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

Group 18:
Leonidas Bakopoulos AM 2018030036
Alexandra Tsipouraki AM 2018030089

This report aims to provide a comprehensive analysis of our groups' implementation of the 'Poker Playing Agents' project, which focuses on developing reinforcement learning agents for poker playing purposes. Specifically, after a thorough research and study of the required environment and algorithms to be implemented, we began constructing our code based on RL Card's poker implementation 'Leduc Hold'em' and the instructions given in the project description. Specifically, our version of poker contains a deck of 20 cards; 5 different ranks (10,J,Q,K,A) and 4 suits. Each player is dealt one card in hand and 2 cards are set on the table. Winning is determined by all the classic poker rules that can be applied and followed in this simple game implementation.

Part A : Environment

Let's take a close look at the code implemented in order to model a proper poker-playing environment. First, we built the classes: Card, Agent, Dealer, Game and Judger, containing all the methods required for our basic game implementation.
The 'Card' class serves as a representation of a single playing card in a card game, like poker. The class encapsulates the card's rank and suit as instance variables, defining which ones are valid (e.g. 'S' for Spades and 'A' for Ace).
After successfully creating the cards, a dealer had to be introduced. Therefore, class 'Dealer' was created, being responsible for managing the deck of cards and facilitating its distribution to the agents (players). Like a real dealer does, in this class methods to shuffle the deck were implemented, ensuring randomness in the distribution, but also to deal cards from the deck whilst removing and returning every time the last card in the deck list. Now the need for a judger appeared, which we filled by creating the homonymous class 'Judger'. In this class, the essential judger is simulated, regarding the hands of the players. It consists of several methods that enable the comparison of hands and the distribution of the pot, when required, since there are cases of clear win or loss (when a player folds). Upon initialization, the 'Judger' object initialises a dictionary called 'rewards' that assigns specific reward values to different card ranks. The higher the rank, the higher was the reward of the card. It is worth mentioning that the judger's reward has nothing to do with the reward that the environment returns. The 'compare_hands()' method, evaluates the hands of the players by analysing the total value of cards in their hands and on the table, calculating a reward for each player. The method returns the index of the player with the highest reward or -1 in case of a tie. The 'split_pot()' method, determines how to distribute the pot based on the comparison of hands, utilising the previously mentioned compare_hands method to identify the winner and divides the pot accordingly.

After the implementation of the aforementioned classes, they were combined in the central class, 'Game'. This class serves as the core component for simulating the poker playing environment and managing the game flow for every single hand. It encompasses various attributes and methods essential for the gameplay. The number of players, the number of cards in hand and on the table, including the player's tokens and, in general, all the current information on the game is provided in this class' constructor. The 'init_game()' method is called at the beginning of every hand, resetting the game state accordingly. It shuffles the deck, deals cards to the players, sets initial bets and updates the pot. The 'step()' method is invoked when a player decides an action during their turn. It handles all available actions of the game, adjusting it accordingly. We modelled the available actions to be three : check, raise and fold.
At this point it should be mentioned that 'check' action also contains 'call' meaning that when the opponent raises, the player can reply by 'checking',meaning, that they bet the amount raised from the opponent, while at the same time the player can 'reraise' by choosing option 'raise'. The above merging of actions was mainly chosen in order to simplify the action handling process. Furthermore, 'game.step()' function, checks the termination of a phase and determines the winner of the hand, if

necessary. Passing on to the 'game.win()' method, it sets the winner of the hand and handles the pot distribution accordingly. The 'game.check_if_game_end()' method controls if the game should end based on the players' token balances. All methods are demonstrated in the main block of the class. After concluding the first, and simpler, part of our environment implementation, we moved on to the agents, first creating the more generic class 'Agent'. This is a crucial component of our poker-playing environment implementation as it represents individual poker-playing agents providing the framework for their decision-making process. It includes a constructor method for initialization, a 'send_action' method to determine the agent's action based on the current fame state , a 'to_str' method for generating a human-readable identifier, and a placeholder method 'reduce_a' for potential internal updates during the game. Subclasses will derive from this class to create the required agents who will follow specific behaviours and strategies, as required from the project.

