Università
Ca'Foscari
Venezia

Bachelor's Degree

in Economics
and Management

Final Thesis

Reinforcement Learning in Normal Form Games

Supervisor
Ch. Prof. Massimo Warglien

Graduand
Claudio
Casellato
861553

Academic Year
2019 / 2020

Abstract

Reinforcement Learning is a framework into which many real-world problems can be casted. It is a learning paradigm which is based on iterative learning. It has shown great promise in the single agent setting, with applications to many fields and much of the research is focused on this setting (Kapoor, 2018). In this thesis we will present the multi-agent case, within which two agents play against each other and try to maximize their own reward. This is more challenging than the single agent setting because agents play against a non-stationary opponent. We apply this framework to the class of games called repeated normal form games. We first describe the general game setting and the requirements of the algorithm. We then introduce the theory behind the algorithm and its derivation. We then investigate how well this algorithm can approximate human learning and the resulting equilibrium that arises from those human interactions by comparing the simulation results to gameplays played by human players.

0 TABLE OF CONTENTS

1 INTRODUCTION

The aim of this thesis is to find an algorithm that approximates how humans learn by playing against an adversary. The aim is of descriptive nature (Fudenberg & Levine, 2007), defined as "how natural agents learn in the context of other learners". We evaluate the predictive power of the developed algorithm by statistically comparing equilibrium results and strategies trajectories to actual human game plays. Many methods have been developed for the multi-agent setting, a review of the main ones can be found in the survey by (Hernandez-Leal, Kaisers, & Baarslag, 2019). From the survey one can immediately see that the multi-agent scenario is highly heterogeneous and dependent on the specific setting, assumptions and goals of experiment. The methods in the survey are, however, of mainly computational and prescriptive nature and many of the assumptions of those algorithms are violated for the application to our setting. In this thesis we analyze one specific setting, which is learning in iterated normal form games, where two players play against each other by selecting simultaneously one of two available discrete actions at each time step. The agents, which are the individuals that play in the role of either player1 or player2, do not have access to either the actions played by the opponent or the strategies of the opponent, however the full action history is revealed to the agents at the end of each playing period, therefore not at every game iteration. Agents have access to just their own rewards at each time step and therefore not the rewards of the opponent, however they do have access to the full reward structure of the game. This is the experimental setting employed by (Selten & Chmura, 2008), from which we take the games bimatrices and the experimental human strategies upon which we evaluate the algorithm. The setting belongs to the more general concept of sequential decision making. Two main lines of research have been developed to study this setting. The first one is game theory and the second one is reinforcement learning. Each have long histories but have usually different notations and different assumptions. A subset of game theory that focus on how *humans* play games is Behavioral Game Theory. The setting in question mainly belongs to the behavioral game theoretic line of research however we utilize concepts from the reinforcement learning literature and adapt it to this setting. The proposed setting is challenging since agents cannot make an explicit model of the opponent, like the belief based framework of fictitious play (Brown, 1951) which requires opponent's actions and rewards to be known at every time step, and it restricts the class of algorithms

that can be applied to this problem. In the game theory literature this class of games are called imperfect information games since the agents do not have access to the full state of the game. In the reinforcement learning literature this setting can be considered as a stateless Markov Game. In the subsequent chapters we will state a formal definition of this setting. The last requirement of the algorithm is that agents must learn in an online fashion, which means that the agent modifies his strategy at every time step or after a small number of time steps. The outline of this thesis is as follow. Chapter 3 provides background information of game theoretical concepts and definitions, namely the games structure, the concept of Nash Equilibrium and the Quantal Response Equilibrium. Chapter 4 introduces the method of Reinforcement Learning by providing a general introduction to the topic, showing the relation between multi-armed bandits, Markov Decision Processes, matrix games and Markov Games, introducing the concept of Generalized Policy Iteration, the derivation of the algorithm and the comparison between the standard model of reinforcement by (Erev & Roth, 1998). Chapter 5 evaluates the results of the simulations against experimental data from human play and make final conclusions.

