for DQN with PER, vanilla DQN, and DDQN. *

(*) Episodes are half of the x-axis because of Weights and Biases logging.

in Figure 9, but general convergence can be observed in Figure 10 by taking the mean over 100 episodes. Both figures differ in their ranges because the peak raw rewards are usually not sustained indefinitely as we saw in tabular Q-learning 12 because here the state space is much bigger and it's likely that the agent hasn't seen this space before, or at least is not processing it in the same way. Comparatively, the PER DQN Network converges faster than the vanilla DQN, which aligns with the background theory that says that sampling more relevant experiences yields faster learning. The DDQN network converges the fastest, outperforming significantly to vanilla DQN and PER DQN; this is also shown in the DDQN paper [6]. The key reason for this may lie in the rapid nature of Atlantis which displays continuous firing of targets that appearing large volumes and diverse positions. DQN tends to overestimate Q-values in highly dynamic environments where the agent must adapt to frequent changes in input, whereas DDQN selects actions that are more consistently beneficial, influencing the long-term return because the game will last for longer.

Thanks to the use of convolutional deep neural networks and the reinforcement learning framework, we are able to train agents that outperform the best humans playing Atlantis by a huge margin, getting 20 times higher score. Humans can come up with heuristics to play beforehand like "whenever a plane appears some distance away of the turret range it should fire" and can also reason about them "because the time the bullet takes to arrive the target will already be there". Despite AI agents being unable to engage with this sort of reasoning, they generate sufficiently good representations to play the game, which coupled with their extremely low reaction times -they can provide an informed response to every four frames of video- it's enough to play super-humanly.

11 Contributions

We both contributed evenly to the task having a collaborative approach over the whole project.

We both chose the Task 1 environment, and discussed the implementation of the different modules. He did the backbone of the code (structure of code and plotting) and I added some key modules (reward functions). Then I leaded the parameter finetuning section. We both discussed several environments for Task 2 and converged on Breakout. He did the backbone structure of the code and tweaked the hyperparameters. I implemented the first version of Prioritized Experience Replay. The code was all discussed, checked, reviewed and debugged. The report was evenly written, and peer-reviewed (between us).

References

[1] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[2] OpenAI Gymnasium, "Breakout environment." OpenAI Gymnasium for Atari, 2024.

[3] P. Nash, "A brief history of reinforcement learning in game play." `https://www.projectnash.com/a-brief-history-of-reinforcement-learning-in-game-play/`, 2021. Accessed: 2024-05-10.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

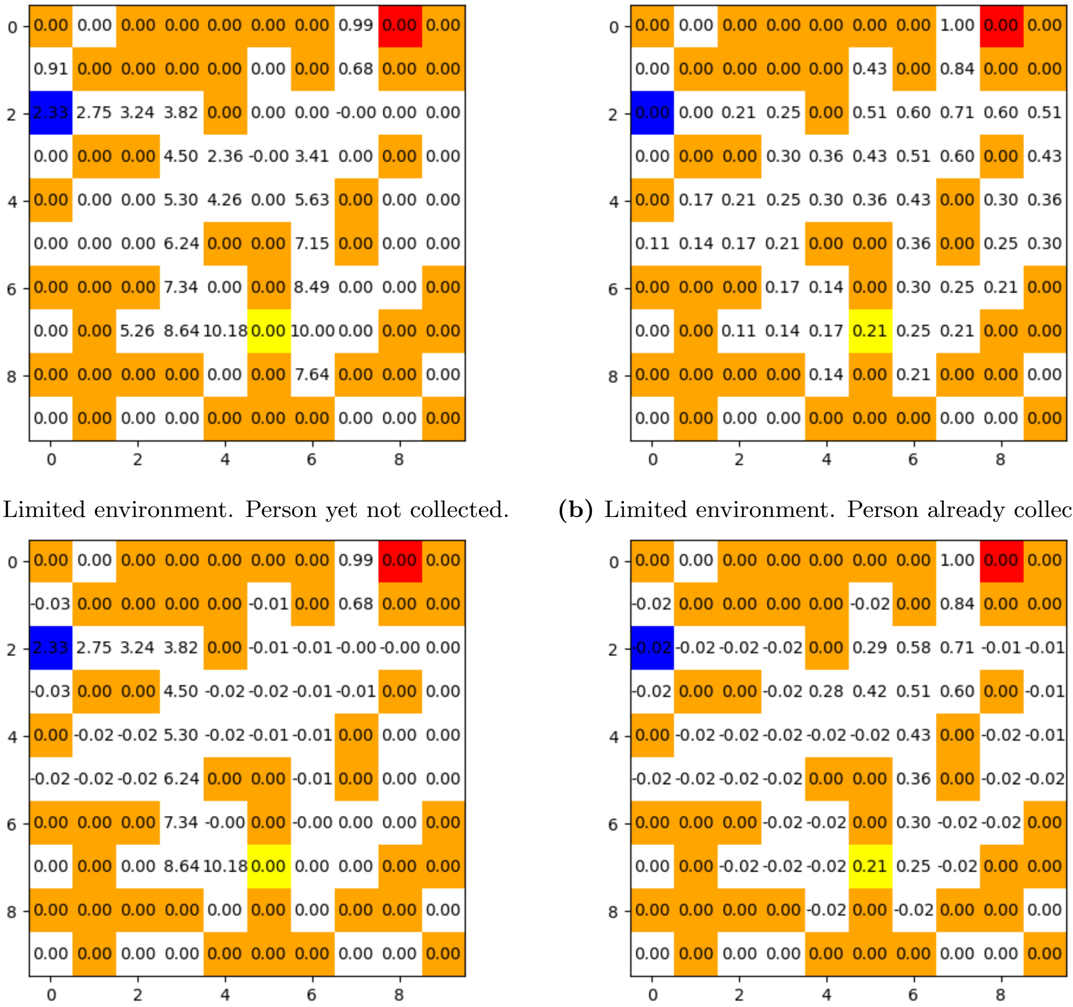
[6] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[8] Howuhh, "Prioritized experience replay." `https://github.com/Howuhh/prioritized_experience_replay`, 2016. Implementation with proportional prioritization.

[9] OpenAI Gymnasium, "Atlantis environment." OpenAI Gymnasium for Atari, 2024.

A Appendix: Q values map



	0		2		4		6		8	
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00	0.00
	0.91	0.00	0.00	0.00	0.00	0.00	0.00	0.68	0.00	0.00
2	2.33	2.75	3.24	3.82	0.00	0.00	0.00	-0.00	0.00	0.00
	0.00	0.00	0.00	4.50	2.36	-0.00	3.41	0.00	0.00	0.00
4	0.00	0.00	0.00	5.30	4.26	0.00	5.63	0.00	0.00	0.00
	0.00	0.00	0.00	6.24	0.00	0.00	7.15	0.00	0.00	0.00
6	0.00	0.00	0.00	7.34	0.00	0.00	8.49	0.00	0.00	0.00
	0.00	0.00	5.26	8.64	10.18	0.00	10.00	0.00	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	7.64	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(a) Limited environment. Person yet not collected.

	0		2		4		6		8	
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.43	0.00	0.84	0.00	0.00
2	0.00	0.00	0.21	0.25	0.00	0.51	0.60	0.71	0.60	0.51
	0.00	0.00	0.00	0.30	0.36	0.43	0.51	0.60	0.00	0.43
4	0.00	0.17	0.21	0.25	0.30	0.36	0.43	0.00	0.30	0.36
	0.11	0.14	0.17	0.21	0.00	0.00	0.36	0.00	0.25	0.30
6	0.00	0.00	0.00	0.17	0.14	0.00	0.30	0.25	0.21	0.00
	0.00	0.00	0.11	0.14	0.17	0.21	0.25	0.21	0.00	0.00
8	0.00	0.00	0.00	0.00	0.14	0.00	0.21	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(b) Limited environment. Person already collected.

