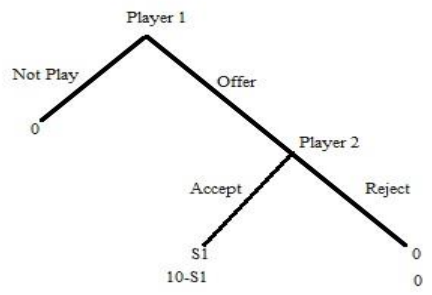
Faculté
des **sciences économiques** et de **gestion**
Université de Strasbourg

The ultimatum game & Reinforcement Learning.



Ortega López Esteban

Santos Moreira Sarah

Master 2 DS2E

Enseignante : Bertrand Koebel
Université de Strasbourg
Année Universitaire 2022-2023

Index

1. Introduction

The ultimatum game is an exercise that has been widely used in game theory and experimental economics (Thaler, 1988). The setting is simple, a situation involving two agents: A and B, must decide how to split an endowment. Agent A is given a certain amount of money and must decide which part to keep for himself and which part offer to Agent B. Agent B can reject the offer and both agents get nothing, or agent B can accept the offer, in these case agent B gets the amount offered by agent A and agent A gets the endowment minus the amount offered. In the theoretical result of the game, agent A will offer the minimum amount to agent B and agent B will accept such offer, indeed, for agent B the dominant strategy is to accept no matter the amount of the offer, since the alternative reward is 0.

For this work we apply Reinforcement Learning (RL) to simulate the ultimatum game with artificial agents. Reinforcement learning is a machine learning field that enables an agent to learn to interact with the environment and take actions to maximize rewards in a specific situation (Sutton & Barto, 2018). Unlike supervised or unsupervised learning, reinforcement learning does not provide training data to the algorithm. The agent learns by being directly in contact with its environment.

The ultimatum game can be seen as a Markov Decision Process (MDPs), where an agent must take sequential actions, but such actions influence rewards, future states of the environment and for that, future rewards (Sutton & Barto, 2018). An option to solve this kind of problems is the Q-learning algorithm. In this algorithm, by a combination of exploration and exploitation actions, an agent is capable to find the value or reward related to a state-action pair, and like this, given enough time to learn, the agent learns the best possible action for a given state. Q-learning is convenient when the exact probabilities, rewards and penalties are not known.

On this work we simulate the ultimatum game with artificial agents created on the python programing language. The agents will learn to play the game by using reinforcement learning, several simulations will be run to search for the optimal values of some of the parameters set in the definition of the game. On these simulations, first the two agents of the game will learn to play the basic design of the game. But also, another type of simulations is proposed where the agent A learns to play against a policy of decisions for agent B. Indeed, the agent A is going to play against fixed rules: Agent B will only accept the offer if the proportion of the current offer from agent A to the endowment is greater than a certain proportion of the endowment, otherwise he rejects it. Agent A will also play against dynamic rules. For the dynamic part, it is the same game as playing with fixed rules but the value of p changes for some episodes. On the second part of this work a short literature revue will be found, then the agents and the game definition will be shown for the simulations proposed, the simulations and results are explained in a fourth part of the work, and finally some closing remarks.

2. Literature on Reinforcement learning and game theory

Reinforcement learning can and has been implemented in game theory but not without limitations. Nowé, Vrancx, & De Hauwere (2012) explain how reinforcement learning can be difficult to use in multiagent games given that the assumptions made for an agent to find an optimal solution are often violated. Nonetheless the authors explain a way to implement reinforcement learning in Repeated games (Q-Learning) and Sequential Games (value iteration and policy iteration). In additions, the authors focus on the coordination issue in multiagent games to achieve a global optimum.

On the same note, Bowling & Veloso (2000), explain more in detail the set of possible solving algorithms for multiagent games in stocasstich games. The authors focus on a presentation of the algorithms used in game theory, such as the Shapley and Pollatschek & Avi-Itzhak approach, and in reinforcement learning, like minmax-Q or nash-Q. With this in mind, the authors propose a matrix with the combinations of algorithms available to solver the multiagent games using reinforcement learning.

Zhong, O. Kimbrough, & Wu (2002) had also used a design to use reinforcement learning in the ultimatum game. Indeed, they use the java programing language to define properties of the game and run several simulations. Among the exercises they propose, cooperation is achieved when both agents keep the memory of the action of their counterpart in each episode. In their work it is also included a multiagent simulation in which the value of intelligence is the main point of focus. For this project inspiration is set on the first simple designs of Zhong, O. Kimbrough, & Wu, however, the focus for this project is on the optimal values on the paramters of the defined agents.