To finally combine all of the above and start implementing our algorithms afterwards, class 'Env' was created, to simulate the complete environment. This class is responsible for coordinating the interactions between agents and the game. It is defined with an initializer method that takes an agent and an opponent as inputs, along with other parameters such as the seed and the number of cards, and initialises all required variables for the game. The 'reset()' method is responsible for resetting the environment at the beginning of each game (sequence of hands until one player will bankrupt), it updates the agent's and opponent's hands, sets up the ante, updates the bank and the last opponent move variables, which will be thoroughly explained in the next chapter of this report along with the various agent implementations and algorithms. It also returns the initial state, the current 'head of the game'/ 'player/dealer' ('μάνα' as it is simply called in greek) and a flag indicating whether the game terminated. Again the 'step()' method is used to perform an action (check,raise,fold) and it handles all the game flow based on the player, his chips and the action chosen. The same method also updates the table cards, forms the current state and once again checks if the game is over, acting accordingly in each case. The 'form_state()' method, creates the state representation of the environment, combining the agent's hand, the table cards, available chips and the last opponent's move into a state vector to be then returned. In utils.py file, there were implemented functions that map this vector to an enumerated state. Last but not least, in the main block an instance of the class is created with an agent and an opponent. A loop simulating the game is run, where the environment is reset and actions are decided by the agents, in the environment, and then performed by them and the opponent alternately. To summarise, the Env class provides the infrastructure for our poker-playing environment, including initialising the game, managing the players' actions, tracking the game states and determining the rewards.

Part B: Implemented Algorithms

After completing the environment, our focus shifted towards the algorithm implementations in order to create and train the performance of our agents. We created five kinds of agents; the Policy Iteration agent, the Q-learning agent, the Random agent and two Threshold agents, whose attributes and strategies will be discussed in detail below. All of the above agents inherit the methods created for the 'Agent' class,with the most important being Agent.send_action(), and implement the various needed algorithms.

First, let's discuss the **Policy Iteration algorithm**. Policy iteration is a model-based iterative algorithm used in reinforcement learning to determine the optimal policy for an agent. In the context of poker, policy iteration will be employed to offline 'train' an agent to make the optimal decisions during a game. More specifically, the PolicyIterationAgent iteratively evaluates and improves a policy until it converges to an optimal solution. The class contains methods for policy evaluation, policy improvement and the main policy iteration process. To specify the methods, we first find the '_init_' method , our class "constructor" where the various attributes P, epsilon, gamma, pi and v are initialised, as shown in the "Frozen Lake" example analysed in the course lectures. Epsilon is a small value used as a stopping criterion while gamma is the discount factor. The P attribute represents the transition probabilities (transition matrix) and it is implemented in a different file, namely utils.py or just a different cell in our google colab version of the code .

The P transition matrix deserves mentioning as it is the main component who is responsible for guiding our policy iteration agent's plays.