	0		2		4		6		8	
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00	0.00
	-0.03	0.00	0.00	0.00	0.00	-0.01	0.00	0.68	0.00	0.00
2	2.33	2.75	3.24	3.82	0.00	-0.01	-0.01	-0.00	-0.00	0.00
	-0.03	0.00	0.00	4.50	-0.02	-0.02	-0.01	-0.01	0.00	0.00
4	0.00	-0.02	-0.02	5.30	-0.02	-0.01	-0.01	0.00	0.00	0.00
	-0.02	-0.02	-0.02	6.24	0.00	0.00	-0.01	0.00	0.00	0.00
6	0.00	0.00	0.00	7.34	-0.00	0.00	-0.00	0.00	0.00	0.00
	0.00	0.00	0.00	8.64	10.18	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(c) Terminal environment. Person yet not collected.

	0		2		4		6		8	
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
	-0.02	0.00	0.00	0.00	0.00	-0.02	0.00	0.84	0.00	0.00
2	-0.02	-0.02	-0.02	-0.02	0.00	0.29	0.58	0.71	-0.01	-0.01
	-0.02	0.00	0.00	-0.02	0.28	0.42	0.51	0.60	0.00	-0.01
4	0.00	-0.02	-0.02	-0.02	-0.02	-0.02	0.43	0.00	-0.02	-0.01
	-0.02	-0.02	-0.02	-0.02	0.00	0.00	0.36	0.00	-0.02	-0.02
6	0.00	0.00	0.00	-0.02	-0.02	0.00	0.30	-0.02	-0.02	0.00
	0.00	0.00	-0.02	-0.02	-0.02	0.21	0.25	-0.02	0.00	0.00
8	0.00	0.00	0.00	0.00	-0.02	0.00	-0.02	0.00	0.00	0.00
	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(d) Terminal environment. Person already collected.

Figure 12: Maximum Q value of an action at each state of an optimal agent. $\gamma = 0.85$

B Appendix: Task 2 Hyperparamaters

Hyperparameter DQN, DDQN and PER	**Value**
Discount Factor γ	0.99
Minibatch size	64
Experience replay buffer size	100000
Target Network update frequency	10000
Network Update frequency	4
Learning rate	0.0001
Initial exploration epsilon	1
Final exploration epsilon	0.01
Epsilon linear decay frames	500000
Replay buffer start size	5000
No op max	30
PER α	0.6
PER β	0.4

Table 2: First set of hyperparameters for DQN, DDQN and PER variants

C Appendix: Breakout strategy

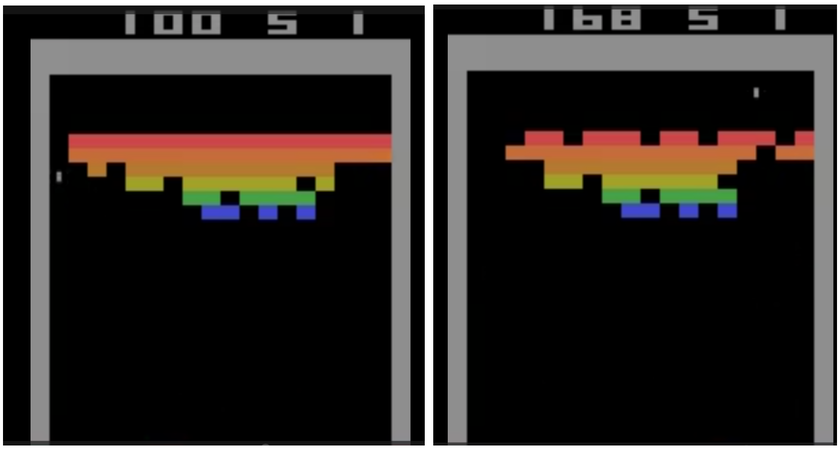


Figure 13: s trategy developed during Breakout, where the agent creates a hole in the edges of the environment in order to bounce the ball between the ceiling and the top row of blocks.

D Code

D.1 Task 1

D.1.1 Maze_env.py

```python

import numpy as np
from matplotlib import pyplot as plt


class Maze_env:
    """
    Represents a maze navigation environment for reinforcement learning tasks.
    It manages the maze layout, positions of start, target, and coin, and rewards/transitions.
    Functionality includes:
    - `__init__(start, target, coin, maze)`: Initializes the environment.
    - `plot_env()`: Visualizes the maze with important positions highlighted.
    - `plot_env_position(position, timestep)`: Visualizes  maze with agent's position at specific timestep.
    - `create_r_matrix()`: Generates a reward matrix based on the maze layout.
    - `reward(state, action)`: Calculates the reward for an action taken from a state.
    - `transition(state, action)`: Determines the new state after an action.
    - `done()`: Checks if the target has been reached, ending the episode.
    - `create_q_matrix()`: Initializes a Q-learning matrix for action selection.
    """

    def __init__(self, start, target, coin, maze, reward_type):
        self.maze = maze
        self.target = target
        self.start = start
        self.coin = coin
        self.reward_type = reward_type
        self.position = 0
        self.R = self.create_r_matrix(self.reward_type)
        print(f" Shape of the R matrix is {self.R.shape}")
        self.Q = self.create_q_matrix()
        print(f" Shape of the Q matrix is {self.Q.shape}")
        self.coin_collected = False
        self.terminate = False

    def plot_env(self):
        cmap = plt.cm.colors.ListedColormap(
            ["white", "orange", "red", "blue", "yellow"]
        )
        maze_plot = self.maze.copy()
        maze_plot[self.target] = 2
        maze_plot[self.start] = 3
        maze_plot[self.coin] = 4
        plt.imshow(maze_plot, cmap=cmap)
        plt.show()

    def plot_env_position(self, position, timestep):
        cmap = plt.cm.colors.ListedColormap(
            ["white", "orange", "red", "blue", "yellow"]
        )
        maze_plot = self.maze.copy()
        maze_plot[self.target] = 2
        maze_plot[position] = 3
        maze_plot[self.coin] = 4
        plt.imshow(maze_plot, cmap=cmap)
        plt.savefig(f"img/plot_{timestep:06d}.png", dpi=300)
        plt.show()
        plt.close()

    def create_r_matrix(self, reward_type):
        """
        This synthesizes the reward and transition functions.
        reward_type (str): The type of reward to use.
        Options are "terminal_movement" and "free_movement".
        """
        actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        num_states = self.maze.shape[0] * self.maze.shape[1] * 2  # times coin state
        coin_states = 2  # 0 for no coin collected, 1 for coin collected
        R = np.full(
            (self.maze.shape[0], self.maze.shape[1], coin_states, len(actions)), np.nan)

        if reward_type == "terminal_movement":
            print("Reward type: Terminal Movement")
            # actions beyond limits get -10 (and terminate)
            # actions to a 0 -10 (and terminate)
            # action to coin get 200
            # action to target get 100
            # allowed actions get -1 (for the time)

            for i in range(self.maze.shape[0]):
                for j in range(self.maze.shape[1]):
                    for coin_state in range(coin_states):
                        for action_index, action in enumerate(actions):
                            new_i, new_j = i + action[0], j + action[1]
                            if new_i >= 0 and new_i < self.maze.shape[0] and new_j >= 0 and new_j < self.maze.shape[1]:
                                # Actions to a wall (1 in the maze) get -1
                                if self.maze[new_i, new_j] == 1:
                                    R[i, j, coin_state, action_index] = -1 # for the fire
                                elif self.maze[new_i, new_j] == 0:
                                    R[i, j, coin_state, action_index] = -0.005  # for an allowed action
                                    if (new_i, new_j) == self.coin and not coin_state:
                                        # print("Assigning coin")
                                        R[i, j, coin_state, action_index] = 10  # coin
                                    elif (new_i, new_j) == self.target:
                                        # print("Assigning target")
                                        R[i, j, coin_state, action_index] = 1  # target
                            else:
                                R[i, j, coin_state, action_index] = -1  # actions beyond the limits are forbidden

            return R
            # then add the transition function so that if reward smaller than -1, then terminate.

        if reward_type == "limited_movement":
            print("Reward␣type:␣Limited␣Movement")
            # actions beyond limits get None (can't move)
            # actions to a 0 (get -10)
            # action to coin get 200
            # action to target get 100
            # allowed actions get -1 (for the time)

            for i in range(self.maze.shape[0]):
                for j in range(self.maze.shape[1]):
                    for coin_state in range(coin_states):
                        for action_index, action in enumerate(actions):
                            new_i, new_j = i + action[0], j + action[1]

                            if new_i >= 0 and new_i < self.maze.shape[0] and new_j >= 0 and new_j < self.maze.shape[1]: # inside of maze
                                # Actions to a wall (1 in the maze) get None
                                if self.maze[new_i, new_j] == 1:
                                    R[i, j, coin_state, action_index] = None # for the fire
                                elif self.maze[new_i, new_j] == 0:
                                    R[i, j, coin_state, action_index] = -0.005  # for an allowed action
                                    if (new_i, new_j) == self.coin and not coin_state:
                                        R[i, j, coin_state, action_index] = 10  # coin
                                    elif (new_i, new_j) == self.target:
                                        R[i, j, coin_state, action_index] = 1  # target
                            else:
                                R[i, j, coin_state, action_index] = None  # actions beyond the limits are forbidden
            return R

    def transition_R(self, state, action, reward_type):
        initial_state = state
        x, y = initial_state
        new_x = x
        new_y = y
        if action == 0:  # up
            new_x -= 1
        elif action == 1:  # down
            new_x += 1
        elif action == 2:  # left
            new_y -= 1
        elif action == 3:  # right
            new_y += 1

        if reward_type == "terminal_movement":
            if new_x >= 0 and new_x < self.maze.shape[0] and new_y >= 0 and new_y < self.maze.shape[1]:
                if self.R[x, y, int(self.coin_collected), action] == -1: # fire
                    # print("Fire")
                    self.terminate = True
                    return state
                elif (new_x,new_y) == self.coin and not self.coin_collected: # coin
                    # print("Coin")
                    self.coin_collected = True
                    # print(self.coin_collected)
                    return new_x,new_y
                elif (new_x,new_y) == self.target: # target
                    # print("Target")
                    self.terminate = True
                    return new_x,new_y
                elif self.R[x, y, int(self.coin_collected), action] == -0.005: # normal action
                    # print("Allowed")
                    return new_x,new_y
            else:
                self.terminate = True  # walls
                # print("Out of bounds")
                return state

        if reward_type == "limited_movement": # should not attempt to access fire or wall
            if new_x >= 0 and new_x < self.maze.shape[0] and new_y >= 0 and new_y < self.maze.shape[1]:
                if (new_x,new_y) == self.coin and not self.coin_collected: # coin
                    self.coin_collected = True
                    return new_x,new_y
                elif (new_x,new_y) == self.target: # target
                    self.terminate = True
                    return new_x,new_y
                elif self.R[x, y, int(self.coin_collected), action] == -0.005: # normal action
                    return new_x,new_y

    def done(self):
        return self.terminate

    def coin_reached(self):
        return self.coin_collected

    def create_q_matrix(self):
        Q = np.zeros_like(self.R)
        return Q

if __name__ == "__main__":
    maze = np.array(
        [
            [1, 0, 1, 1, 1, 1, 1, 0, 0, 1],
            [0, 1, 1, 1, 1, 0, 1, 0, 1, 1],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 1, 0, 0, 0, 0, 0, 1, 0],
            [1, 0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 1, 0, 1, 0, 0],
            [1, 1, 1, 0, 0, 1, 0, 0, 0, 1],
            [0, 1, 0, 0, 0, 0, 0, 0, 1, 1],
            [1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
            [0, 1, 0, 0, 1, 1, 1, 0, 0, 1],
        ]
    )
    env = Maze_env((2, 0), (0, 8), (7, 5), maze, reward_type="terminal_movement")
    env.plot_env()
    print("Info")
    print(env.R[0, 7, 0])
    print(env.R[0, 7, 1])
    print(env.R[7, 5, 0])
    print(env.R[7, 5, 1])
    print(env.R[7, 4, 0])
    print(env.R[7, 4, 1])
```