Le Gléau et al. (2020), propose a model for the multiagent version of the ultimatum game as well. This version is called the pirate game, in which one agent must offer an amount of coins to a group of pirates and they must vote to accept or not the offer made. The authors use an Artifitial Neural Network for the model, which is a more advanced approach than the ones mentionned before. Simulations are run for different designs of the game in which the pirates can be selfish or when they can have a prosocial vote, that is, each pirate considers the social benefit to make a decision. The authors find that often the model proposed leads to the theoretical optimum.

3. Definition of agents (players) and the Ultimatum game.

Players

Object-Oriented Programming (OOP) is based on the concept of objects which are each defined with their own attributes, this part of the work focuses on explaining the design of the elements on the game using OOP in the python programming language, the code will be attached as annex to this work so that the reader can follow along. Every object also contains its own methods. The first method *__init__()* is called class constructor. Class methods are like normal functions with the exception that the first argument of each method is *self*. Self.epsilon, self.alpha and self.actions are attributes attached to the class object and they are placeholders. Furthermore, the code contains different methods such as the methods move and update_qtable.

In the ultimatum game there are two players, so two different classes are used, which will be named player 1 and player 2. Player 1 is the one obtaining the endowment and making an offer to player 2. Player 2 will have the option to accept or reject the offer made by player 1.

The class Player 1 takes a constructor that has self, epsilon, alpha and action as attributes. Self refers to the Object itself. Epsilon is the probability of doing a random action between 0 and 1 for each action taken by player 1, and alpha is the learning rate. Action refers to the number of actions that player 1 has, which depends on the endowment. If the endowment is 50 units, player 1 will have 51 possibilities of action since the agent can offer to the counterpart from 0 to 50 units. Given the structure of the game, player 1 only has one state since he always does the first move, so it doesn´t depend on player 2's actions. Therefore, by default state is 0.

Method *move* defines the decision process done by player 1 following an epsilon-greedy strategy and takes the state in which the player is as the argument. In this case the state is always 0 and method *move* returns an integer, representing the offer to be made to player 2.

To store the Q-values for each state action pair we create a Q-table. There are n columns, where n is the number of actions. There is only 1 row, so we have only one state. Since the player 1 knows nothing about the environment or the expected rewards for any state action pair, all the q-values in the table are first initialized to zero. This is the Q-table for player 1:

	Action				
	0	1	2	---	n
State (0)	0	0	0	0	0

To determine if the player 1 will choose *exploration* or *exploitation* at each episode, we use a random number between 0 and 1 (with a uniform distribution). If this number is greater than epsilon, then the player 1 will choose *exploitation* as his next action. This means that he would choose the action with the highest Q value for his current state from the Q table. Otherwise, if the random number is smaller than epsilon then his next action will be *exploration*. This means that he would choose a random number between 0 and the endowment, which will be the offer.

We need to update the function Q(s,a), meaning the q-table for the player 1 needs to be updated according to the action taken. To do so, we use the method “update_qtable”. The formula for calculating the new Q-value for state action pair is as follows: The current value multiplied by one minus alpha, added to the reward value multiplied by alpha.

$$Q(s,a) = current\_value * (1 - alpha) + reward * alpha$$

The second class represents the second player: Player 2. Once again, we have a constructor for the player 2 class with different parameters. Epsilon represents the parameter for the epsilon greedy strategy. This means that the agent will take a random action with the probability of epsilon. alpha_learn_rate represents the learning rate used to update the Q-table. The states of the game for Player 2 depend on the offer Player 1 makes considering his endowment. There are two possible actions for Player 2 , which are either to accept (1) or reject (0) Player 1’s offer.

Q-table for player 2:

Action

State

	0 (reject)	1 (accept)
0	0	0
1	0	0
2	0	0
3	0	0
…	0	0
m	0	0

For the player 2, we also used the method move. This method defines the decision process done by player 2 following an epsilon-greedy strategy. It takes as argument the state in which the player 2 is (the offer made by player 1) and returns an integer, 0 if he rejects, 1 if he accepts. **G**iven the state, which in this case will be the offer made by the player 1, the player 2 will decide according to an epsilon - greedy strategy. If the random number strategy is greater than epsilon then the greedy action is to be taken else if the random number strategy is not greater than epsilon, the exploring action is to be made. It's the same strategy as the player 1. Then, the Q-table for the player 2 needs to be updated according to the action he has taken. the new value in the table is the alpha weighted reward.

Q(s,a) = current_value * (1 - alpha) + reward * alpha

Game

Finally, we define the class for the ultimatum game, called ultimatum. The first method is the class constructor. The following attributes are defined: episodes, player_1, player_2, endowment, and policy. Episodes is the number of episodes to be played in each simulation. Player_1 and player_2 are two classes defined in the player module. The endowment is the amount of money that will be divided in the game. The policy argument defines the proportion of the endowment that player 2 will ask as a minimum to accept the offer in case the game is set to have fixed or dynamic rules. Empty lists are created for results, rewards, and payoffs to save the results for every episode and to access this information later.