2 GAME THEORY

Game theory is the discipline studying the interactions among two or more rational and intelligent agents whose objective is to maximize their respective reward. One can analyze many complex scenarios that arise from the real world and can frame the problem into this framework. Game theory is often used to analyze the competition between economic agents (Neumann & Morgenstern, 1944). In this thesis we study games with just two discrete actions that iterate throughout a playing period of 200 iterations. Game theory usually focuses on the equilibrium of the game and not the dynamics to reach such equilibrium.

2.1 REPEATED NORMAL FORM MATRIX GAMES

Finite Horizon Repeated Normal Form Matrix Games are a type of games where players take actions simultaneously, for a finite number of game iterations. Players do not see the actions of the opponent after having taken the action and therefore can also not estimate the reward of the opponent for the joint actions but only have their own reward feedback to base their subsequent decisions. This setting is referred in the literature as imperfect information games. They can be represented by a reward matrix for each player, the cross product of those matrices is called a bimatrix.

Formally, a game G in strategic form is described by the tuple G = (N, S, U) where N is the number of players, in our case N=2. S is the space of pure strategies profile of the game: $S = S_i \times S_{-i}$, where S_i is the space of pure strategies of player1 indexed by i, S_{-i} is the space of pure strategies of the opponent indexed by -i, which refers to all the players except i. An element of the strategy profile $s \in S$ (also called the joint action set (Neto, 2005)) corresponds to the tuple $s = (s_i, s_{-i})$. In our case $s_i \in \{0,1\}$, and corresponds to the two actions available to the player i. U = $u_i \times u_{-i}$ is the space of outcomes of the strategy profile. It represents the space of utilities functions (also called payoff functions or reward functions in the reinforcement learning literature) of the players for the strategy profile. Each element is a function that maps the strategy profile to a real number which is the reward for the player: $u \in U,\ u = (u_i, u_{-i})$ where $u_i: S \rightarrow \mathbb{R}$ is the function that maps the strategy profile to the reward of player i. Since the games are iterated up to time T=200, each player receives a reward at every discrete time step $t \in \mathbb{N}$, which is the result of the outcome of the action chosen by the players: $R_t^i = u_i(s_i, s_{-i})$.

Figure 1 shows a depiction of a bi-matrix game.

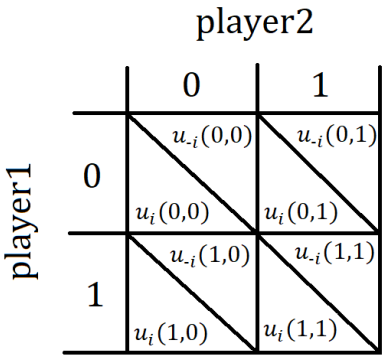


Figure 1 Bi-matrix Game

2.2 MIXED STRATEGIES

Pure strategies are a subset of the more general mixed strategies and can be considered as deterministic strategies, putting probability 1 on one action. Mixed strategies are generalization of pure strategies where the agent puts a non-zero probability on each available action. Formally a mixed strategy of player i at time t can be defined as a probability distribution over pure strategies: $\pi_i^t \in \Delta^i(s_i)$ where $\Delta^i(s_i)$ is the set of probability distributions of player i, the support of which are the actions of player i. Elements of π_i^t have the form $\pi_i^t(s_i)$: $s_i \to \mathbb{R}$, and $\sum_{s_i} \pi_i^t(s_i) = 1$ so that it is a valid probability distribution (McKelvey & Palfrey, 1995).

The joint strategy space of the players is defined as $\Delta = \Delta^i \times \Delta^{-i}$ and an element of this space is the tuple $\tilde{\pi} = (\pi_i, \pi_{-i})$. The joint mixed strategy at time t is the probability distribution of the form: $\tilde{\pi}^t = \prod_i \pi_i^t(s_i)$, which is the product of the individual probability distributions which are assumed to be independent. Mixed strategies are essential for games which do not have a pure strategies equilibrium. It is not guaranteed for a game to have a pure strategy equilibrium but is guaranteed to have a mixed strategy equilibrium (Nash, 1950). For example, the game "rock, paper, scissors" does not have a pure strategy equilibrium but does have a mixed strategy equilibrium which is the uniform probability over actions.