Listing 1: Maze environment code

D.1.2 agent.py

```python

import numpy as np
import cv2
import os
from matplotlib import pyplot as plt
from maze_env import Maze_env
from tqdm.auto import tqdm


class Q_learning:
    """
    Implements the Q-learning algorithm for reinforcement learning tasks within a predefined environment.
    This class is responsible for learning optimal action-selection policies to maximize rewards over episodes of interactions with the environment.

    - `__init__(alpha, gamma, epsilon, episodes, steps, env, states)`: Initializes the learning parameters, environment, and states.
    - `plot_rewards()`: Plots the rewards accumulated over each episode, visualizing the learning progress.
    - `show_Q_spec(coord)`: Displays Q-values for a specific coordinate/state.
    - `greedy_policy(state)`: Selects an action based on a greedy policy (highest Q-value) with an epsilon chance of random action for exploration.
    - `softmax_policy(state, temperature)`: Selects an action based on the softmax of Q-values, factoring in the temperature for exploration-exploitation balance.
    - `train()`: Conducts the learning process over a specified number of episodes and steps per episode, updating Q-values based on the received rewards.
    - `create_video()`: Generates a video from saved images of the agent's journey through the maze, illustrating the learned policy in action.
    - `test(limit)`: Evaluates the learned policy by navigating the environment for a given number of steps, visualizing the path taken and summarizing the rewards.

    The class utilizes epsilon-greedy and softmax policies for action selection, balancing the exploration of the state space with the exploitation of known rewards.
    """

    def __init__(self, alpha, gamma, epsilon, episodes, steps, env, policy):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.temperature = 50.0
        self.policy = policy
        self.R = env.R
        self.R_mod = self.R
        self.Q = env.Q
        self.episodes = episodes
        self.steps = steps
        self.start = env.start
        self.target = env.target
        self.coin = env.coin
        self.env = env
        self.episodes_rewards = []
        self.max_list_size = 10
        self.list_rewards = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        self.threshold = 2
        self.window_size = 20
        self.current_average = 0

        # print("Initial Q matrix shape is '{}'".format(self.Q.shape))
        # print("Initial Q matrix values are '{}'".format(self.Q))

    def plot_rewards(self):
        plt.plot(self.episodes_rewards)
        plt.show()

    def show_Q_spec(self, coord):
        i, j = coord
        print(self.Q[i, j, int(self.env.coin_reached()), :])

    def greedy_policy(self, state):
        i, j = state
        available_actions = np.where(~np.isnan(self.R_mod[i, j, int(self.env.coin_reached())]))[0]
        # print(available_actions)
        q_values = [self.Q[i, j, int(self.env.coin_reached()), a] for a in available_actions]
        best_actions = available_actions[np.where(q_values == np.max(q_values))[0]]
        # print(best_actions)

        # available_actions = np.array([0, 1, 2, 3])
        # q_values = [self.Q[state, a] for a in available_actions]
        # best_actions = available_actions[np.where(q_values == np.max(q_values))[0]]

        if np.random.uniform() < self.epsilon:
            # a = np.random.choice(4)
            a = np.random.choice(available_actions)
        else:
            #                       a = np.argmax(self.Q[s,:])
            a = np.random.choice(best_actions)
        # print(a)
        return a

    def softmax_policy(self, state):
        i, j = state
        available_actions = np.where(~np.isnan(self.R_mod[i, j, int(self.env.coin_reached())]))[0]
        # print(f"Available actions: {available_actions}")
        q_values = [self.Q[i, j, int(self.env.coin_reached()), a] for a in available_actions]
        max_q_value = np.max(q_values)
        exp_values = np.exp((q_values - max_q_value) / self.temperature)
        action_probs = exp_values / np.sum(exp_values)
        # print(f"Actions Probability: {action_probs}")
        # Sample an action based on the probabilities
        selected_action_index = np.random.choice(len(action_probs), p=action_probs)
        selected_action = available_actions[selected_action_index]
        # print(f"Selected Action: {selected_action}")

        return selected_action

    def train(self):
        print("Target is '{}'".format(self.target))
        print("Starting state is '{}'".format(self.start))

        for episode in tqdm(range(self.episodes), desc= f"Training agent on {self.episodes} episodes", unit="episode", total=self.episodes):
            # print("New episode")
            s = self.start
            episode_reward = 0
            self.env.coin_collected = False
            self.env.terminate = False
            # print("New episode")
            for timestep in range(self.steps):
                # print(self.env.coin_reached())
                i, j = s
                # Epsilon-greedy action choice
                if self.policy == "greedy":
                    a = self.greedy_policy(s)
                elif self.policy == "softmax":
                    a = self.softmax_policy(s)
                else:
                    raise ValueError("Policy must be 'greedy' or 'softmax'")
                #a = self.softmax_policy(s, self.temperature)
                # Environment updating
                # r = env.reward(s, a)
                # print(self.R_mod[i,j,int(self.env.coin_collected)])
                r = self.R_mod[i, j, int(self.env.coin_reached()), a]
                # print(r)
                # if self.env.coin_reached():
                #     print()
                #     print("Coin collected")
                # print(self.env.coin_reached())
                # print("Reward")
                episode_reward += r
                new_state = self.env.transition_R((i, j), a, self.env.reward_type)
                # print(f"Action is {a}, and state is {(i, j)}")
                new_i, new_j = new_state

                if r == 10: # picked up coin for first time
                    # print("COOOOOOOOOOOOOOOOIIIIIIIINNNNNNN")
                    # Current q in state o and next in state 1 for coin_collected
                    self.Q[i, j, 0, a] = self.Q[i, j, 0, a] + self.alpha * (r + self.gamma * np.max(
                        self.Q[new_i, new_j, 1, :]) - self.Q[i, j, 0, a])
                else:
                    self.Q[i, j, int(self.env.coin_reached()), a] = self.Q[i, j, int(
                        self.env.coin_reached()), a] + self.alpha * (r + self.gamma * np.max(
                        self.Q[new_i, new_j, int(self.env.coin_reached()), :]) - self.Q[
                                                                        i, j, int(self.env.coin_reached()), a])

                if self.env.done():
                    # print("Death")
                    # print(self.env.terminate)
                    break
                s = new_state

            self.episodes_rewards.append(episode_reward)

            self.list_rewards.append(episode_reward)
            if len(self.list_rewards) > self.max_list_size:
                self.list_rewards.pop(0)
