Reinforcement Learning and Dynamic Optimization,
Report on Poker Playing Agent Project : Second Assignment

Group 18:
Leonidas Bakopoulos AM 2018030036
Alexandra Tsipouraki AM 2018030089

1.Introduction

In our previous class project, we confronted the challenge of crafting a poker-playing agent by simplifying rules and shrinking significantly its otherwise overwhelming state space. However, to tackle the true magnitude of poker's intricate state space, considering a full deck of cards and all poker rules applied in a head's up poker game, we turn to the alliance of reinforcement learning and, specifically, to the Deep Q-Network (DQN) algorithm, a pioneering advancement that has enabled machines to learn optimal strategies in complex and dynamic environments.

2.Environment Implementation

As opposed to our first poker agent's simplified approach, here, we are dealing with a 'semi-realistic' poker game agent implementation; with the word realistic referring to the deck size (full deck, all suits), the game rules (winning combinations) and in general to the complexity of our problem and the word 'semi', referring mostly to our betting size. Considering all of the above, we decided that the Limit Hold'em environment found in RL Card's github repository is ideal, since it develops exactly what we need; a head-to-head limited betting poker game.In order to fully understand the given implementation, we first examined the Github repository files: rlcard/examples/run_random.py and rlcard/examples/run_rl.py . Our code has a total of six main files, namely : agent.py, main.py, network.py, other_agents.py and replay_buffer.py. As part of our environment we considered the RandomAgent class that was imported from RL card's limit hold'em, including the functions that all agents should inherit to work properly in our environment. Based on these functions, two Threshold_Agent classes were implemented, one defensive (Tight) and one loose, as "strong" opponents against our DQN Agent. In order to ensure the strength of our opponents against the agent, a rule based already implemented opponent was used. In summary, our threshold agents implement a simple (threshold-based) agent who makes decisions based on the strength of its hand and the cards on the table. It follows predefined rules to determine whether to raise or call based on its evaluation of its hand's strength, demonstrating a basic decision-making strategy used by poker-playing agents. Finally, a combination of the DQN agents (trained against the threshold one created by us and the already implemented one by RL card) was tested against a human player. More detailed information on the threshold opponents can be found in the 'appendix' section of the report.

3. Algorithm Implementation

Moving to the main part of this project, let's first discuss DQN. At its core, DQN is a deep reinforcement learning algorithm designed to navigate through large and complex state spaces and learn optimal policies for decision-making under uncertainty. The fundamental structure of a DQN has two key components; the neural network architecture and the experience replay. The neural network serves as an approximation of the Q-function, which quantifies the expected cumulative rewards for each possible action in a given state. This function approximator allows DQN to generalise its decisions across the expansive state space, enabling effective decision-making even in states that weren't explicitly encountered during training. Furthermore, the addition of experience replay, where past experiences are stored and sampled, moderates issues related to events happening one after another in our sequence. This makes for a more robust and efficient learning process. In our implementation, we didn't use the traditional single-network DQN. Instead, we employed a variant (called Double Q Learning) that incorporates both a model and a target network. The model network is continuously trained, while the target network is consistently updated by the temperature factor (τ * model_weights). This update process involves passing values (more specifically a percentage of the model's weight, proportional to the τ factor) to the target network, allowing us to calculate the Q-values of state s and state s' with time differentiation.

Let's first emphasise on the basic parts of our code. In the Agent class, two main functions were implemented, namely 'eval_step()' and 'step()'. These functions are mandatory as they are invoked by the RL Card environment which we are using. The 'eval_step()' function is responsible for sending the best 'legal' action to be taken during the evaluation/testing phase, while 'step()' is used during the training process, following an epsilon-greedy logic.

Another fundamental function is 'agent_step()', which is called in the main part of the code. It pushes the experience into the replay buffer, effectively storing it in memory. Subsequently, it samples from the replay buffer and initiates the training process. It involves updating the neural network's parameters using the sampled experiences to improve the agent's performance over time.

As for the training process, it's performed as described in the official DQN paper, with the only different being that our network may suggest an optimal move which isn't allowed to be used. During training, the agent's neural network learns to approximate the optimal action-value function by minimising the difference between the predicted Q-values. The target Q-values are calculated using the Bellman equation, incorporating the reward received and estimated Q-value of the next state (TD-error).

Despite the network's ability to provide suggestions our responsibility lies in ensuring that the chosen action adheres to the game constraints. This emphasises the need to consider game rules and valid actions when making decisions. As a result, the training process involves updating the neural network based on experiences collected during gameplay, while adhering to the logic of selecting the best permissible action in all cases.