The method *episode* defines the steps taken in each episode of the game. First, player 1 makes a move, meaning that he makes an offer to player 2. Player 2 then accepts the offer or rejects it. Depending on the action taken by player 2, the rewards are set. The q values of the q tables are updated for both players according to the actions taken. Actions, rewards, and payoffs are stored in the lists previously initialized to keep track of the game.

Method *play* defines the behavior of the game. The method takes as an argument if the game will follow a policy for agent B or not, and if so if the policy is dynamic. This method calls the method episode for the number of times that the attribute episodes is set to be. The results for each episode are saved into a table that will be used to analyze the outcome of the whole simulation.

4. Ultimatum game simulations

Different versions of the ultimatum game are performed using the agents and the previously defined game rules. Initially, a simulation of 10,000 episodes is performed to demonstrate the behavior of the game, then an exploration exercise is performed with the epsilon and alpha parameters for each of the agents. When the optimal parameters are found, a new set of simulations is performed, and the results are compared. In addition, the game experiment is performed using fixed and dynamic rules in the decisions of agent B to verify if agent A is able to identify the rules and internalize them in his strategy. In all games, an initial endowment of 100 units is proposed.

Repeated one shot game.

To begin, a version of the game is made with preset parameters to verify the normal performance according to the theory. The epsilon and alpha parameters are initialized at 0.1 and 0.2 respectively for the two agents. An instance of the game with 10,000 episodes is created. Partial results are shown below:

1. Table. Partial Results for ultimatum repeated game

	A's offer	B's action	B's average actions	A's reward	B's reward	A's payoffs	B's payoffs
0	0	0	0.000000	0	0	0	0
1	0	0	0.000000	0	0	0	0
2	0	0	0.000000	0	0	0	0
1000	26	1	0.774226	74	26	53744	23756
1001	26	1	0.774451	74	26	53818	23782
1002	26	1	0.774676	74	26	53892	23808
9997	4	0	0.872274	0	0	762124	109976
9998	4	1	0.872287	96	4	762220	109980
9999	4	1	0.872300	96	4	762316	109984

The table shows that agent A's offers start at 0, since it is the highest value historically obtained, while agent B maximizes his utility, which started at 0, by rejecting all bids. As the episodes elapse, a growth of agent A's offers is observed in the first episodes, however the level of the offer decreases as more episodes are played, at the same time the average action of agent B increases steadily approaching one, reflecting the theoretical result that agent B will always accept any offer from agent A no matter how minimal it may be. Thus, agent A's gains for each episode increase over time, while agent B's gains, while not zero, decrease over time. A more in-depth analysis of this initial game can be found in the appendices. The results of the search for optimal learning parameters for each agent are shown below

Exploring the parameters of the game

For the exploring of the different parameters several simulations are proposed with variation in one parameter and keeping the rest constant, an analysis on the average of results across the simulations is made to find the optimal value for the different agents. First, different values for epsilon are proposed.

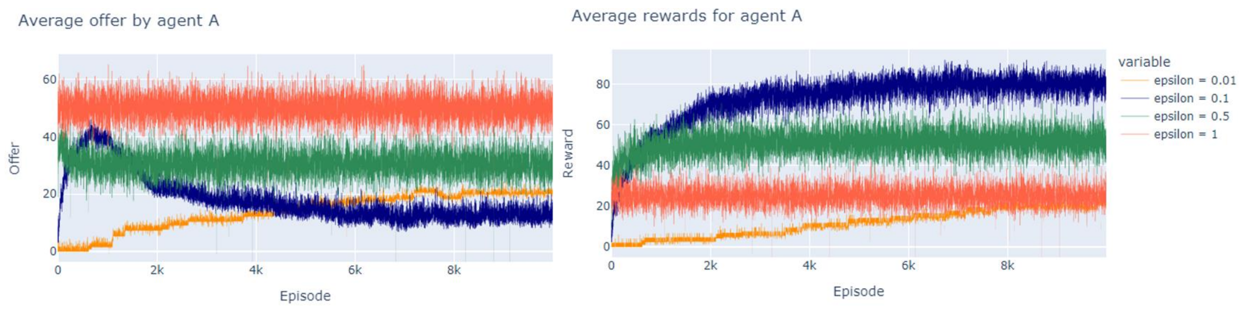
Epsilon

In this exercise epsilon will take different representative values: 0.01, 0.1, 0.5 and 1. A total of 50 simulations (each one with 10.000 episodes) for each value of epsilon. As for alpha, the parameter is set to 0.2 for all simulations, the endowment is constant at 100 units.