            window = self.list_rewards[-self.window_size:]
            window_average = sum(window) / self.window_size
            self.current_average = window_average

            if episode == self.episodes - 1:
                if self.policy == "greedy":

                    print(
                        "Episode {} finished. Episode Reward {}. Timesteps {}. Average {}. Epsilon {}".format(
                            episode,
                            episode_reward,
                            timestep,
                            window_average,
                            self.epsilon,
                        )
                    )
                else:
                    print(
                        "Episode {} finished. Episode Reward {}. Timesteps {}. Average {}. Temp {}".format(
                            episode,
                            episode_reward,
                            timestep,
                            window_average,
                            self.temperature,
                        )
                    )
            lambda_rate = 0.1
            mimimum_epsilon = 0.00
            initial_epsilon = 1
            self.epsilon = mimimum_epsilon + (initial_epsilon - mimimum_epsilon) * np.exp(-lambda_rate * episode)

            lambda_rate_temp = 0.1
            minimum_temperature = 0.001
            initial_temperature = 5
            self.temperature = minimum_temperature + (initial_temperature - minimum_temperature) * np.exp(-lambda_rate_temp * episode)

    def create_video(self):
        image_folder = "img"  # Directory containing your saved plot images
        video_name = "video_agent.mp4"

        images = [
            img
            for img in os.listdir(image_folder)
            if img.endswith((".jpg", ".jpeg", ".png"))
        ]
        frame = cv2.imread(os.path.join(image_folder, images[0]))
        height, width, layers = frame.shape

        video = cv2.VideoWriter(
            video_name, cv2.VideoWriter_fourcc(*"mp4v"), 1, (width, height)
        )

        for image in images:
            video.write(cv2.imread(os.path.join(image_folder, image)))

        cv2.destroyAllWindows()
        video.release()

    def test(self, limit=40):
        s = self.start
        print("Starting state is '{}'".format(s))
        episode_reward = 0
        env.coin_collected = False
        env.terminate = False
        for timestep in range(limit):
            i, j = s
            self.env.plot_env_position(s, timestep)
            a = np.argmax(self.Q[i, j, int(self.env.coin_reached())])

            # Environment updating
            r = self.R_mod[i, j, int(self.env.coin_reached()), a]
            print(f"Step {timestep}. Action is {a}. State is {(i, j)}. Q value of {self.Q[i, j, int(self.env.coin_reached()), a]}. And reward {r}")
            episode_reward += r
            new_state = self.env.transition_R((i, j), a, self.env.reward_type)
            new_i, new_j = new_state

            if env.done():
                self.env.plot_env_position(new_state, timestep+1)
                break
            s = new_state
        # print('Episode Reward {}.Q matrix values:\n{}'.format(episode_reward, self.Q.round(1)))
        self.create_video()


if __name__ == "__main__":
    maze = np.array(
        [
            [1, 0, 1, 1, 1, 1, 1, 0, 0, 1],
            [0, 1, 1, 1, 1, 0, 1, 0, 1, 1],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 1, 1, 0, 0, 0, 0, 0, 1, 0],
            [1, 0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 1, 0, 1, 0, 0],
            [1, 1, 1, 0, 0, 1, 0, 0, 0, 1],
            [0, 1, 0, 0, 0, 0, 0, 0, 1, 1],
            [1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
            [0, 1, 0, 0, 1, 1, 1, 0, 0, 1],
        ]
    )
    env = Maze_env(start=(2, 0), target=(0, 8), coin=(7, 5), maze=maze, reward_type="limited_movement" )

    q_learning = Q_learning(alpha=0.9, gamma=0.85, epsilon=1, episodes=250, steps=200, env=env, policy="softmax")
    print("INFO. State is (ROW, COLUMN IS_COIN). Action is [up, down, left, right]")
    print(f" R values for state (0, 7, 0) {q_learning.R_mod[0, 7, 0]}")
    print(f" R values for state (0, 7, 1) {q_learning.R_mod[0, 7, 1]}")
    print(f" R values for state (7, 5, 0) {q_learning.R_mod[7, 5, 0]}")
    print(f" R values for state (7, 5, 1) {q_learning.R_mod[7, 5, 1]}")
    print(f" R values for state (7, 4, 0) {q_learning.R_mod[7, 4, 0]}")
    print(f" R values for state (7, 4, 1) {q_learning.R_mod[7, 4, 1]}")

    q_learning.train()
    q_learning.plot_rewards()
    # q_learning.test()
    print(f" Q values for state (3, 3, 0) {q_learning.Q[3, 3, 0]}")
    print(f" Q values for state (7, 4, 0) {q_learning.Q[7, 4, 0]}")
```

Listing 2: Q-Learning agent code

D.2 Task 2 code

D.2.1 buffer.py

```
from collections import namedtuple, deque
import numpy as np
import random
import torch


class ReplayMemory(object):
    def __init__(self, capacity, use_per=False, alpha=0.6, epsilon=0.001):
        self.use_per = use_per
        self.capacity = capacity
        self.memory = deque([], maxlen=capacity)
        self.epsilon = epsilon
        self.count = 0
        if self.use_per:
            self.alpha = alpha
            self.sum_tree = SumTree(capacity)
            self.max_priority = 1.0

    def push(self, transition):
        self.memory.append(transition)
        if self.use_per:
            self.sum_tree.add(self.max_priority, self.count)
            self.count = (self.count + 1) % self.capacity

    def sample(self, batch_size, beta=0.4):
        batch = []
        idxs = []
        is_weights = []
        if self.use_per:
            total_priority = self.sum_tree.total
            # print(f"Total Priority: {total_priority}")
            segment = total_priority / batch_size
            for i in range(batch_size):
                # Guard against sampling error: https://github.com/rlcode/per/issues/4
                while True:
                    s = random.uniform(segment * i, segment * (i + 1))
                    tree_idx, priority, idx = self.sum_tree.get(s)
                    if idx is not None:
                        break
                    else:
                        print("Attempted to sample unitialised memory")
                sampling_probability = priority / total_priority
                is_weight = (len(self.memory) * sampling_probability) ** -beta
                is_weights.append(is_weight)
                # print(idx)
                batch.append(self.memory[idx])
                idxs.append(tree_idx)
            max_weight = max(is_weights)
            is_weights = [w / max_weight for w in is_weights]
        else:
            batch = random.sample(self.memory, batch_size)
            is_weights = [1.0] * batch_size
            idxs = None
        return batch, idxs, is_weights

    def update_priority(self, idxs, priorities):
        if self.use_per:
            for idx, priority in zip(idxs, priorities):
                adjusted_priority = (priority + self.epsilon) ** self.alpha
                self.max_priority = max(self.max_priority, adjusted_priority)
                self.sum_tree.update(idx, adjusted_priority)
        else:
            raise ValueError("Not using PER")

    def __len__(self):
        return len(self.memory)