We extend the notion of u_i as in (McKelvey & Palfrey, 1995) to accept mixed strategies as inputs, having domain in Δ by the rule $u_i(\tilde{\pi}^t) = \sum_{s \in S} \tilde{\pi}^t \, u_i(s_i, s_{-i})$, which is the expected utility of player i when playing the joint strategy $\tilde{\pi}^t$ with the opponent at time t.

In the reinforcement learning literature the concept of mixed strategy corresponds to the "policy" and throughout this thesis we will use the naming conventions interchangeably.

2.3 BEST RESPONSE AND NASH EQUILIBRIUM

The concept of Nash Equilibrium was developed by John Nash (Nash, 1950) and refers to the equilibrium point in strategy space where no agent has incentive to deviate from this point, since a deviation would lead the agent to have a lower expected reward for the new strategy, formally: $u_i(\tilde{\pi}^*) \geq u_i(\pi_i, \pi_{-i}{}^*)$, in words the expected utility of agent i playing the optimal joint mixed strategy $\tilde{\pi}^*$ is greater than playing a different joint strategy (Leslie & Collins, 2005). When players know the strategy of the opponent the best they can do is best respond to the opponent's strategy. This is one of the requirements of the learning algorithm called the principle of rationality. Formally a best response for player i is a strategy: $\pi_i{}^* \in Br_i(\pi_{-i}) \in \Delta^i$, where $Br_i: \Delta^{-i} \rightarrow \Delta^i$ is the probability distribution of agent i conditioned on the opponent's mixed strategy that maximizes the expected reward and belongs to the space of probability distributions over the actions of player i and therefore the support of the best response is s_i. The second requirement of the learning algorithm is that it must converge in the limit to a stationary strategy, which means that in the limit the strategy does not change (Bowling & Veloso, 2002). The Nash Equilibrium for a 2x2 matrix game can be computed analytically and the resulting strategies are those that make the opponent indifferent between choosing her actions, therefore make the expected value of the actions equal: $\pi_i{}^*: u_i(s_{-i} = 0, \pi_i) = u_i(s_{-i} = 1, \pi_i)$, where $\pi_i{}^*$ is the optimal strategy for player i.

2.4 QUANTAL BEST RESPONSE EQUILIBRIUM

The quantal best response equilibrium concept was introduced by (McKelvey & Palfrey, 1995) and assumes that players give quantal best responses to the behavior of the others. The assumption is that players make mistakes and take those mistakes into account. In the exponential form of quantal response equilibrium, the probabilities are proportional to an exponential function with the expected payoff times a parameter in the exponent (Selten & Chmura, 2008). The resulting best response strategy is the SoftMax probability mass function having lambda and the expected value of the actions as parameters:

$$1)\ Br^{i}(\pi_{-i}) = \frac{e^{\lambda \hat{u}_i(s_i,\pi_{-i})}}{\sum_{s_i} e^{\lambda \hat{u}_i(s_i,\pi_{-i})}}$$

The assumption of this equilibrium is that estimates of player i of the expected value of the pure strategies $\hat{u}_i(s_i, \pi_{-i})$ are noisy estimates of the true expected value $u_i(s_i, \pi_{-i})$. More formally: $\hat{u}_i(s_i, \pi_{-i}) = u_i(s_i, \pi_{-i}) + \varepsilon_{s_i}$, where ε_{s_i} is the noise for the pure strategy s_i of player i. The error vector that represent the errors of all the pure strategies expected value estimate $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_N)$ is distributed according to the joint density function $f_i(\varepsilon)$. The parameter λ is inversely related to the level of error of the estimates $\hat{u}_i$. When $\lambda \to +\infty$ the estimates have no error. When $\lambda \to 0$ the estimates consist only of error. The quantal best response policy is the same as the one we derive in this thesis and we show that it's the maximum entropy policy that maximizes u_i.