As expected, the average offer of agent A in every episode is 50 when epsilon is set to 1, indeed, in that case the agent takes a random action in every episode, given that all actions have the same probability, the expected value is the mean of all possible actions. The path of offers for an epsilon equal to 0.1 is the one with the fastest decrease, which means that when the agent takes a random action only on 10% of the episodes, learns faster to minimize the offer. When epsilon is too small, the path of the offers grows in the timelapse considered, most likely to decrease after much longer; and as for an epsilon of 0.5 it grows quite fast in the first hundreds of episodes, rapidly decreases but stays rather constant around 30 units for the rest of the simulation.

In consequence, it is observed that when epsilon is set to 0.1 the rewards grow faster than the other values, with an epsilon too small the rewards grow at a rather slow pace, while with an epsilon too large, the growth stops after the first hundreds of episodes. For agent A the optimal value of epsilon is 0.1 according to these results.

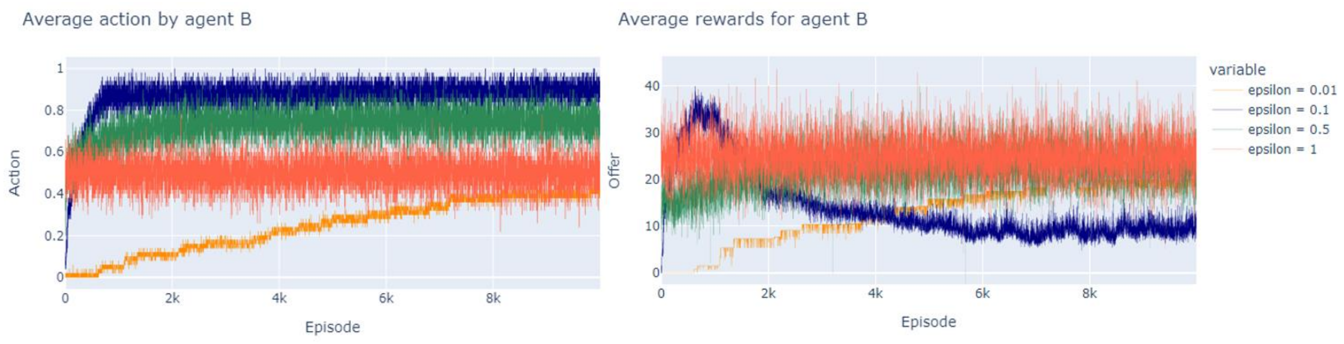
2. Offers and rewards for agent A under different epsilon-greedy strategies



As for the agent B, it can be showed that with an epsilon of value 0.1 the learning towards the theoretical result of the game is faster, that is, the agent learns faster to accept all the offers made by the agent A. The agent also learns to accept the offers with an epsilon as small as 0.01 but at a much slower pace. With an epsilon of 0.5 the agent also learns to accept the offers, and will accept in most cases, while that with an epsilon of 1, as expected, the decisions are random for which the agent will accept the offer on 50% of episodes on average. Interestingly though, the rewards for agent B grow fast in the first thousand episodes but the constantly decrease when epsilon takes a value of 0.1, this is because the learning process of agent B causes for the agent to accept every offer, agent A learns this behavior and continuously decreases the offer that agent B accepts. Rewards in every episode grow slowly for agent B with an epsilon of 0.01, getting to be greater per episode than with an epsilon of 0.1.

With epsilon taking values of 0.5 and 1 the rewards are rather constant around the level of 30 units, which could lead to think that given that agent B randomly accepts the offers from agent A, this last one does not learn to minimize the offers as fast, for that, the split of the endowment is more equative for the agents. Analyzing the total payoff of the average rewards per episode it can be concluded that agent B maximizes the payoff by acting randomly in every episode, that is, an epsilon of value 1.