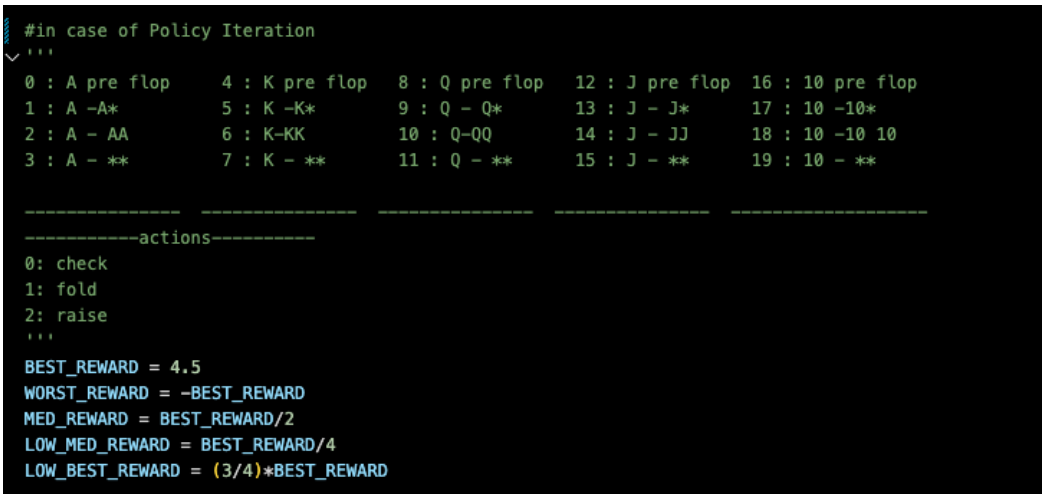


Figure 1 :Transition matrix for Policy Iteration against Random agent

Let's first discuss the state space chosen for the implementation. Since we were required to create a simple poker game using 5 different ranks of cards only, as human -and amateur- poker players, we decided to model the transition states by considering the following; our states represent different combinations of our own card and the table cards. We have chosen these states based on whether or not the combination is likely to win or lose. By defining the state space this way, we are considering the strength of our hand relative to the community cards on the table. To get into more detail, the state space contains five pre-flop and fifteen flop states. At the pre-flop states we consider Ace to be the best card, hence our agent considers higher winning probability and never decides to fold; similarly, when having a Ten pre-flop, we consider a smaller probability of winning so our agent can consider folding since the beginning. After the first round of betting, there is the flop stage where the two table cards are uncovered. After that, we created the states considering that the only cases who needed to be modelled differently were the ones where we found a pair or three of a kind, since in all other cases highest card wins, which we modelled as a state too. Hence, we narrowed the remaining states down to the fifteen post-flop you can see in the figure above. This approximation will probably help our agent converge at a faster rate since we diminished the complexity of every run of the game but it doesn't affect the game complexity since the full states wouldn't be significantly greater considering the fact that we were instructed to ignore the card suits. As for the rewards associated with each state- action pair, they reflect the expected outcomes of taking that action in a particular state. We have defined five different reward values, such as the best reward, worst reward, medium reward and two cases of low medium and low best reward.The final two represent the stages in between medium and worst or best accordingly. These rewards combined with the state space help guide the agent's decision making process by providing incentives for choosing certain actions in specific states. Another detail worth mentioning is that the transition probabilities from one state to another were given approximate values due to the complexity of the precise calculations caused by the abstraction level of our states. Concluding, by constructing a state space that captures the strength of our hand in relation to the table cards, we aim to develop a policy that maximises our chances of winning in different poker scenarios. The policy iteration algorithm can then iteratively improve this policy by evaluating and updating the value function and policy until convergence is reached. Returning now to the PolicyIterationAgent methods, the 'send_action()' function returns an action based on the current policy by calling the 'pi' function, which represents the policy. The state is passed as an argument to pi, which returns the corresponding action. Essentially, as we've already mentioned 'send_action()' is a convenience method for obtaining an action for a given state. The 'policy_evaluation' function calculates the value function V(s) for the current policy pi. V(s) represents the expected cumulative reward starting from a particular state and following the policy. The function initialises prev_V as an array of zeros with the same length as the environment's states. Then, in a loop, it updates v by iterating over all states and applying the Bellman equation which combines the immediate reward, discounted future rewards, and the cost-to-go from the next state. The algorithm iteratively updates v until the values converge, meaning that the change in v between iterations falls below a specified threshold, epsilon. The policy_improvement function, constructs an improved policy based on a given value function and the discount factor gamma. It initialises a q-value array Q with zeros, where each row corresponds to a state and each column represents an action. The function then iterates over all states and actions, calculating the Q-values by summing the

expected immediate reward and discounted future rewards for each possible transition from the current state-action pair. After computing all Q-values the function then creates a new policy ‘new_pi’ by selecting the action with the highest Q-value for each state. The resulting new_pi is a lambda function that maps each state to its corresponding optimal action. Finally, the function sets self.pi to the new policy and returns it.
The policy_iteration function performs the policy iteration process. It starts by initialising a random set of actions for each state and uses them to define the initial policy, pi. The function enters a loop where it keeps track of the previous policy and performs: policy evaluation, policy improvement and checks if the updated policy is the same as the previous policy. If they match, it means that the algorithm has converged and the loop is terminated. The function counts the number of iterations and then returns the converged value function ‘V’ and policy ‘pi’ as the final results.

In summary, in the class being described, the policy iteration algorithm is implemented, iteratively evaluating and improving policies until an optimal policy is found. The ‘policy_evaluation’ function calculates the value function for a given policy, the ‘policy_improvement’ function constructs an improved policy based on a value function, and the policy_iteration function combines these steps to find the optimal policy.
Now moving on to the **Q-learning agent implementation**, a class named Q_Learning_Agent was created. As for policy iteration, here the agent initialises its parameters in the ‘ __init__’ method, including the state size, action size, learning rate a, discount factor gamma and other optional arguments. The Q values are initialised randomly unless a pre-existing Q-table is provided (will be explained later).
The train method is responsible for updating the Q-values based on the training tuple. It first calculates the target Q-value using the Bellman equation, which incorporates the immediate reward and the maximum Q-value for the next state. The agent then updates the Q-value for the previous state-action pair using the learning rate a. If the values don’t change in between two consecutive iterations, a variable for convergence is incremented and if it reaches a threshold of 10.000, then the Q-values are printed and the algorithm is considered to be converging. The ‘send_action’ method selects an action based on the current state and eps-greedy algorithm. It gradually reduces the exploration rate (eps) to encourage more exploitation over time. If the agent is playing against a human player, the exploration rate is reduced to a value of 0.001, since when playing against real players we save a pre-trained q-learning agent in order for the game to begin as soon as the player initialises it. The method, then randomly selects an action with probability equal to the value of epsilon or chooses the action with the highest Q-value given a state otherwise.