```

```
# https://github.com/Howuhh/prioritized_experience_replay/blob/main/memory/tree.py
class SumTree:
    """This will be binary tree stored as a list (self.tree), where:
     - the experiences priorities are the leaves, stored in the second half of the list
     - the remaining positions (first half) are the binary sums of children nodes
     - the root tree (the first element) is the sum of all the elements"""
    def __init__(self, size):
        self.nodes = [0] * (2 * size - 1)
        self.data = [None] * size

        self.size = size
        self.count = 0
        self.real_size = 0

    @property
    def total(self):
        return self.nodes[0]

    def update(self, data_idx, value):
        idx = data_idx + self.size - 1  # child index in tree array
        change = value - self.nodes[idx]
        self.nodes[idx] = value
        parent = (idx - 1) // 2
        while parent >= 0:
            self.nodes[parent] += change
            parent = (parent - 1) // 2

    def add(self, value, data):
        self.data[self.count] = data
        self.update(self.count, value)
        self.count = (self.count + 1) % self.size
        self.real_size = min(self.size, self.real_size + 1)

    def get(self, cumsum):
        assert cumsum <= self.total

        idx = 0
        while 2 * idx + 1 < len(self.nodes):
            left, right = 2*idx + 1, 2*idx + 2
            if cumsum <= self.nodes[left]:
                idx = left
            else:
                idx = right
                cumsum = cumsum - self.nodes[left]

        data_idx = idx - self.size + 1
        return data_idx, self.nodes[idx], self.data[data_idx]
```

Listing 3: Buffe which allows Prioritized Sampling

D.2.2 dqn.py

```
import gymnasium as gym
from gymnasium.utils.save_video import save_video

import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import os
from buffer import ReplayMemory
from logger import Logger

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device is {device}")

Transition = namedtuple('Transition',
                        ('state', 'action', 'reward', 'next_state', 'done'))

os.environ['https_proxy'] = "http://hpc-proxy00.city.ac.uk:3128"


class DQN(nn.Module):

    def __init__(self, n_observations, n_actions, hidden_units=512):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, hidden_units)
        self.layer2 = nn.Linear(hidden_units, hidden_units)
        self.layer3 = nn.Linear(hidden_units, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)


class DQNCNN(nn.Module): # DQN/DDQN
    def __init__(self, input_shape, n_actions, hidden_units=512):
        super(DQNCNN, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        conv_out_size = self.get_conv_out_size(input_shape)

        self.value = nn.Sequential(
            nn.Linear(conv_out_size, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, n_actions)
        )

    def get_conv_out_size(self, shape):
        conv_size = self.conv(torch.zeros(1, *shape))
        return int(np.prod(conv_size.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        return self.value(conv_out)


class Agent:
    def __init__(self, env, per=False, double = False, logger = None):

        self.logger = logger
        self.GAMMA = 0.99
        self.LR = 1e-4
        self.ALPHA = 1
        self.update_frequency = 4
        self.update_target_frequency = 10000 # 20k for tuned ddqn
        self.batch_size = 64
        self.per = per
        self.double_dqn = double

        self.replay = ReplayMemory(100000, use_per=self.per)
        if self.per:
            self.alpha = self.replay.alpha
            self.sum_tree = self.replay.sum_tree
            self.max_priority = self.replay.max_priority
        self.memory = self.replay.memory

        self.max_episodes = 5000
        self.number_episodes = 0
        self.max_timesteps = 2000
        self.number_timesteps = 0
        self.epsilon = 1

        # Get number of actions from gym action space
        self.env = env
        self.n_actions = 4
        self.number_lives = 5
        # self.n_actions = env.action_space.shape[0]
        # num_bins = 61  # Number of bins for each action dimension
        # self.n_actions = num_bins ** self.n_actions
        print(self.n_actions)
        print(f"Number actions: {self.n_actions}")
        seed = None
        self.random_state = np.random.RandomState() if seed is None else np.random.RandomState(seed)

        # Get the number of state observations
        self.state, self.info = env.reset()
        print(f"State shape: {self.state.shape}")
        # self.n_observations = len(self.state)
        self.n_observations = self.state.shape
        self.policy_net = DQNCNN(self.n_observations, self.n_actions, hidden_units=512).to(device)
        self.target_net = DQNCNN(self.n_observations, self.n_actions, hidden_units=512).to(device)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.LR, amsgrad=True)
        print(self.n_observations)
        print(env.observation_space.shape)
        # self.policy_net = DQN(env.observation_space.shape, self.n_actions).to(device)
        # self.target_net = DQN(env.observation_space.shape, self.n_actions).to(device)

        self.video = []

    # def discrete2cont_action(self, action):
    #     # Map the discrete action index to continuous torques
    #     num_bins = 61
    #     action_indices = np.unravel_index(action, (num_bins, num_bins, num_bins))
    #     torque_min = -1.0
    #     torque_max = 1.0
    #     torques = [torque_min + (torque_max - torque_min) * idx / (num_bins - 1) for idx in action_indices]
    #     return np.array(torques)

    def has_sufficient_experience(self):
        """True if agent has enough experience to train on a batch of samples; False otherwise."""
        # return len(self.memory) >= self.batch_size
        if len(self.memory) == 5000:
            print("Sufficient experience recently obtained!!!")
        return len(self.memory) >= 5000

    def has_full_experience(self):
        """True if agent has enough experience to train on a batch of samples; False otherwise."""
        # return len(self.memory) >= self.batch_size
        if len(self.memory) == 100000:
            return len(self.memory) >= 100000

    def save(self, filepath):
        checkpoint = {
            "q-network-state": self.policy_net.state_dict(),
            "optimizer-state": self.optimizer.state_dict(),
        }
        torch.save(checkpoint, filepath)

    def choose_action(self, state):
        # print(state.shape)
        # need to reshape state array and convert to tensor
        state_tensor = (torch.from_numpy(np.array(state)).unsqueeze(dim=0).to(device)).float()
        # choose uniform at random if agent has insufficient experience
        if not self.has_sufficient_experience():
            action = self.uniform_random_policy(state_tensor)
        else:
            # print("Sufficient experience")
            action = self.epsilon_greedy_policy(state_tensor, self.epsilon)
        return action

    def epsilon_greedy_policy(self, state, epsilon):
        """With probability epsilon explore randomly; otherwise exploit knowledge optimally."""
        if self.random_state.random() < epsilon:
            action = self.uniform_random_policy(state)
        else:
            action = self.greedy_policy(state)
        return action

    def uniform_random_policy(self, state):
        """Choose an action uniformly at random."""
        # random_vector = np.random.(low=-1, high=1, size=self.n_actions)
        # return random_vector
        return self.random_state.randint(self.n_actions)

    def greedy_policy(self, state):
        # print(state.shape)
        # print(state.dtype)
        """Choose an action that maximizes the action_values given the current state."""
        action = (self.policy_net(state)
                  .argmax()
                  .cpu()  # action_values might reside on the GPU!