3. Actions and rewards for agent B under different epsilon-greedy strategies



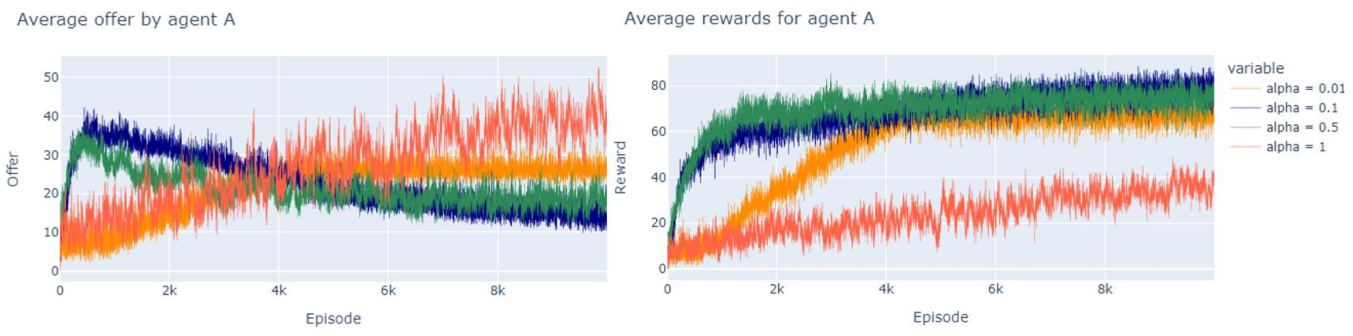
Alpha

For this exercise, the parameter alpha will take different representative values: 0.01, 0.1, 0.5, 1 while epsilon will stay constant at a value of 0.1, the value that showed the fastest learning path for both agents. Then again 50 simulations, each one of 10.000 episodes, will be run for each value of alpha for both agents.

The results show that on average, the offers made by the agent A decrease faster with an alpha learning rate of 0.5 than with an alpha of 0.1, and by the end of the 10.000 episodes the offers are very similar with both parameters. On the other hand, when the learning rate is either too high or too low the offers for agent B increase over time on average, indeed, with an alpha of 0.5 the offers grow up until around 30 units and stay roughly constant, while as for an alpha of 1, the offers rise until around 40 units and show a less steady behavior over the timelapse.

Following these results, it ends up being the value of alpha equal to 0.5 the one that makes the average reward per episode grow the faster, although by the end of the game, the maximum average reward is obtained with an alpha value of 0.1. The total payoff for agent A is then maximized with an alpha of value 0,5.

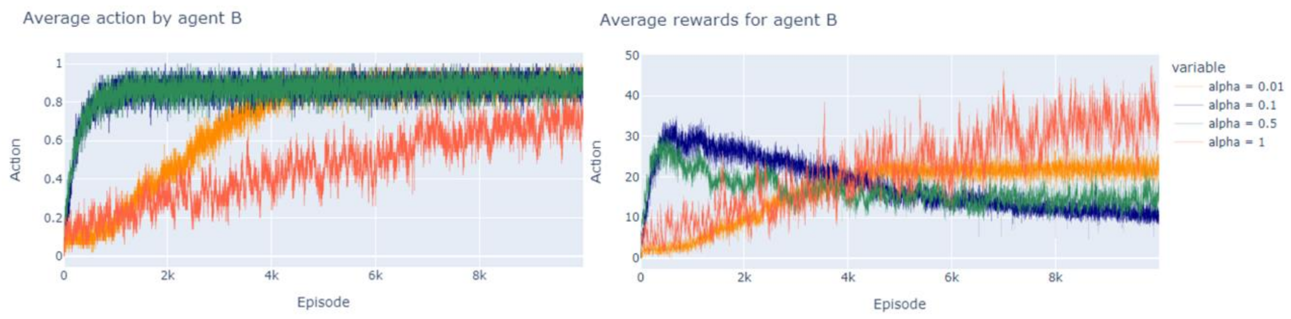
4. Offers and rewards for agent B with different alpha learning rates



In a similar way, when alpha takes the values of 0.1 and 0.5 the learning process for the agent B is faster, that is, the agent learns to accept every offer made by agent A earlier in the game. The path of the actions made by agent B with these two values of alpha is roughly the same. The learning process when alpha is equal to 0.01 also converges to the result where agent B accepts every offer made by agent A, however it does in a slower way, as for an alpha with value of 1, the average action by agent B does increase over time, meaning also learns to accept more offers, but it does it in an even slower way and also does not quite converge to accept every offer.