In the project’s requirements, we are asked to create opponents for our agents as part of the environment. Hence, we created a random agent, two threshold agents (one “loose” and one “tight” player) and a human agent class where all the needed actions are taken in terms of game and environment initialisations, prompts for the user and a small interface. The random agent class represents an agent that selects actions randomly. It returns a random integer in the range from 0 to 2 representing the action to be taken. The threshold agent classes have the additional methods set_hand, set_table and set_round to set the current hand,table and round respectively. In the send_action method, a dictionary that maps card ranks to actions for the first round is defined. If it’s the first round it returns the corresponding action for the agent’s hand rank. Otherwise, it checks if any card on the table has the same rank as the agent’s hand and returns the action accordingly. If no matching card is found, it returns 1, representing the default action to fold. The Threshold_Agent_A is a subclass of Threshold_Agent_D and overrides the send_action method. It uses a different policy in order to place a bet on the first round. Similar to the defensive threshold agent, it returns the corresponding action for the agent’s hand rank if the current round is 0. If a matching card is found on the table, it returns 2, representing the action to raise. Otherwise it returns 0, indicating the action to check. It is once again worth mentioning that we created a state transition matrix for each of the threshold agents. These new matrices contain 33 states since now we take into account the opponent’s last action too. They follow the same approximation logic that was used for the Policy Iteration matrix. This time, the matrix corresponding to the defensive agent (P_THRESHOLD_D) guides the agent towards performing safer moves that will not cost him a lot of chips in case of losing the hand. On the opposite side, the matrix corresponding to the aggressive agent plays in an unsafe way, betting with worse hands that the other agent.

Figure 2 : Transition matrix for Policy Iteration against the Threshold agents

In the utils.py file we also have some more methods to convert our state vectors into a numerical representation based on certain conditions and flags. More detailed explanation can be found in the comments sections close by each method.
Lastly, everything comes together in the main.py file. The play_a_game() method is where the gameplay logic is defined. It takes in the environment, an agent and an opponent agent as well as other optional parameters. The function plays multiple hands until a player bankrupts. It handles all game rounds, actions, rewards and updates the state. In more detail, inside the iterative part of the function (while loop) the agent chooses an action through the send_action method. This action is given as an argument to the env.step function. Then, env.step() calls the game.step(), who checks if the move is correct for this round of the game (e.g. we can't allow re-raise after a re-raise) and otherwise it transforms it into an acceptable move and then it is executed. The training_main() method, is the main entry point for training the agents. It takes parameters to configure and initialise the agents and the training process accordingly, utilising play_a_game. At the end of the training process, it saves the decisions made by the agent and plots the cumulative reward. At this point it is worth mentioning that this function was mainly implemented in order to save the final trained Q-learning agent playing against a threshold agent in order for it to be pre-trained and ready for when it's playing against a human agent. That is also when the testing_main function is used as it loads the pre-trained Q-values for the Q-learning agent from a CSV file. It initialises the agent and the human agent and then sets up the environment and plays a specified number of games using the play_a_game function.

Experiments & Results:
In this section, we will present the experiments conducted using the implemented code and discuss the results obtained. The experiments aimed to evaluate the performance of all the different agents in our poker-playing environment .

Hyperparameter fine-tuning:
In this small paragraph we aim to explain the fine-tuning of our algorithm's hyperparameters.
gamma : For gamma we decided that both Q-learning and Policy Iteration should have the same value, considering the fact that they are both implemented in the same environment. After conducting some experiments it was proved that gamma should have a value of 0.9 or 1. We begun with a value of 0.1 that yielded negative mean rewards for our agents, then 0.6 improved them but was still smaller than 0.9 and finally 0.9 and 1 yielded the same, very good mean reward results so we decided finally to tune it to 0.9.
epsilon : For the Policy Iteration agent we used a small positive float to make sure that consecutive V's are equal.

a : This parameter has a value of 0.4 for the first 20.000 episodes. After that, we reduced it slowly in order for our Q-learning agent to converge.