In the context of reinforcement learning and training agents using experiences, a replay buffer is a crucial component. It stores past experiences (state, action, reward, next state, done) of an agent's interactions with an environment. The replay buffer serves as a memory that allows an agent to sample (unoformely, in its simpler implementation) and learn from these experiences in a more efficient manner, separating the learning process from the agent's current interactions with the environment. Prioritized Experience Replay is a technique that assigns different priorities to experiences in the

replay buffer to improve learning efficiency and the effectiveness of the agent's training. In our implementation we first created a simple ReplayBuffer class without prioritization and then PriorizedExperienceReplay class came to extend the basic replay buffer, having additional methods and attributes for managing priorities. After initialising the prioritized experience replay buffer with the provided parameters, we included two hyperparameters: `alpha` and `beta`. `alpha` controlled the prioritization strength, and `beta` the importance-sampling correction. Then, with the method 'add()' the basic replay buffer's `add` method was extended . This method additionally updates the priority of the new experience, setting it to the maximum value, in order to ensure that this particular tuple will be selected at least once. As for the `sample()`method, this is the core of the prioritized replay mechanism. It assigns sampling probabilities for experiences based on their priorities and the importance-sampling weights. It uses the calculated probabilities to randomly select experiences.Lastly, the `update_priorities()` method , updates the priorities of the selected experiences in the replay buffer. It's used during the training process to adjust the priorities based on the TD error of the experiences. Experiences with higher TD errors are considered more informative ('surprising') and the agent gives them higher priority in the learning process. Finally, by assigning priorities to experiences and sampling based on these priorities, prioritized experience replay focuses the agent's learning on experiences that are more important for improving its performance.

Network's hidden layers (both target and model)	2	Memory size	10^5
Hidden layer's size	(512,256)	gamma	.99
Dropout	One layer 20%	τ	.005
Activation functions	Relu	Horizon	$1.5 * 10^6$
Batch size	64		

Table 1: Empirically tested chosen hyper-parameters of the NN and the MDP

Note: The hyperparameters of the table above, were selected empirically, after some experiments against the random opponent. The hyperparameters left (ex. Learning rate, exploration policy etc) were finally defined based on the results of the following experiments (see Experiments & Results).

4. Experiments & Results

After the coding phase came to an end, our next step involved conducting experiments to find the best hyperparameter tuning / best final implementation. The outcomes of each experiment are explained in the subsequent section of this report.

Note: In the following figures is shown the average cumulative reward per hand that the agent achieved in a tournament of 2000 hands playing against the random opponent. In these tournaments, the agent acted without any noise.

Experiment 1: Use of target network (or not)

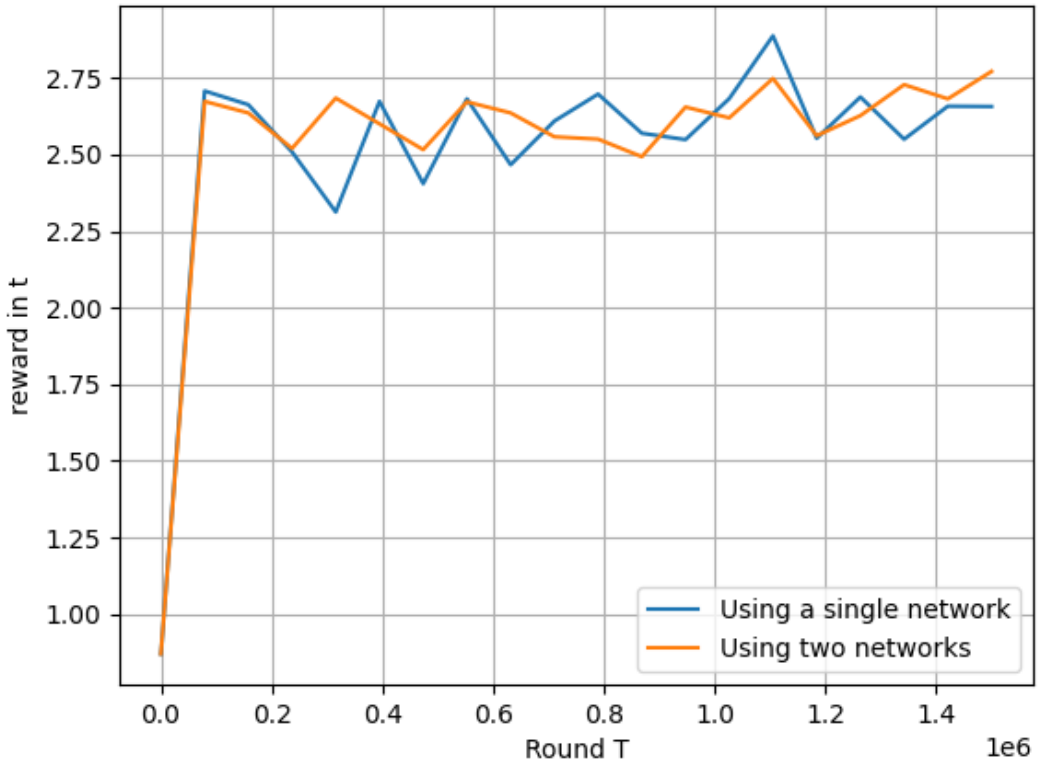


Figure 1 : Use (or not) of the target network , rounds T are 200.000 and the evaluation process happened for each 10.000 episodes