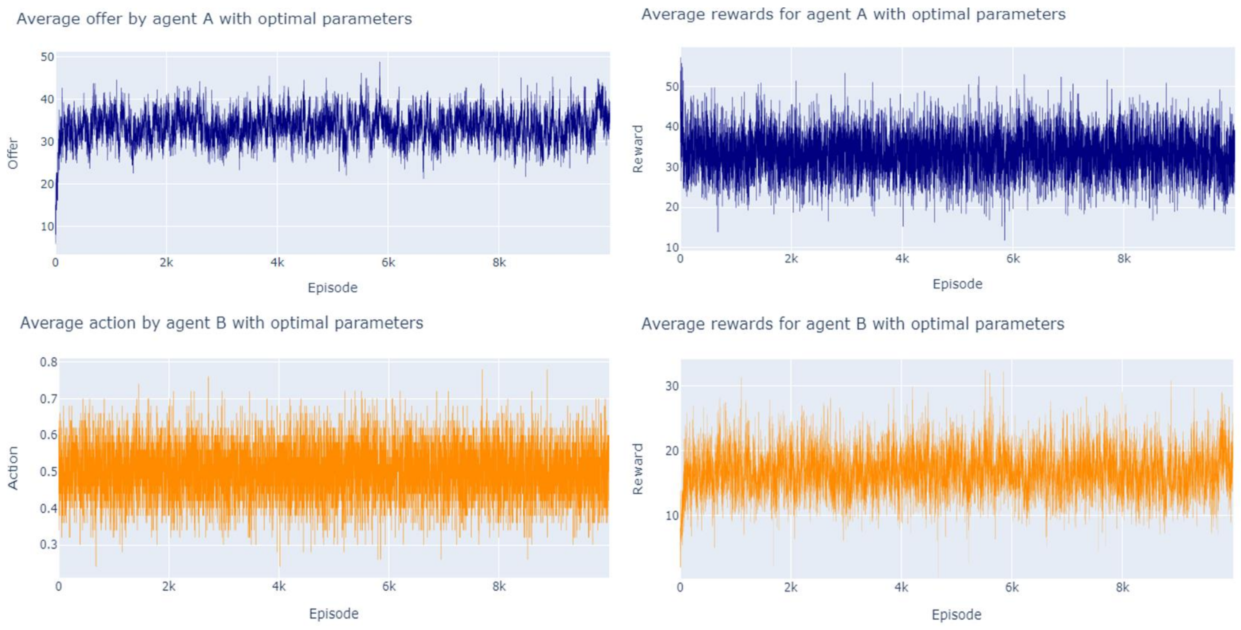
Because of the learning path for agent B, it can be noted than the average reward per episode decreases for the values of alpha 0.1 and 0.5. Certainly, the average reward in these cases increases very fast in the first episodes but then decreases constantly over time, because the agent learns fast to accept every offer made by agent A, agent A learns this behavior and every time makes a lower offer, one that agent B accepts and because of this gets a lower reward. On the other hand, when alpha is rather low, the rewards for agent B increase during the first thousands of episodes ant then stay somewhat constant for the rest of the game, generating a higher payoff in the end. Finally, with an alpha value of 1, since the agent does not learn as fast to accept every offer, the agent A can no longer learn to lower the offer, for this reason agent A makes offers than imply a higher reward for agent B. Like this, agent B maximizes total payoff by setting the alpha learning rate to 1, meaning that the q value for every state-action pair is updated by taking the full value of the las reward in that pair state-action.

5. Actions and rewards for agent B with different alpha learning rates



With the results found for the optimal values of epsilon and alpha for both agents in the game, a new one-shot game set of simulations is proposed. In this case, however, both agents will set the parameters to their optimal value. It is found that in this scenario a sort of cooperation is achieved. Indeed, the offers made by agent A set around 30 to 40 units, far from the theoretical result of the game, like this, the rewards are also bounded. The decision process made by agent B, with an epsilon and alpha value of 1, is rather random, which makes difficult the learning process for agent A, therefore causing the stagnation of the offer level and generating a higher payoff for agent B. A deeper discussion about the optimality of the parameters can be achieved by testing different combinations of them, given that the interaction might cause different results. This question is opened for future research.

6. Actions and rewards for agents A and B under optimal parameters



Learning against rules

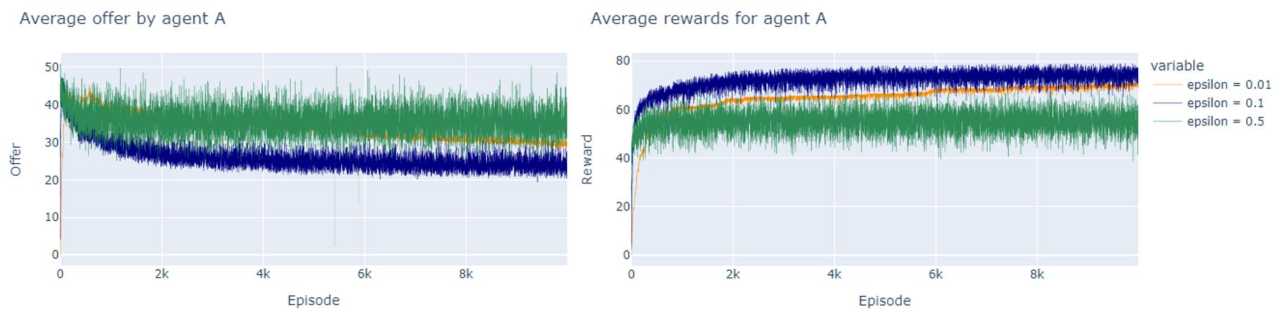
Now, another version of the game is proposed in which the decision process for the agent B is limited to a rule or policy. With this design it is desired to know if the agent A can distinguish and follow the policy for the agent B so that the offers are accepted. In this case only the learning of the agent A is of interest, agent B becomes part of the environment for agent A.