1. Demonstrate that Policy Iteration indeed solves our problem optimally

In the above graphs are depicted the actions that the policy iteration agent takes during a game of poker after training against the two threshold agents. In the first column are written the states as they were created for the game (see Figure 2) and in the second column of each graph are the actions taken by the agent at each state. By taking a close look at our list of transitions / transition matrices P_THRESHOLD_A and P_THRESHOLD_D, we can notice that our agent converges to the policy that it was taught to follow, since the agent always chooses the action that was assigned with the highest reward in the list of transitions. Therefore, we can use this information to state that the policy iteration algorithm converges to what we considered to be the optimal policy . After showing the convergence, we now focused on the effectiveness of our various agents in terms of victory rates and mean rewards.

2. Show the average reward achieved for different opponent types

As it was explained in the code implementation and environment setup parts of the report, our agents beginning 'bankroll' is a total of 4.5 small blinds, with a maximum profit -if the opponent bankrupts- of 4.5 small blinds as well. In our training_main function (in the main.py file) we measured the mean reward of our Policy Iteration and Q-Learning agents during a period of 1.000 episodes (full games till bankruptcy), when playing against both the threshold and the random agents. The results can be seen in the last line of every column in Figure 3 of the appendix section as well as in the table 1 attached below. It can be easily noticed that the Q-learning agent achieves the maximum reward (+4.5) playing against every opponent, whilst the policy iteration agent yields it's best results when playing against the defensive ('tight') threshold agent, then against the random opponent and it's worst results against the aggressive ('loose') threshold opponent. The above results seem to be reasonable since the Q-learning agent learns how to play autonomously, without being influenced by the environment's hand-written rules. On the other hand, the policy iteration agent follows the rules we dictated to him through our transition matrices. Considering the fact that there is a big level of abstraction in our state space and that we modelled the agent to follow our amateur-player rules, it seems reasonable that it has a smaller mean reward than Q-learning. Here, it is also worth mentioning some of the disadvantages of the Policy iteration algorithm that we notices while implementing it. First, due to the high level of abstraction that we used for the state space, the measuring of the transition probabilities from one state to another was very complicated, hence we decided to use some approximations there too. Secondly, after implementing and training the Q-learning agent we noticed that due to various parameters of the game environment (small amount of initial chips, no 'all-in' action, game.step() action "approximations") it's a weakly dominant strategy for our agents to play aggressively in order to force the hand of the opponent. Let's consider though that the Policy Iteration agent is still on the winning side in all cases with an excellent result when playing against the defensive threshold opponent.

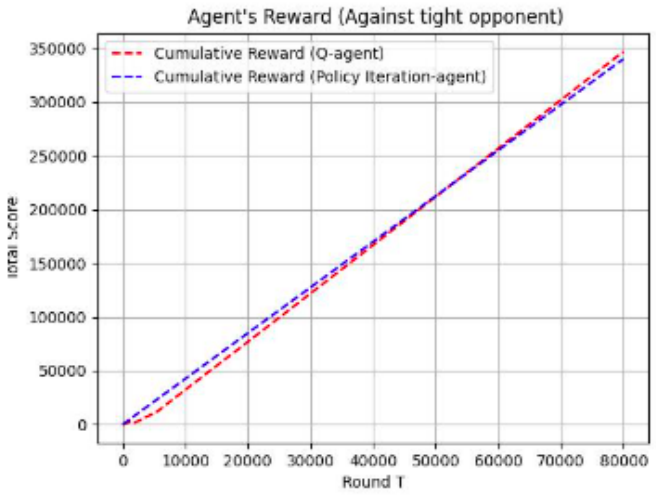
Opponent	*Policy Iteration (mean reward/profit)*	*Q-learning (mean reward/profit)*
Threshold_Defensive	+4.25	+4.5
Threshold_Aggressive	+2.5105	+4.5
Random	+3.7527	+4.5