In the above figure (figure 1) are plotted the cumulative rewards for both DQN implementations , with and without the use of the target network. As it has been explained in the algorithm creation section, the addition of a second network increases the stability in learning , improving the exploration-exploitation trade-off and ultimately leading to more efficient learning in a challenging and complex environment like poker. In the above figure, the use of the target network doesn't seem to yield such improved results compared to single network implementation. Nonetheless, the results of the double network still are positive and slightly better than the single network approach, hence, based on theory, we continued our experiments using the double network implementation.

Experiment 2: Epsilon value

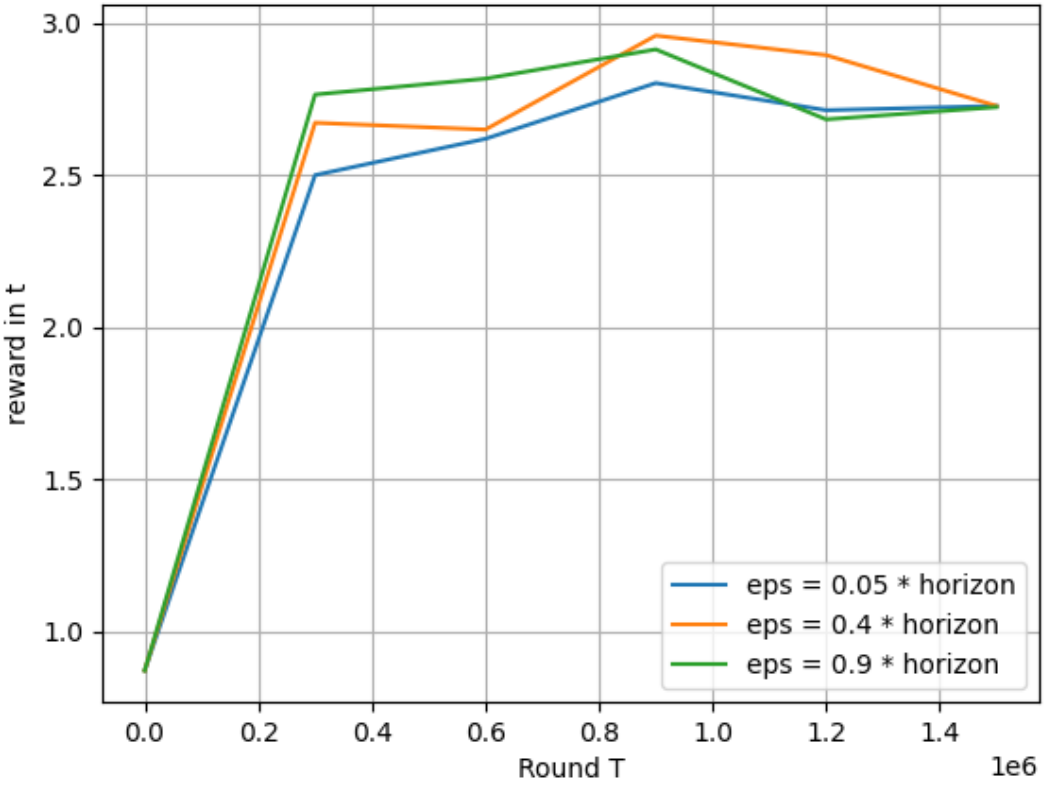


Figure 2: Epsilon parameter value (T=1_500_000 and evaluate_every=250_000)

The chart presented above illustrates the frequency at which exploration occurs as a percentage of total rounds. The epsilon parameter is systematically varied across the range of 1 to 0.1 to ensure that exploration remains a part of the learning process, as setting it to 0 could potentially lead to issues related to overfitting. It can be seen that choosing an epsilon parameter that explores 40% of the horizon yielded the best results. This can be attributed to the balance it struck between exploration and exploitation in the learning process. Also, in a poker game it's crucial to avoid getting stuck in local optima. By maintaining a moderate level of exploration the agent occasionally deviates from its current policy to discover better strategies, providing robustness to the agent while also escaping potential local optima.