Fixed Rules

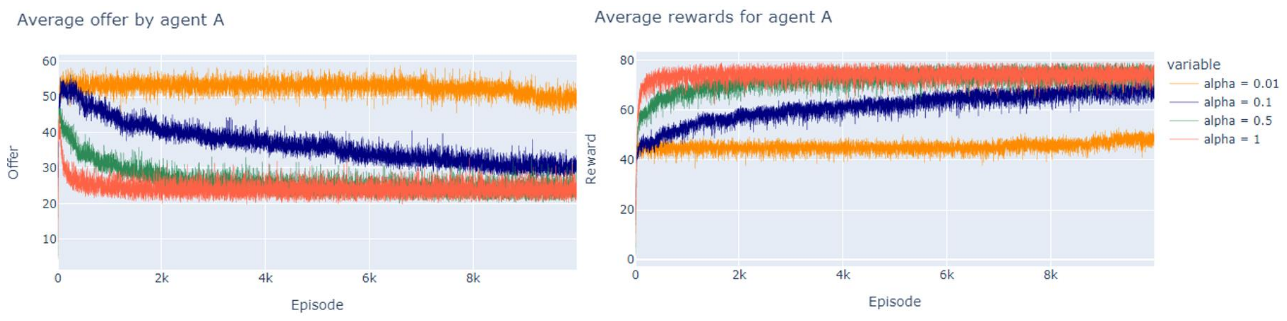
For this first design, agent B follows the rule in which only offers higher than a certain proportion of the endowment are accepted. Thus, the parameter policy = p ϵ [0,1] is introduced, and agent B would accept the offer if the offer made is greater than p*(endowment). A total of 50 simulations is run to see the average behavior of the agent A. The policy parameter is set to 0.2.

The results show that for different values of epsilon and alpha, agent A learns to lower the offers to the limit set by agent B. Meaning that agent A offers a minimum of around 20 units so that the agent B would accept the split of the endowment. Agent A continues this behavior until the end of the game on average in the simulations. The learning process again is faster when epsilon takes the value of 0.1, like this, the offers drop faster, and the rewards grow faster. On the other hand, when introducing the rule, the parameter alpha finds its optimal value at 1, making the process of learning completely relying on the last value for each state-action pair.

7. Average offers and rewards of agent A for different values of epsilon against fixed policy



8. Average offers and rewards of agent A for different values of alpha against fixed policy



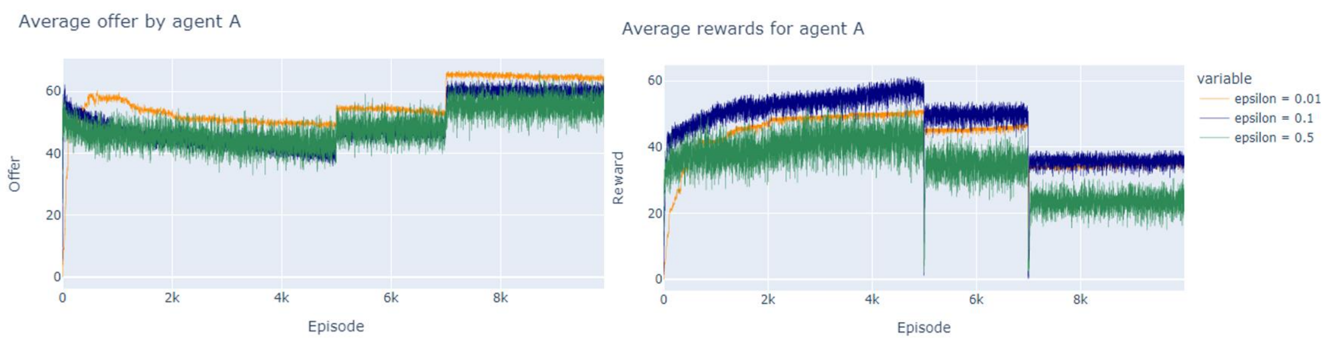
Dynamic rules.