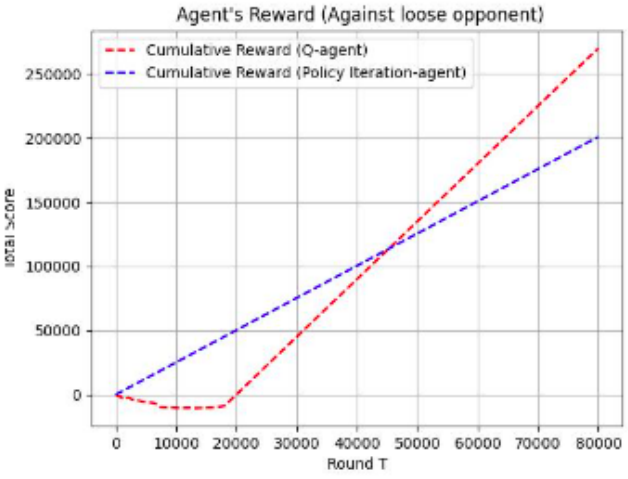
Table 1: Mean agent reward for all opponents

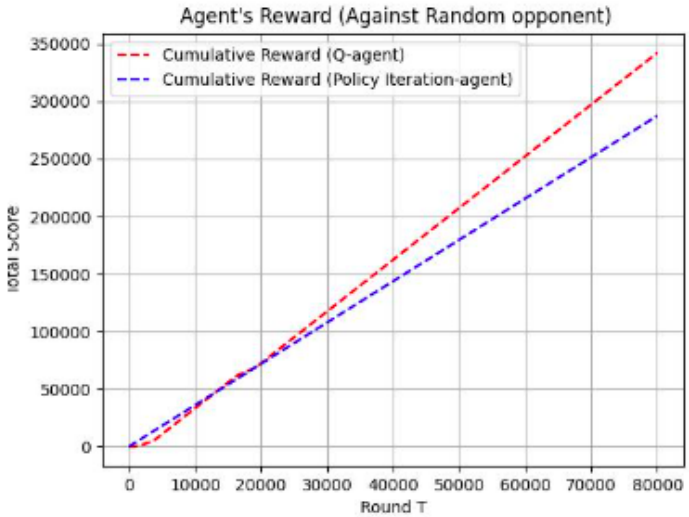
3. Prove that our Q-learning algorithm correctly learns the optimal policy as well. Demonstrate the convergence speed to the optimal.

First, as it was mentioned above, Q-learning yields the optimal mean reward. In figure 3 of the appendix, you may notice three different cell colours being used. Green symbolises that both the agents follow the same action at the same state/game situation. Orange cells contain cases where one agent chooses check(0) and the other raises (2).We consider this to be a small difference since, as it was mentioned, Q-learning learns to exploit the fact that being more aggressive is considered at last a better policy whereas policy iteration is a more conservative player. As a matter of fact, in most of the cases when P.I. checks, Q-learning

raises. The red cells are the ones where the two agents differ. Once again we notice that Q-learning is more aggressive. Since the final mean reward of Q-learning gets it's maximum value, it can be stated that Q-learning learns the optimal policy, which is a more aggressive one than the one we 'taught' P.I to follow.
The convergence of Q-learning is shown in the figures, as it's cumulative reward for a horizon of 80.000 episodes is depicted together with the one of the Policy Iteration agent. From the graphs it can be seen that Q-learning agent converge every time. An interesting fact that proves our aforementioned statements it that the Q-learning agent converges at a faster rate against the random and tight opponents whereas it has a greater difficulty learning against the loose opponent who, as we mentioned, is the most difficult one to win against. It is worth mentioning that the convergence time of Q-learning agent against the tight opponent is less than the time against the random. The previous, was caused by the fact that against the random agent, Q-learning has to learn against a noisy/stochastic opponent policy.







Figures 4,5,6 (from left to right and from top down) : Q-learning and Policy Iteration Agent's cumulative reward for 80.000 episodes against the defensive (tight) threshold opponent, the aggressive(loose) opponent and the random opponent.

Extra : 4. What would happen if there was no 'ante' :
After completing our experiments we moved on to the question above; What would happen if there was no 'ante' ? Based on the game's theory we would expect that our agents played in a more defensive way, deciding to fold when their hand card is considered to be weak. (e.g. 10 pre flop -> fold). In order to prove our theory, we conducted an experiment by deactivating the

ante and the results, that proved us to be correct, can be seen in figure 7 in the appendix. It is worth mentioning that the duration of this experiment was of about 6-8 hours, probably due to all the folding cases that slowed down our agents converging speed. More plots can be found in the folder 'images' of our project and in the appendix section below.