                  .item())
        return action

    def select_greedy_actions(self, states, q_network):
        _, actions = q_network(states).max(dim=1, keepdim=True)
        # print(actions)
        return actions

    def evaluate_selected_actions(self, states,actions,rewards,dones,gamma,q_network):
        """Compute the Q-values by evaluating the actions given the current states and Q-network."""
        next_q_values = q_network(states).gather(dim=1, index=actions)
        q_values = rewards + (gamma * next_q_values * (1 - dones))
        return q_values

    def q_learning_update(self, states,rewards,dones,gamma,q_network):
        """Q-Learning update with explicitly decoupled action selection and evaluation steps."""
        actions = self.select_greedy_actions(states, q_network)
        q_values = self.evaluate_selected_actions(states, actions, rewards, dones, gamma, q_network)
        return q_values

    def double_q_learning_update(self, states,rewards,dones,gamma,q_network1, q_network2):
        """Q-Learning update with explicitly decoupled action selection and evaluation steps."""
        actions = self.select_greedy_actions(states, q_network1)
        q_values = self.evaluate_selected_actions(states, actions, rewards, dones, gamma, q_network2)
        return q_values

    def learn(self, experiences, is_weights, idxs):
        """Update the agent's state based on a collection of recent experiences."""
        states, actions, rewards, next_states, dones = (torch.Tensor(np.array(vs)).to(device) for vs in zip(*experiences))

        actions = (actions.long()).unsqueeze(dim=1)
        rewards = rewards.unsqueeze(dim=1)
        dones = dones.unsqueeze(dim=1)

        if self.double_dqn:
            target_q_values = self.double_q_learning_update(next_states,rewards, dones,self.GAMMA,self.policy_net,
                                                            self.target_net)
        else:
            target_q_values = self.q_learning_update(next_states,rewards,dones,self.GAMMA,self.target_net)
        online_q_values = (self.policy_net(states).gather(dim=1, index=actions))
        losses = F.mse_loss(online_q_values, target_q_values, reduction='none')
        td_errors = torch.sqrt(losses)  # used for PER
        is_weights_tensor = torch.tensor(np.array(is_weights), dtype=torch.float32, device=device)
        weighted_losses = losses * is_weights_tensor  # Apply IS weights
        loss = weighted_losses.mean()
       # updates the parameters of the online network
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if self.replay.use_per:
            self.replay.update_priority(idxs, td_errors.cpu().detach().numpy()) #necessary?


    def step(self, state, action, reward, next_state, done):
        experience = Transition(state, action, reward, next_state, done)
        self.replay.push(experience)
        if not done:
            self.number_timesteps += 1
            # every so often the agent should learn from experiences
            if self.number_timesteps % self.update_frequency == 0 and self.has_sufficient_experience():

                batch, idxs, is_weights = self.replay.sample(self.batch_size)
                self.learn(experiences=batch, is_weights=is_weights, idxs=idxs)

            if self.number_timesteps % self.update_target_frequency == 0:
                self.target_net.load_state_dict(self.policy_net.state_dict())

    def train_for_at_most(self):
        """Train agent for a maximum number of timesteps."""
        state, info = self.env.reset()
        state, _, _, _, _ = self.env.step(1)

        self.number_lives = 5
        score = 0
        done = False
        episode_timestep = 0
       # for t in range(self.max_timesteps):
        while not done:
            action = self.choose_action(state)
            # print(f"Action Dis: {action} Timestep: {episode_timestep}")
            # action_cont = self.discrete2cont_action(action)
            next_state, reward, done, truncated, info = self.env.step(action)
            reward = min(1, reward)
            if info.get("lives") < self.number_lives:
                self.number_lives = info.get("lives")
                self.step(state, action, reward, next_state, True)
                next_state, _, _, _, _ = self.env.step(1)

            else:
                self.step(state, action, reward, next_state, done)
            self.epsilon = np.interp(self.number_timesteps, [0, 500000], [1, 0.01])
            episode_timestep +=1
            state = next_state
            score += reward
            if done or truncated:
                print(f"Episode: {self.number_episodes} Timesteps {episode_timestep}  Died :(")
                self.number_episodes += 1
                self.video = []
                break
        if self.number_episodes % 200 == 0:
            print(f"Episode: {self.number_episodes} finished in {episode_timestep} timesteps score: {score}")
            with open('prints.txt', 'a') as f:
                f.write(f"\nEpisode: {self.number_episodes} finished in {episode_timestep} timesteps score: {score}")
        return score

    def train(self):
        scores = []
        target_score = float("inf")
        most_recent_scores = deque(maxlen=100)
        best_score = float("-inf")
        self.policy_net.train()
        self.target_net.train()
        with open('prints.txt', 'w') as f:
            f.write("Starting prints")
        for i in range(self.max_episodes):
            score = self.train_for_at_most()
            logger.log({'Score': score})
            scores.append(score)
            most_recent_scores.append(score)
            average_score = np.mean(most_recent_scores)
            logger.log({'Mean Score 100 Episodes': average_score})

            if average_score >= target_score or self.number_timesteps >= 4000000: # 3 million episode limit
                print(f"\nEnvironment solved in {i:d} episodes!\tAverage Score: {average_score:.2f}")
                checkpoint_filepath = f"rl_chk/new-dqn-per-checkpoint{self.number_episodes}.pth"
                os.makedirs(os.path.dirname(checkpoint_filepath), exist_ok=True)
                self.save(checkpoint_filepath)
                break
            elif average_score > best_score:
                best_score = average_score
                plt.plot(average_score)
                plt.savefig("rewards.png")
                with open('prints.txt', 'a') as f:
                    f.write("\nSaving checkpoint")
                print("Saving checkpoint")
                checkpoint_filepath = f"rl_chk/new-dqn-per-checkpoint_4mil.pth"
                self.save(checkpoint_filepath)
            if (i + 1) % 100 == 0:
                plt.plot(scores)
                plt.savefig("rewards.png")
                with open('prints.txt', 'a') as f:
                    f.write(f"\n\rEpisode: {i + 1}\tAverage Score: {average_score:.2f} Epsilon: {self.epsilon} N_Frames: {self.number_timesteps}")
                print(f"\rEpisode: {i + 1}\tAverage Score: {average_score:.2f} Epsilon: {self.epsilon} N_Frames: {self.number_timesteps}")

        return scores


def Preprocessing_env(env):

    env = gym.wrappers.AtariPreprocessing(env, noop_max=30,
                                         screen_size=84, terminal_on_life_loss=False,
                                         grayscale_obs=True, grayscale_newaxis=False, scale_obs=False)

    env = gym.wrappers.FrameStack(env, 4)
    return env

if "main":
    # env = gym.make('CartPole-v1', render_mode="rgb_array")
    env = gym.make("BreakoutNoFrameskip-v4", render_mode="rgb_array")
    # env = gym.make('Hopper-v4')
    env = Preprocessing_env(env)

    wandb_logger = Logger(
        f"PER-DQN-New",
        project='INM707-Task2')
    logger = wandb_logger.get_logger()

    dqn = Agent(env, per=True, double=False, logger = logger)
    scores = dqn.train()
```

```python
    plt.plot(scores)
    plt.savefig("rewards.png")
    # plt.show()
```