Experiment 3 : Learning rate parameter tuning

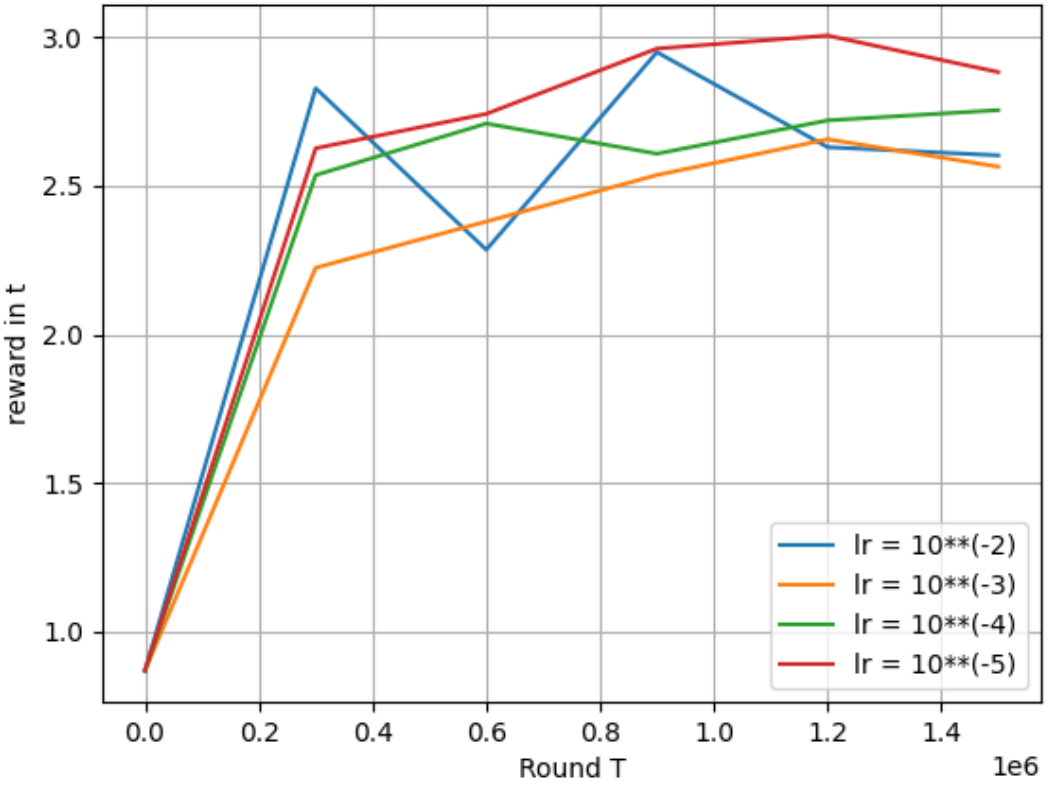


Figure 3: Learning rate parameter fine-tuning

Fine-tuning the learning rate (lr) is a common practice in reinforcement learning. Based on common practices and lr values we opted for learning rates varying from 10^(-2) to 10^(-5), as you can see in figure 2 above. A learning rate with a high value (e.g. 10^(-2)) can cause the training process to diverge with the model's parameters oscillating or overshooting the optimal values. On the other hand, much lower learning rates may converge very slowly. The choice of 10^(-5) could represent a good compromise between rapid convergence and stability. Also, a lower learning rate favours exploitation, where the agent relies more on its current knowledge, making it less likely to deviate from what it has learned. In our case, it seems that 10^(-5) strikes a good balance between stability and convergence speed. However, it should be noted that the lr value depends on various problem and network characteristics and therefore it's a valuable practice to experiment with different lr's to find the one that works best for each specific task.

Experiment 4 : Decreasing (or not) the learning rate after the end of exploration