Conclusions :

In conclusion, after completing this project we observed that in our case it was easier for a Q-learning agent to learn/converge when facing a player with a logical strategy compared to a random player, as it can be seen in figures 4,5,6. The Q-learning algorithm demonstrates a faster learning speed in such cases. Furthermore, running the experiments with the maximum state size (2^{10}) doesn't seem to play a significant role, as even in the largest state spaces the convergence speed remains relatively similar (see figures 8,9,10 in the appendix), the cause behind this behaviour is that the really different states are about 80, considering the fact that we are not interested in the cards suits. Hence, the number of states that we used (20) doesn't differ much from 80.What truly matters is the number of genuinely different states that exist within the environment. It is worth noting that the main difference between running experiments with a different state size was the duration of every episode (almost x10 for the 2^{10} state size) . Lastly, creating an environment and understanding its specifications before and while developing it, proves once again to be a challenging task. Some possible improvements would be for our agent to store more information concerning the previous opponent's moves, total chips, total pot etc . All of the above will definitely have a great cost on the state space, hence they need to be addressed using a different approach (DRL).

Appendix :

Note: In order to achieve reproducible results, a sequence of pseudorandom integers was used as a seed in every episode.

Threshold opponent (Aggressive)	Policy Iteration	Q_learning
State 0	2	2
State 1	1	1
State 2	2	2
State 3	0	2
State 4	0	2
State 5	2	2
State 6	0	0
State 7	1	1
State 8	0	0
State 9	0	2
State 10	0	2
State 11	2	2
State 12	2	0
State 13	2	2
State 14	1	1
State 15	2	0
State 16	0	2
State 17	0	2
State 18	1	2
State 19	2	2
State 20	2	2
State 21	2	2
State 22	1	1
State 23	0	2
State 24	0	2
State 25	0	2
State 26	1	0
State 27	2	0
State 28	2	2
State 29	2	2
State 30	1	1
State 31	0	0
State 32	0	2
Mean reward	2.5105	4.5

Threshold opponent(Defensive)	Policy Iteration	Q_learning (80*10^3)
State 0	2	2
State 1	1	1
State 2	2	2
State 3	0	0
State 4	0	2
State 5	0	0
State 6	2	2
State 7	1	1
State 8	0	2
State 9	0	2
State 10	0	2
State 11	1	1
State 12	2	2
State 13	2	2
State 14	1	0
State 15	2	2
State 16	0	2
State 17	0	2
State 18	1	1
State 19	2	2
State 20	2	2
State 21	2	2
State 22	1	1
State 23	0	2
State 24	0	2
State 25	0	2
State 26	1	1
State 27	2	2
State 28	0	0
State 29	2	0
State 30	1	2
State 31	0	2
State 32	0	2
Mean reward	4.25	4.5

Random opponent	Policy Iteration	Q_learning (80*10^3)
State 0	0	2
State 1	2	2
State 2	2	2
State 3	0	0
State 4	0	2
State 5	2	2
State 6	2	2
State 7	0	2
State 8	0	2
State 9	2	2
State 10	2	2
State 11	0	2
State 12	0	2
State 13	2	2
State 14	2	2
State 15	1	2
State 16	0	2
State 17	0	2
State 18	2	2
State 19	1	2
Mean reward	3.7527	4.5

Figure 3: Transition matrix results after training with different opponents for both the Policy Iteration and the Q-learning agent, mean rewards