Listing 4: DQN and extensions code

D.2.3 dqn_inference.py

```python
import gymnasium as gym
from gymnasium.utils.save_video import save_video

import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class DQN(nn.Module):

    def __init__(self, n_observations, n_actions, hidden_units=512):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, hidden_units)
        self.layer2 = nn.Linear(hidden_units, hidden_units)
        self.layer3 = nn.Linear(hidden_units, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)


class DQNCNN(nn.Module): # DQN/DDQN
    def __init__(self, input_shape, n_actions, hidden_units=512):
        super(DQNCNN, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        conv_out_size = self.get_conv_out_size(input_shape)

        self.value = nn.Sequential(
            nn.Linear(conv_out_size, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, n_actions)
        )

    def get_conv_out_size(self, shape):
        conv_size = self.conv(torch.zeros(1, *shape))
        return int(np.prod(conv_size.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        return self.value(conv_out)


class Agent:
    def __init__(self, env, per=False, double=False, logger=None):
        self.logger = logger
        self.max_timesteps = 5000
        self.number_timesteps = 0
        self.number_episodes = 0
        self.epsilon = 1
        # Get number of actions from gym action space
        self.env = env
        self.n_actions = 4
        self.number_lives = 5
        # self.n_actions = env.action_space.shape[0]
        # num_bins = 61  # Number of bins for each action dimension
        # self.n_actions = num_bins ** self.n_actions
        print(self.n_actions)
        print(f"Number actions: {self.n_actions}")
        seed = None
        self.random_state = np.random.RandomState() if seed is None else np.random.RandomState(seed)

        # Get the number of state observations
        self.state, self.info = env.reset()
        print(f"State shape: {self.state.shape}")
        # self.n_observations = len(self.state)
        self.n_observations = self.state.shape
        checkpoint = torch.load(f"rl_chk/dqn-per-checkpoint_4mil.pth")
        self.policy_net = DQNCNN(self.n_observations, self.n_actions, hidden_units=512).to(device)
        self.policy_net.load_state_dict(checkpoint['q-network-state'])

        print(self.n_observations)
        print(env.observation_space.shape)
        # self.policy_net = DQN(env.observation_space.shape, self.n_actions).to(device)
        # self.target_net = DQN(env.observation_space.shape, self.n_actions).to(device)

        self.video = []
    # def discrete2cont_action(self, action):
    #     # Map the discrete action index to continuous torques
    #     num_bins = 61
    #     action_indices = np.unravel_index(action, (num_bins, num_bins, num_bins))
    #     torque_min = -1.0
    #     torque_max = 1.0
    #     torques = [torque_min + (torque_max - torque_min) * idx / (num_bins - 1) for idx in action_indices]
    #     return np.array(torques)

    def choose_action(self, state):
        # need to reshape state array and convert to tensor
        state_tensor = (torch.from_numpy(np.array(state)).unsqueeze(dim=0).to(device)).float()
        action = self.epsilon_greedy_policy(state_tensor, self.epsilon)
        return action

    def epsilon_greedy_policy(self, state, epsilon):
        """With probability epsilon explore randomly; otherwise exploit knowledge optimally."""
        action = self.greedy_policy(state)
        return action

    def uniform_random_policy(self, state):
        """Choose an action uniformly at random."""
        # random_vector = np.random.(low=-1, high=1, size=self.n_actions)
        # return random_vector
        return self.random_state.randint(self.n_actions)

    def greedy_policy(self, state):
        # print(state.shape)
        # print(state.dtype)
        """Choose an action that maximizes the action_values given the current state."""
        action = (self.policy_net(state)
                  .argmax()
                  .cpu()  # action_values might reside on the GPU!
                  .item())
        return action

    def select_greedy_actions(self, states, q_network):
        _, actions = q_network(states).max(dim=1, keepdim=True)
        # print(actions)
        return actions

    def step(self, state, action, reward, next_state, done):
        if not done:
            self.number_timesteps += 1

    def train_for_at_most(self):
        """Train agent for a maximum number of timesteps."""
        state, self.info = self.env.reset()
        score = 0
        done = False
        episode_timestep = 0
        state, _, _, _, _ = self.env.step(1)
        self.policy_net.eval()
        with torch.no_grad():
            for t in range(self.max_timesteps):
        #     while not done:

                action = self.choose_action(state)
                next_state, reward, done, truncated, info = self.env.step(action)
                if info.get("lives") < self.number_lives:
                    self.number_lives = info.get("lives")
                    next_state, _, _, _, _ = self.env.step(1)

                self.video.append(self.env.render())
                self.step(state, action, reward, next_state, done)
                episode_timestep +=1
```

```python
                state = next_state
                score += reward
                if done or truncated:
                    print("GAME OVER!")
                    save_video(self.video, "videos", fps=25, name_prefix="video-inference")
                    self.number_episodes += 1
                    self.video = []
                    break
            print(f"Episode {self.number_episodes} finished in {episode_timestep} timesteps score: {score}")
            if not done:
                print("TOO LONG!")
                save_video(self.video, "videos", fps=25, name_prefix="video-inference")
                self.number_episodes += 1
                self.video = []
        return score

    def train(self):
        scores = []
        target_score = float("inf")
        most_recent_scores = deque(maxlen=100)
        score = self.train_for_at_most()
        scores.append(score)
        most_recent_scores.append(score)
        return scores


def Preprocessing_env(env):

    env = gym.wrappers.AtariPreprocessing(env, noop_max=30,
                                          screen_size=84, terminal_on_life_loss=False,
                                          grayscale_obs=True, grayscale_newaxis=False, scale_obs=False)

    env = gym.wrappers.FrameStack(env, 4)
    return env

if "main":
    # env = gym.make('CartPole-v1', render_mode="rgb_array")
    # env = gym.make('Hopper-v4', render_mode="rgb_array")
    env = gym.make("BreakoutNoFrameskip-v4", render_mode="rgb_array")
    env = Preprocessing_env(env)
    dqn = Agent(env, per=False, double=True)
    scores = dqn.train()
    # plt.plot(scores)
    # plt.savefig("rewards.png")
    # plt.show()
```