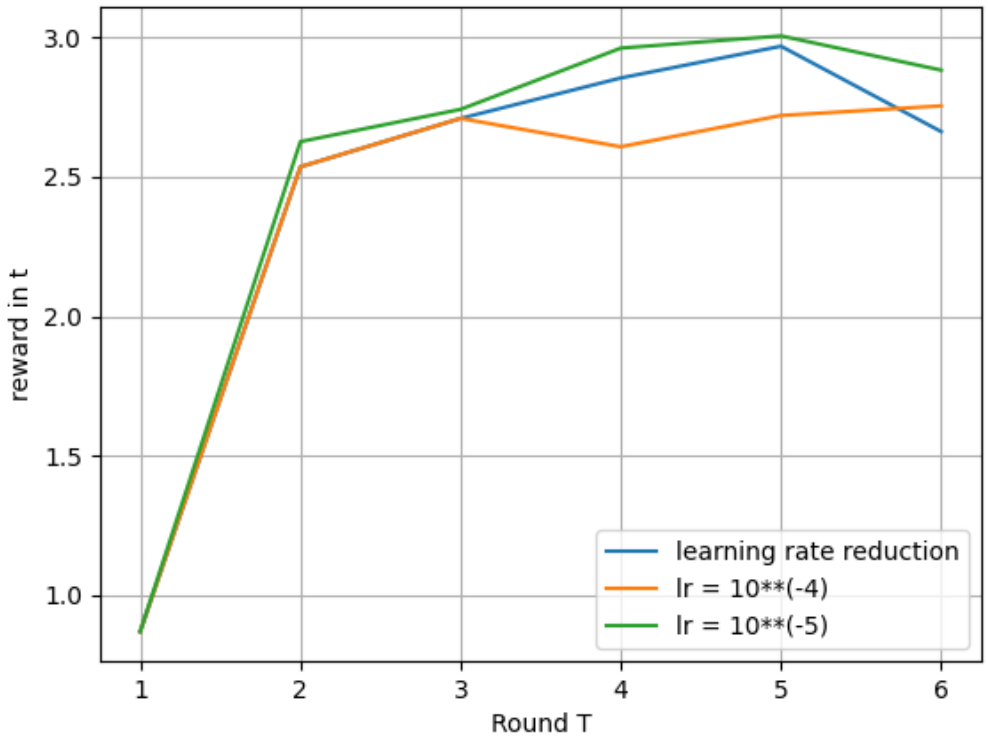


Figure 4: Is learning rate reduction helpful?

In order to proceed with our final, fine-tuned model we examined a possible learning rate reduction during training (after the termination of the exploration phase), which yielded negative results. Hence, the learning rate was tuned to a constant 10^(-5) value, and remained such throughout the rest of the experiments.

Experiment 5: Dropout

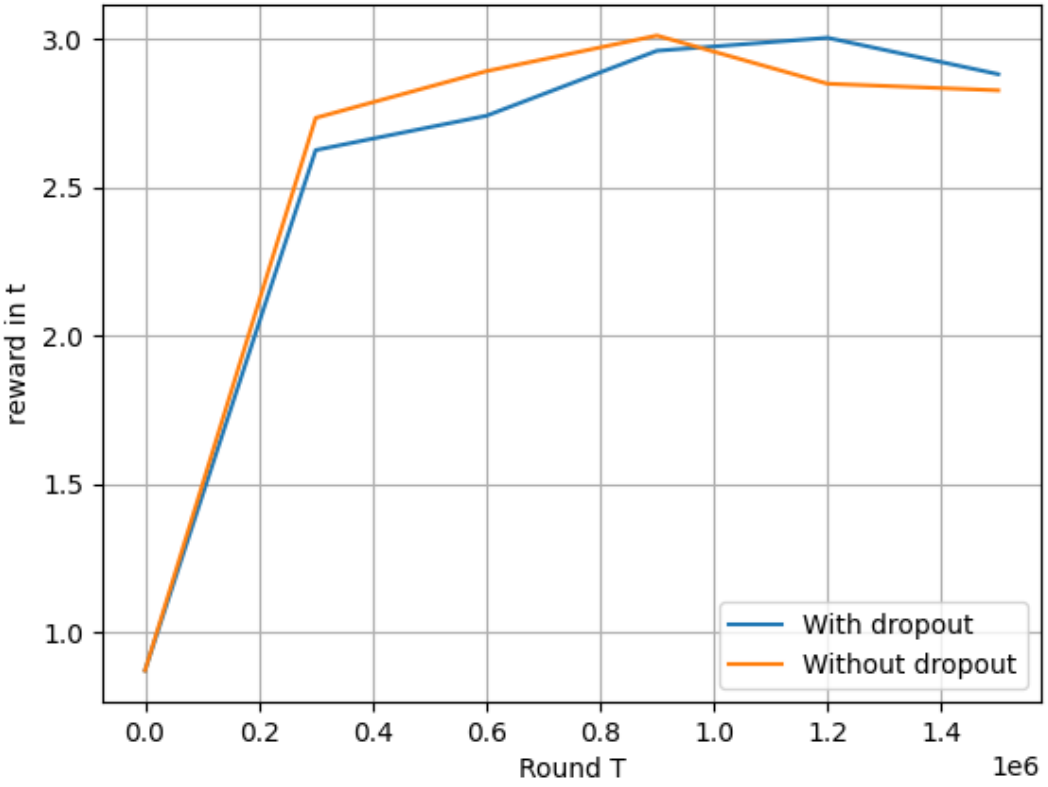


Figure 5: Could a dropout layer be used in DRL?

Last but not least, we examined the use of a dropout layer, in order to avoid overfit. The use of that layer, is not that common in DRL but it can help the agent, not overfit [1] in that small size of action space. As it is shown in the figure above, the agent is trained (slightly) better with the use of a dropout layer.