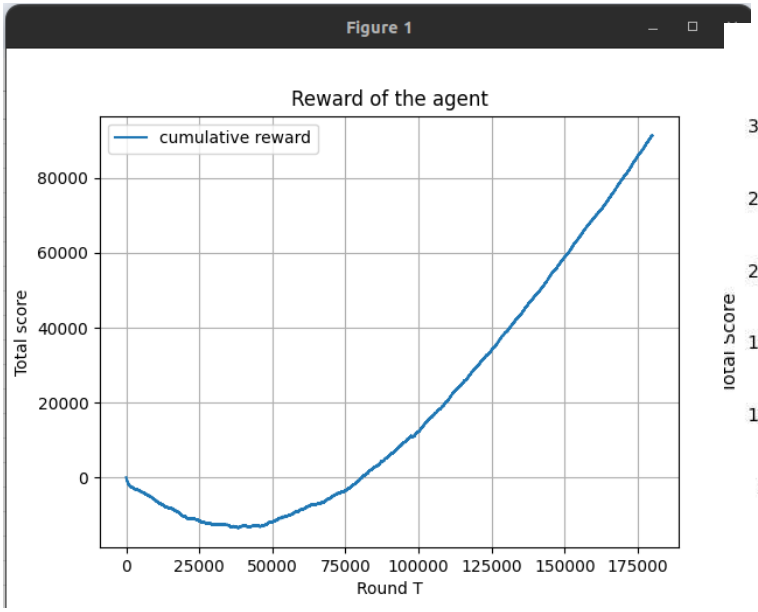


Figure 7: Q-learning no-ante

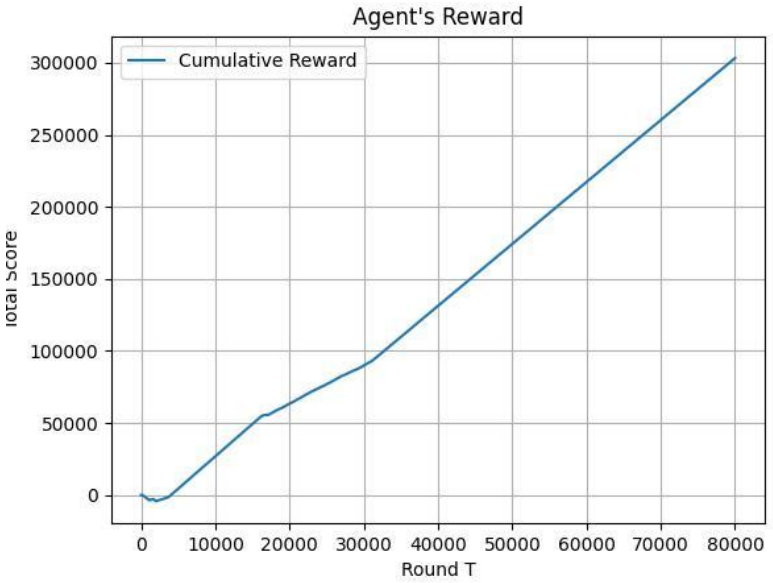


Figure 8 : Q-learning against the random agent , state space= 2^{10}

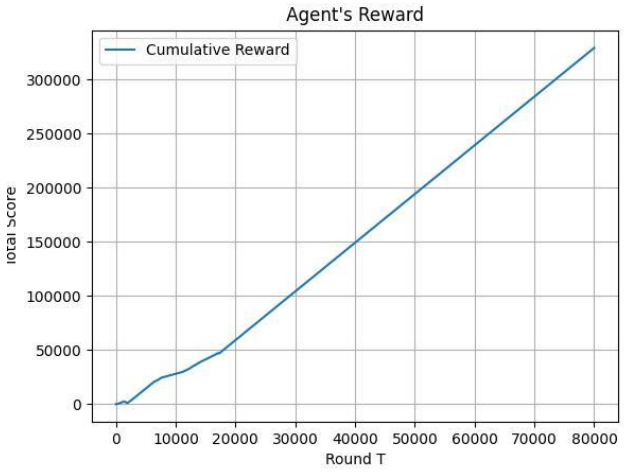


Figure 9 : Q-learning against the defensive threshold agent, state space= 2^{10}

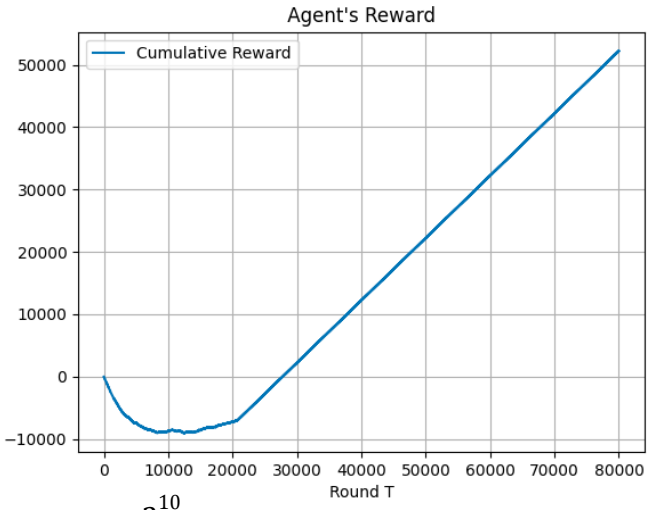


Figure 10 : Q-learning against the aggressive threshold agent, state space= 2^{10}