Listing 5: DQN inference

D.2.4 Individual Part

```python
import gymnasium as gym
from gymnasium.utils.save_video import save_video

import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import numpy as np
import os
from buffer import ReplayMemory
from logger import Logger

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device is {device}")

Transition = namedtuple('Transition',
                        ('state', 'action', 'reward', 'next_state', 'done'))

# os.environ['https_proxy'] = "http://hpc-proxy00.city.ac.uk:3128"


class DQN(nn.Module):

    def __init__(self, n_observations, n_actions, hidden_units=512):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, hidden_units)
        self.layer2 = nn.Linear(hidden_units, hidden_units)
        self.layer3 = nn.Linear(hidden_units, n_actions)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)


class DQNCNN(nn.Module): # DQN/DDQN
    def __init__(self, input_shape, n_actions, hidden_units=512):
        super(DQNCNN, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )
        conv_out_size = self.get_conv_out_size(input_shape)

        self.value = nn.Sequential(
            nn.Linear(conv_out_size, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, n_actions)
        )

    def get_conv_out_size(self, shape):
        conv_size = self.conv(torch.zeros(1, *shape))
        return int(np.prod(conv_size.size()))

    def forward(self, x):
        conv_out = self.conv(x).view(x.size()[0], -1)
        return self.value(conv_out)


class Agent:
    def __init__(self, env, per=False, double = False, logger = None):

        self.logger = logger
        self.GAMMA = 0.99
        self.LR = 1e-4
        self.ALPHA = 1
        self.update_frequency = 4
        self.update_target_frequency = 10000 # 20k for tuned ddqn
        self.batch_size = 64
        self.per = per
        self.double_dqn = double

        self.replay = ReplayMemory(100000, use_per=self.per)
        if self.per:
            self.alpha = self.replay.alpha
            self.sum_tree = self.replay.sum_tree
            self.max_priority = self.replay.max_priority
        self.memory = self.replay.memory

        self.max_episodes = 5000
        self.number_episodes = 0
        self.max_timesteps = 2000
        self.number_timesteps = 0
        self.epsilon = 1

        # Get number of actions from gym action space
        self.env = env
        self.n_actions = 4
        self.number_lives = 5
        # self.n_actions = env.action_space.shape[0]
        # num_bins = 61  # Number of bins for each action dimension
        # self.n_actions = num_bins ** self.n_actions
        print(self.n_actions)
        print(f"Number actions: {self.n_actions}")
        seed = None
        self.random_state = np.random.RandomState() if seed is None else np.random.RandomState
            (seed)

        # Get the number of state observations
        self.state, self.info = env.reset()
        print(f"State shape: {self.state.shape}")
        # self.n_observations = len(self.state)
        self.n_observations = self.state.shape
        self.policy_net = DQNCNN(self.n_observations, self.n_actions, hidden_units=512).to(
            device)
        self.target_net = DQNCNN(self.n_observations, self.n_actions, hidden_units=512).to(
            device)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.optimizer = optim.AdamW(self.policy_net.parameters(), lr=self.LR, amsgrad=True)
        print(self.n_observations)
        print(env.observation_space.shape)
        # self.policy_net = DQN(env.observation_space.shape, self.n_actions).to(device)
        # self.target_net = DQN(env.observation_space.shape, self.n_actions).to(device)

        self.video = []

    # def discrete2cont_action(self, action):
    #     # Map the discrete action index to continuous torques
    #     num_bins = 61
    #     action_indices = np.unravel_index(action, (num_bins, num_bins, num_bins))
    #     torque_min = -1.0
    #     torque_max = 1.0
    #     torques = [torque_min + (torque_max - torque_min) * idx / (num_bins - 1) for idx in action_indices]
    #     return np.array(torques)

    def has_sufficient_experience(self):
        """True if agent has enough experience to train on a batch of samples; False otherwise."""
        # return len(self.memory) >= self.batch_size
        if len(self.memory) == 5000:
            print("Sufficient experience recently obtained!!!")
        return len(self.memory) >= 5000

    def has_full_experience(self):
        """True if agent has enough experience to train on a batch of samples; False otherwise."""
        # return len(self.memory) >= self.batch_size
        if len(self.memory) == 100000:
            return len(self.memory) >= 100000

    def save(self, filepath):
        checkpoint = {
            "q-network-state": self.policy_net.state_dict(),
            "optimizer-state": self.optimizer.state_dict(),
        }
        torch.save(checkpoint, filepath)

    def choose_action(self, state):
        # print(state.shape)
        # need to reshape state array and convert to tensor
        state_tensor = (torch.from_numpy(np.array(state)).unsqueeze(dim=0).to(device)).float()
        # choose uniform at random if agent has insufficient experience
        if not self.has_sufficient_experience():
            action = self.uniform_random_policy(state_tensor)
        else:
            # print("Sufficient experience")
            action = self.epsilon_greedy_policy(state_tensor, self.epsilon)
        return action

    def epsilon_greedy_policy(self, state, epsilon):
        """With probability epsilon explore randomly; otherwise exploit knowledge optimally."""
        if self.random_state.random() < epsilon:
            action = self.uniform_random_policy(state)
        else:
            action = self.greedy_policy(state)
        return action

    def uniform_random_policy(self, state):
        """Choose an action uniformly at random."""
        # random_vector = np.random.(low=-1, high=1, size=self.n_actions)
        # return random_vector
        return self.random_state.randint(self.n_actions)

    def greedy_policy(self, state):
        # print(state.shape)
        # print(state.dtype)
        """Choose an action that maximizes the action_values given the current state."""
        action = (self.policy_net(state)
                  .argmax()
                  .cpu()  # action_values might reside on the GPU!
                  .item())
        return action

    def select_greedy_actions(self, states, q_network):
        _, actions = q_network(states).max(dim=1, keepdim=True)
        # print(actions)
        return actions

    def evaluate_selected_actions(self, states,actions,rewards,dones,gamma,q_network):
        """Compute the Q-values by evaluating the actions given the current states and Q-network."""
        next_q_values = q_network(states).gather(dim=1, index=actions)
        q_values = rewards + (gamma * next_q_values * (1 - dones))
        return q_values

    def q_learning_update(self, states,rewards,dones,gamma,q_network):
        """Q-Learning update with explicitly decoupled action selection and evaluation steps."""
        actions = self.select_greedy_actions(states, q_network)
        q_values = self.evaluate_selected_actions(states, actions, rewards, dones, gamma, q_network)
        return q_values

    def double_q_learning_update(self, states,rewards,dones,gamma,q_network1, q_network2):
        """Q-Learning update with explicitly decoupled action selection and evaluation steps."""
        actions = self.select_greedy_actions(states, q_network1)
        q_values = self.evaluate_selected_actions(states, actions, rewards, dones, gamma, q_network2)
        return q_values

    def learn(self, experiences, is_weights, idxs):
        """Update the agent's state based on a collection of recent experiences."""
        states, actions, rewards, next_states, dones = (torch.Tensor(np.array(vs)).to(device) for vs in zip(*experiences))

        actions = (actions.long()).unsqueeze(dim=1)
        rewards = rewards.unsqueeze(dim=1)
        dones = dones.unsqueeze(dim=1)

        if self.double_dqn:
            target_q_values = self.double_q_learning_update(next_states,rewards, dones,self.GAMMA,self.policy_net,
                                                                              self.target_net)
        else:
            target_q_values = self.q_learning_update(next_states,rewards,dones,self.GAMMA,self.target_net)
        online_q_values = (self.policy_net(states).gather(dim=1, index=actions))
        losses = F.mse_loss(online_q_values, target_q_values, reduction='none')
        td_errors = torch.sqrt(losses)  # used for PER
        is_weights_tensor = torch.tensor(np.array(is_weights), dtype=torch.float32, device=device)
        weighted_losses = losses * is_weights_tensor  # Apply IS weights
        loss = weighted_losses.mean()
       # updates the parameters of the online network
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if self.replay.use_per:
            self.replay.update_priority(idxs, td_errors.cpu().detach().numpy()) #necessary?


    def step(self, state, action, reward, next_state, done):
        experience = Transition(state, action, reward, next_state, done)
        self.replay.push(experience)
        if not done:
            self.number_timesteps += 1
            # every so often the agent should learn from experiences
            if self.number_timesteps % self.update_frequency == 0 and self.has_sufficient_experience():

                batch, idxs, is_weights = self.replay.sample(self.batch_size)
                self.learn(experiences=batch, is_weights=is_weights, idxs=idxs)

            if self.number_timesteps % self.update_target_frequency == 0:
                self.target_net.load_state_dict(self.policy_net.state_dict())

    def train_for_at_most(self):
        """Train agent for a maximum number of timesteps."""
        state, info = self.env.reset()
        state, _, _, _, _ = self.env.step(1)

        self.number_lives = 5
        score = 0
        done = False
        episode_timestep = 0
       # for t in range(self.max_timesteps):
        while not done:
            action = self.choose_action(state)
            # print(f"Action Dis: {action} Timestep: {episode_timestep}")
            # action_cont = self.discrete2cont_action(action)
            next_state, reward, done, truncated, info = self.env.step(action)
            reward = min(1, reward)
            if info.get("lives") < self.number_lives:
                self.number_lives = info.get("lives")
                self.step(state, action, reward, next_state, True)
                next_state, _, _, _, _ = self.env.step(1)

            else:
                self.step(state, action, reward, next_state, done)
            self.epsilon = np.interp(self.number_timesteps, [0, 500000], [1, 0.01])
            episode_timestep +=1
            state = next_state
            score += reward
            if done or truncated:
                print(f"Episode: {self.number_episodes} Timesteps {episode_timestep}  Died :(")
                self.number_episodes += 1
                self.video = []
                break
        if self.number_episodes % 200 == 0:
            print(f"Episode: {self.number_episodes} finished in {episode_timestep} timesteps score: {score}")
            with open('prints.txt', 'a') as f:
                f.write(f"\nEpisode: {self.number_episodes} finished in {episode_timestep} timesteps score: {score}")
        return score

    def train(self):
        scores = []
        target_score = float("inf")
        most_recent_scores = deque(maxlen=100)
        best_score = float("-inf")
        self.policy_net.train()
        self.target_net.train()
        with open('prints.txt', 'w') as f:
            f.write("Starting prints")
        for i in range(self.max_episodes):
            score = self.train_for_at_most()
            logger.log({'Score': score})
            scores.append(score)
            most_recent_scores.append(score)
            average_score = np.mean(most_recent_scores)
            logger.log({'Mean Score 100 Episodes': average_score})

            if average_score >= target_score or self.number_timesteps >= 4000000: # 3 million episode limit
                print(f"\nEnvironment solved in {i:d} episodes!\tAverage Score: {average_score:.2f}")
                checkpoint_filepath = f"rl_chk/new-dqn-per-checkpoint{self.number_episodes}.pth"
                os.makedirs(os.path.dirname(checkpoint_filepath), exist_ok=True)
                self.save(checkpoint_filepath)
                break
            elif average_score > best_score:
                best_score = average_score
                plt.plot(average_score)
                plt.savefig("rewards.png")
                with open('prints.txt', 'a') as f:
                    f.write("\nSaving checkpoint")
                print("Saving checkpoint")
                checkpoint_filepath = f"rl_chk/new-dqn-per-checkpoint_4mil.pth"
                self.save(checkpoint_filepath)
            if (i + 1) % 100 == 0:
                plt.plot(scores)
                plt.savefig("rewards.png")
                with open('prints.txt', 'a') as f:
                    f.write(f"\n\rEpisode: {i + 1}\tAverage Score: {average_score:.2f} Epsilon: {self.epsilon} N_Frames: {self.number_timesteps}")
                print(f"\rEpisode: {i + 1}\tAverage Score: {average_score:.2f} Epsilon: {self.epsilon} N_Frames: {self.number_timesteps}")

        return scores


def Preprocessing_env(env):

    env = gym.wrappers.AtariPreprocessing(env, noop_max=30,
                                          screen_size=84, terminal_on_life_loss=False,
                                          grayscale_obs=True, grayscale_newaxis=False, scale_obs=False)

    env = gym.wrappers.FrameStack(env, 4)
    return env

if "main":
    env = gym.make("AtlantisNoFrameskip-v4", render_mode="rgb_array")
    env = Preprocessing_env(env)

    wandb_logger = Logger(
        f"logger-DQN-Atlantis",
        project='INM707-Task2')
    logger = wandb_logger.get_logger()

    dqn = Agent(env, per=True, double=False, logger = logger)
    scores = dqn.train()
    plt.plot(scores)
    plt.savefig("rewards.png")
    # plt.show()
```

Listing 6: Individual Part: dqn.py