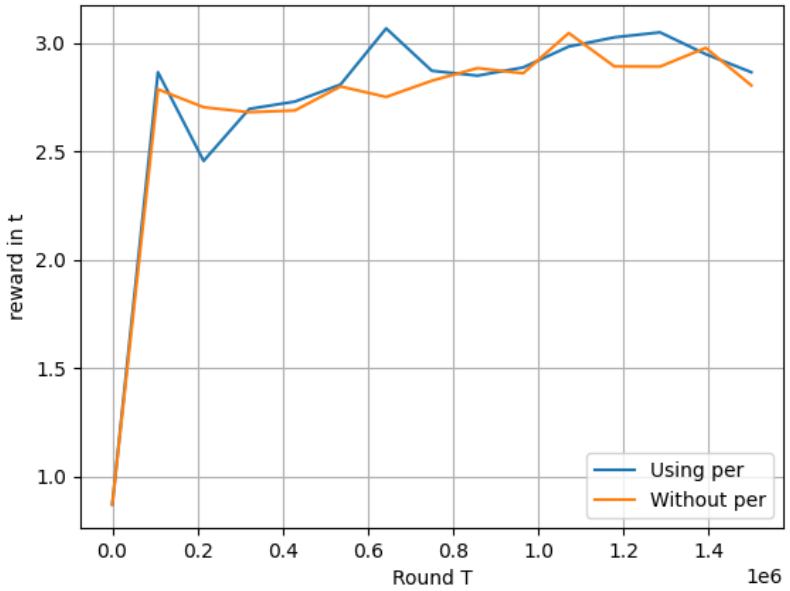
Network's hidden layers (both target and model)	2	Memory size	10^5
Hidden layer's size	(512,256)	gamma	.99
Dropout	One layer 20%	τ	.005
Activation functions	Relu	Horizon	$1.5 * 10^6$
Batch size	64	Exploration period	(5%, 40%, 90%) * horizon
Learning rate	10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}		

Table 2: Final hyperparameters of the agent and the MDP

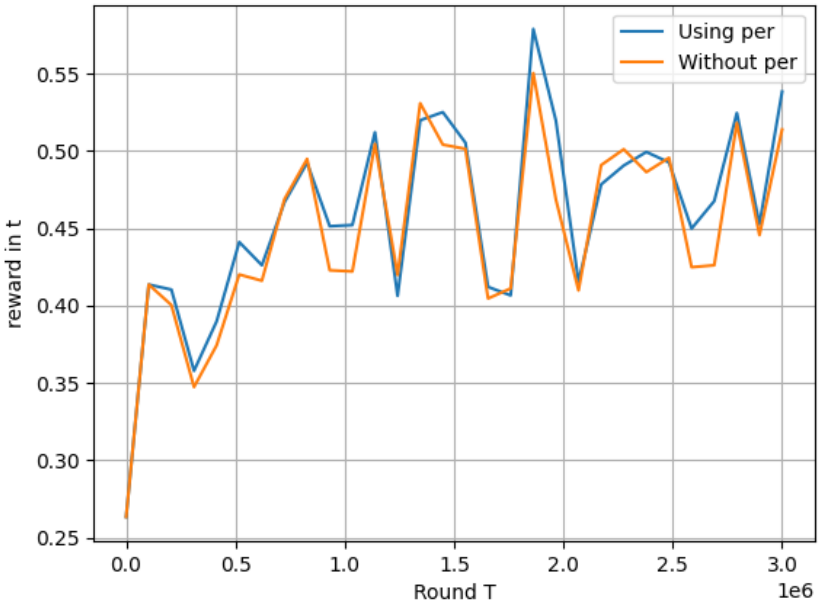
In table 2 are shown the aforementioned empirically chosen parameters with the tested ones being highlighted.

After the previous experiments, the agent should be evaluated against the two opponents, both threshold ones and the random opponent. In these final experiments/evaluations, both "random" replay memory and prioritized experience replay buffer will be used. The result of that comparison will be utilized to finally select the best agent settings. Below, will be found all final testing experiments.