In this version of the game, agent B still follows a rule in which only offers higher than a certain proportion of the endowment will be accepted. However, the policy parameter will change in time, the policy is dynamic so to study if the agent A is capable of recognize the change in the limit, learn it and incorporate it in the strategy. Again, 50 simulations are run, in each, the game consists of 10.000 episodes where the p parameter changes like this:

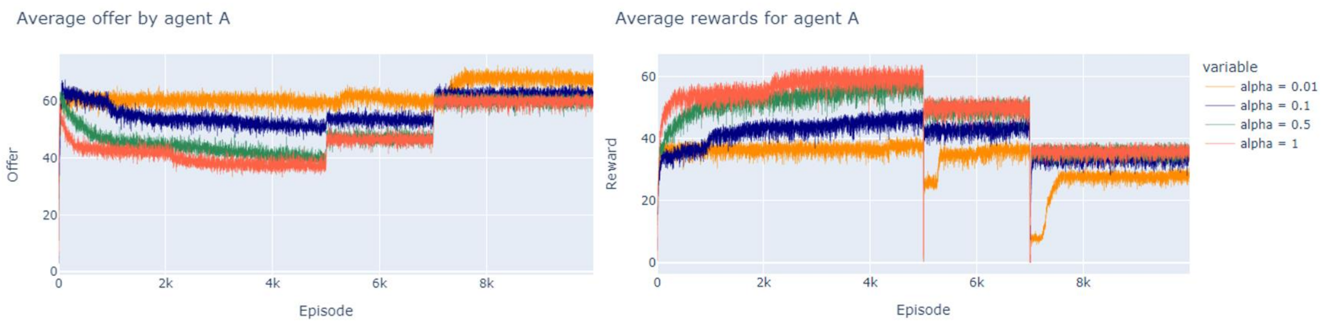
Episodes					
	1	2000	5000	7000	1000
p	0.4	0.35	0.45	0.60	0.4

The results show that with the reinforcement learning algorithm proposed agent A is successful at following the change in the policy set by agent B in different phases of the game. Indeed, the average offer made by agent A decreases until the limit set by agent B until a new limit is set, then the learner agent will note that offers are no longer accepted, for which the offers will shift towards a higher value. The plot for the average rewards per episode for agent B shows the sharp decreases when the policy is changed, in these episodes the agent A continues to make an offer too low for B to accept, after some episodes the epsilon-greedy strategy makes agent A change the offer for a higher one, there the rewards are again positive and agent A continues within the range of accepted offers. In this case again it is noted that the learning process and the maximization of the payoff for agent B is when the epsilon parameter is set to 0.1 and alpha takes a value of 1.

9. Offers and rewards of agent A under different values for epsilon



10. Average offers and rewards of agent A under different values for alpha



5. Closing remarks

The ultimatum game is a dynamic exercise in which an agent (A) is given a sum of money and must decide how to split it with another agent (B). When agent A makes an offer, agent B must decide whether to accept it or not. According to the theoretical result, the agent A would offer the minimum amount possible to agent B and this last one would always accept it, like this both agents maximize their utility. However, experiments under this framework have shown outcomes different from those in the theory.

In this work we design a simple version of the ultimatum game and its agents and run several simulations using the python programming language. The agents are capable of learning from the game by using reinforcement learning, more precisely, the q-learning algorithm. By doing so, the optimal values for the agents' strategies are found, and the results are close to those offered by game theory. In this case, the optimal values for the epsilon and alpha parameters are different for both agents and a sort of cooperation is obtained out of randomness in the decisions of agent B. The learning is put to the test by facing the agent learning against fixed and dynamic rules of behavior of the counterpart, in both cases the agent was successful in identifying the rules and changing the strategy, however in this case the optimal value for alpha is 1. It can be concluded that the q-learning can perform well in such a simple design. However, further research can be done on the effect of interaction of the different values of the parameters on the definition of the agents.

6. References

Bowling, M., & Veloso, M. (2000). An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning. *School of Computer Science - Carnegie Mellon University*.

Le Gléau, T., Marjou, X., Lemlouma, T., & Radier, B. (2020). Multi-Agents Ultimatum Game with Reinforcement Learning. *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Trust-worthiness* (pp. 267-278). The PAAMS Collection.

Nowé, A., Vrancx, P., & De Hauwere, Y.-M. (2012). Game Theory and Multi-agent Reinforcement Learning. In M. Wiering, & M. van Otterlo, *Reinforcement Learning: State of the art* (pp. 441-470).

Sutton, R., & Barto, A. (2018). *Reinforcement Learning: An introduction.* London, England: The MIT Press.

Thaler, R. (1988). Anomalies: The Ultimatum Game. *The Journal of Economic Perspectives, 2*(4), 195 - 206.

Zhong, F., O. Kimbrough, S., & Wu, D. (2002). Cooperative Agent Systems: Artificial Agents Play the Ultimatum Game. *Group Decision and Negotiation*, 433-447.