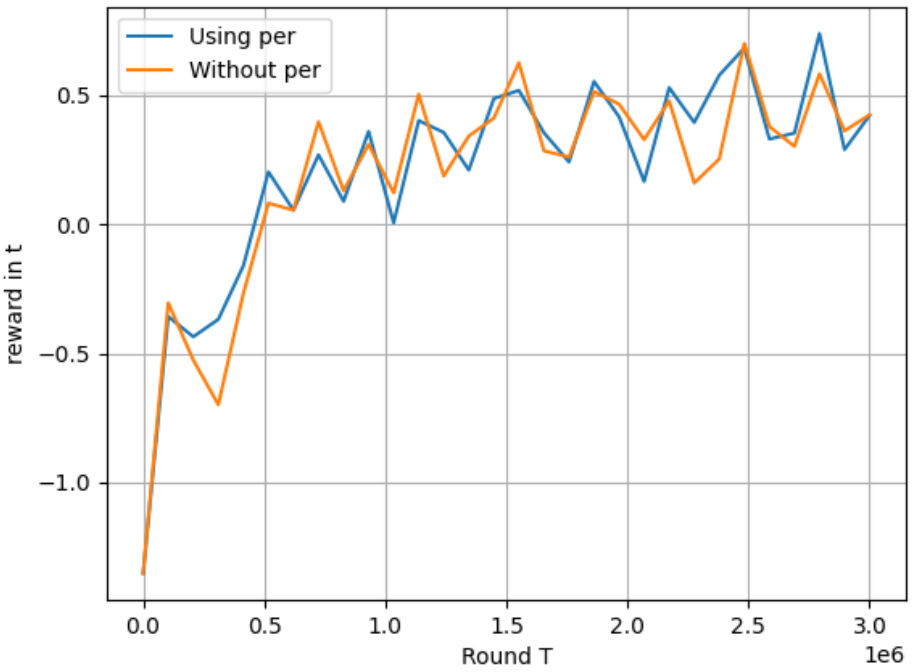
Final Testing: DQN against the Random agent with and without per:



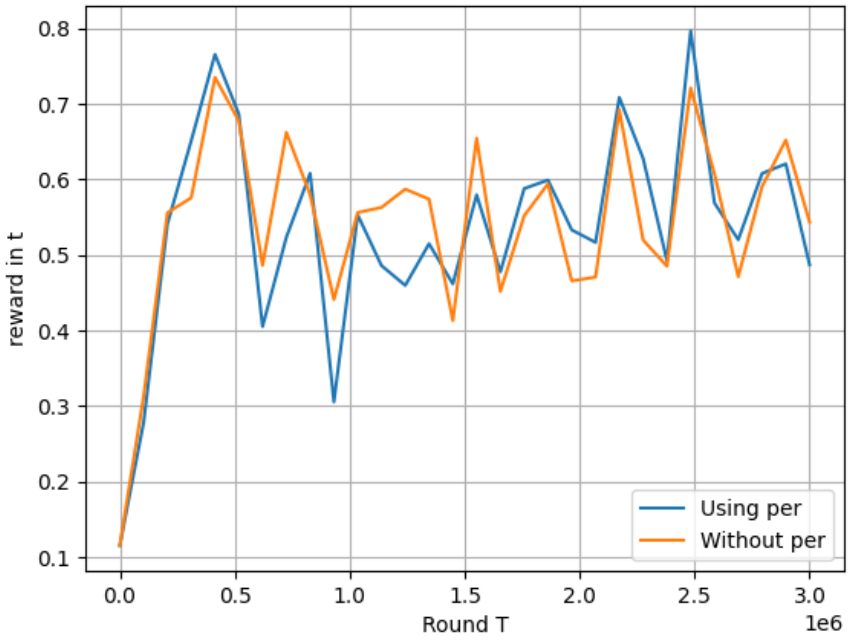
Final Testing: DQN against the (tight) Threshold agent with and without per:



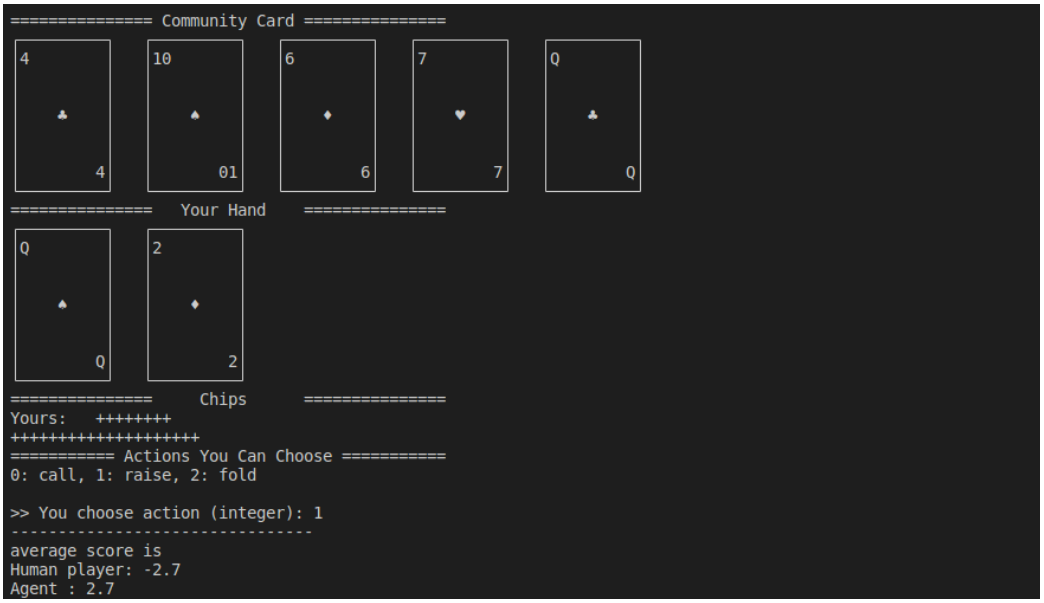
Final Testing: DQN against the (loose) Threshold agent with and without per:



Final Testing: DQN against the (best) Threshold agent with and without per:



Final Testing: mixed DQN against the human agent



For the final evaluation, a combination of the pre-trained DQN agents was created (see Appendix for more details) and tested against a human player. This evaluation revealed the policy of the agent. More precisely, the DQN agent, follows a more offensive approach, taking advantage of the small amount of available chips in every hand. This policy, leads the opponent (human) to fold even when having a hand with some potential. It's worth mentioning the fact that in the previous phase of the project, the tabular Q-learning agent was winning using the same technique.

5. Conclusions / Future work

To conclude, as it can be deducted from our experiments, the DQN agent has demonstrated its dominance over every opponent within the chosen environment. In the best-case scenario, it leads random opponents to bankruptcy, while even the rule-based adversaries consistently lose more than half the big blind in every hand. Moreover, within a confined action space, certain hyperparameters appear to have diminished significance, with the priority of exploration parameters like epsilon-greedy strategies diminishing as well. The use of an extra exploration technique, such as parameter noise for space exploration, could probably lead to the optimization of the learning process in more complex environments, but in this case it may be dispensable. However, relying solely on exploitation in the long run is an ill-advised strategy, as it can lead the agent to overfitting.

Furthermore, the necessity of a deep reinforcement learning algorithm in this environment is evident, as the complexity of the tasks renders the creation of a tabular Q-learning approach unfeasible. While it is tempting to consider applying the DQN in our older environment, compatibility issues may pose a substantial obstacle. It's also worth noting that the convergence speed of the DQN is influenced by numerous hyperparameters, making it essential to strike a balance between them for optimal performance. Hence, it’s not safe to compare our new implementation to the previous one since their hyper-parameter tuning differs, as does their environment. However, based on the agent’s performance, it’s very likely that the convergence speed when using the DQN agent in our previous setup will be faster than the tabular Q-learning agent’s.

Looking ahead, future work in this environment could explore the challenge of extending the state space to incorporate additional information such as total chips and opponents' chip totals, as well as maintaining an action record (here it should be noted that we tried to do it but we didn’t complete it due to software errors). These enhancements could potentially lead to further improvements in the agent's performance and open up new avenues for research and development.

6. ‘Extra’ References (except from the lectures)

[1] A study on overfitting in deep Reinforcement Learning by Zhang C,Vinyals O,Munos R,Bengio S 2018

7. Appendix

In order to test our agent against realistic opponents, two types of threshold agents were developed. Each player, in the preflop state, evaluates its hand and acts accordingly. In every other post flop state, (flop, turn, river) the policy of the agent depends wether it "hit" a pair or not. The policy of each threshold player depends on its playing attitude; the "tight" opponent, when having strong hand, just 'calls' (if it is legal), while in every other case 'checks'. On the other hand, the "loose" opponent acts aggressively with a strong hand (raises), and in every other case, checks.

For the agents to be compatible with the environment, their implementations should guarantee that they will send legal actions. For that to be achieved, an interface was created to map any illegal action to the closer legal one. For example, in case that the agent selects an illegal raise, the interface, tries to "send" the call action (if it is legal), then checks (if it is legal) and then folds. The goal of that mechanism, is to create a semi-realistic opponent that selects to fold only if it is necessary and not just because it is an (everytime) available move.

As you can imagine, the idea behind learning was introduced, considering that the implementation of an agent whose policy is based on a knowledge base (like a threshold agent), would be a difficult task. Although some simple threshold agents were implemented, the already implemented threshold agent called "LimitholdemRuleAgentV1" was used as the main opponent against our DQN-agent. This opponent, plays when only having strong preflop hands, and continues to bet in case of a high card, pair(or double pair), three of a kind and flush. The previously mentioned policy, doesn't differ significantly from an amateur player's.

In order to really test the DQN agent, an experiment between a mixed-trained DQN agent and a human player was performed. For the creation of the DQN agent, a linear combination of the pre-trained (using per) Q-network against the 'tight' threshold agent and the LimitholdemRuleAgentV1 were used. More precisely the new network consists of:

New_newtork = w_t * tight + w_b * best

Where, *w_t* equals to 0.4, *tight* are all the parameters of the Q Network against the tight opponent, *w_b* equals to 0.6 and *best* are the parameters of the Q Network against RL card's threshold opponent.