3 REINFORCEMENT LEARNING AND MARKOV GAMES

Reinforcement learning (RL) was first idealized as a framework describing how animals adapt their behavior based on their interactions within an environment. The main concept of this setting is the reinforcement feedback. The reinforcement is a reward feedback for the action that the agent played. (Leslie & Collins, 2005). Reinforcement learning is based on solving a Markov Decision Process (MDP) by converging to an optimal strategy that maximizes the reward of the agent. To achieve this goal two main frameworks have been developed. The first one is called "value-based" RL and relies on the estimation of an optimal value function from which an optimal policy is derived by first starting from a random value function and iteratively modifying it to converge to the optimal one. The second is called "policy based" RL and it first randomly selects a policy for which a value function is then estimated, it subsequently modifies the policy improving the previous one. The value of a policy is the expected sum of rewards the agent receives for the actions played following the current policy. One needs to estimate this value since the dynamics of the environment and the reward function are not know a priori. The Generalized Policy Iteration is a generalization of this process which combines policy evaluation and policy improvement independent of the granularity and other details of the two processes (Sutton & Barto, 2018). One essential concept of reinforcement learning is the exploration-exploitation tradeoff where the agent has to decide whether to select the highest value action or explore by selecting a suboptimal

action with lower value. This tradeoff is related to the uncertainty of the value of actions. The more certain the value of actions the less the agent has to explore.

3.1 ENTROPY REGULARIZED MDPS

RL algorithms based on the value function like Q-learning or SARSA are a type of deterministic algorithm since the action selection is of the form $argmax_a\, Q(a)$ and therefore always selects the action with the highest Q-value. The operator that selects the actions is called hard-max operator. This means that the agent always exploits the acquired knowledge and never explores new options. To obviate this problem one trick has been introduced to make the algorithm explore and works by taking a random action with some probability. Even though this can be considered as a stochastic policy, the exploration is not intrinsic to the policy and the stochasticity does not depend on the rewards. Furthermore this stochastic policy is not rational since it does not select higher value actions more often but selects actions with uniform probability, thereby having high regret. The regret is a measure of the deviation of the rewards obtained from the optimal rewards. Another way of exploring is, with some probability, randomly select an action with probability proportional to its utility. This Framework is called Boltzmann exploration and the probability of selecting the action is computed by the SoftMax operator. This exploration strategy achieves lower regret than random exploration (Cesa-Bianchi, Gentile, Lugosi, & Neu, 2017). The Boltzmann exploration strategy is related to the approach of this thesis, the only difference being that instead of using the SoftMax as the exploration policy, we directly use the SoftMax as the policy. The approach of this thesis is called maximum entropy reinforcement learning. It works by adding a smoothing function to the reward which induces the policy to be stochastic and spreads the probability mass over all the support in proportion to the expected reward of the actions. The action selection operator becomes the SoftMax operator (Asadi & Littman, 2017). This approach resides in the framework of Entropy Regularized Markov Decision Processes (ERMDP) (Haarnoja, Tang, Abbeel, & Levine, 2017). We will now introduce the framework and show the relation to the stateless Markov Games (Littman, 1994).

A Markov Decision Process (MDP) for a single agent is described by the tuple (S, A, T, R) where S is the set of states, A is the set of actions available to the agent and the element $a \in A$ is an action which belongs to the set of actions, R is the reward function:

$R: S \times A \rightarrow \mathbb{R}$ and $T: S \times S \times R \times A \rightarrow \Delta(S)$ is a stochastic transition function where $\Delta(S)$ is the set of probability distributions over S. Solving the finite horizon MDP means fining an optimal policy π^* that maximizes the undiscounted sum of future rewards:

$$2)\ \pi_i{}^* = \ argmax_{\pi_i} \sum_{t=0}^{T} r_t^i$$

where r_t^i is the reward given to the agent i at time t. Another formulation of the objective is the average reward objective which more closely resembles our scenario:

$$3)\ \pi_i{}^* = \ argmax_{\pi_i} \frac{1}{T} \sum_{t=0}^{T} r_t^i$$

The policy π of agent i belongs to the space of policies previously described. In the case of matrix games the set of states S has cardinality 0: $|S| = 0$ since the action taken by the agent does not change the state (as would in an extensive form games which we do not treat) (Busoniu, Babuska, & Schutter, 2010). The transition function is a probability distribution over the state and reward at time t conditioned only on the previous state and action, following the Markov independence property such that all state transitions are independent from each other except for the previous one:

$$T = P(s_t \in S, r_t \in R | s_{t-1} \in S, a_{t-1} \in A)$$

If the strategy of the opponent is fixed we can consider it as part of the environment and therefore include her in the transition function. The standard MDP framework doesn't consider multiple adapting agents and the framework of Markov Games (MG) has been developed to extend the notion of an MDP to multiple agents (Littman, 1994). We can consider matrix games as a stateless MG and therefore we can apply the previous definitions of matrix games also to this restricted class of MG. Formally a Markov Game is a tuple $(S, A^1, \ldots, A^n, r^1, \ldots, r^n, p)$ where S is the state space, A^i is the action space of player i, $r^i: S \times A^1 \times \ldots \times A^n \rightarrow \mathbb{R}$ is the payoff function for player i, p: $S \times R \times A^1 \times \ldots \times A^n \rightarrow \Delta(S)$ is the transition probability map (Hu & Wellman, 2003).

We can show that the objective 3) corresponds to the expected utility $u_i(\tilde{\pi})$ of a joint policy $\tilde{\pi}$ as introduced in chapter 3. To see this consider the expected value of the objective 3):

$$4)\ \mathbb{E}_{s_i,s_{-i}\sim\tilde{\pi}}\left[\frac{1}{T}\sum_{t=0}^{T} r_t^i\right] = \frac{1}{T}\sum_{t=0}^{T}\mathbb{E}_{s_i,s_{-i}\sim\tilde{\pi}}\left[r_t^i\right] = \frac{T}{T}\mathbb{E}_{s_i,s_{-i}\sim\tilde{\pi}}[r^i] = \mathbb{E}_{s_i,s_{-i}\sim\tilde{\pi}}[r^i] = u_i(\tilde{\pi})$$

for a stationary joint distribution $\tilde{\pi}$ of the players. The pure strategies s_i, s_{-i} correspond to the actions that the players take which are sampled according to the joint action distribution $\tilde{\pi}$ previously introduced. We used the linearity of the expectation to transform the expected sum of rewards into the sum of expectations.

If we directly use the objective 3) the resulting policy will be deterministic since the policy that maximizes the objective will put probability one to the action with the highest expected reward, which is the same problem of the standard Q-learning style algorithms discussed earlier. To avoid this we will formulate a new regularized objective by method of Lagrange multipliers (Haarnoja, Zhou, Abbeel, & Levine, 2018). The new regularized average reward objective becomes:

$$5)\ \pi_i{}^* = argmax_{\pi_i}\ \frac{1}{T}\sum_{t=0}^{T} r_t^i + \tau\Omega(\pi_i)$$

where τ is the Lagrange multiplier and $\Omega(\pi)$ is the policy regularizer which acts as a smoothing function. The choice of the regularizer influences the form of the final policy. We chose the entropy function as the regularizer and derive the optimal policy that maximizes the new objective (Leslie & Collins, 2005). The entropy function is an information theoretic concept derived by (Shannon, 1948). The entropy function has the form:

$$6)\ H(x) = -\sum x ln(x)$$

Where ln is the natural logarithm.

The new regularized objective then becomes:

$$7)\ \pi_i{}^* = argmax_{\pi_i}\ \frac{1}{T}\sum_{t=0}^{T} r_t^i - \tau\sum_{s_i}\pi_i(s_i)\ ln(\pi_i(s_i))$$

We now take the expectation of the new objective:

$$8)\ u_i^r(\tilde{\pi}) = \mathbb{E}_{s_i,s_{-i}\sim\tilde{\pi}}\left[\frac{1}{T}\sum_{t=0}^{T} r_t^i - \tau\sum_{s_i}\pi_i(s_i)\ ln(\pi_i(s_i))\right] =$$

$$= u_i(\tilde{\pi}) - \tau\sum_{s_i}\pi_i(s_i)\ ln(\pi_i(s_i))$$

Where $u_i^r(\tilde{\pi})$ is the regularized utility function. We derived the formulation with the same steps as in 4), by noticing that the entropy is not a random variable that depends on the

joint policy and therefore is treated as a constant and by the property of the expectation of a constant can be placed outside the expectation leading to the result of equation 8).

We now analytically derive the optimal policy assuming a stationary opponent strategy by taking the partial derivative of the regularized objective function 8) with respect to agent i's policy and set it equal to zero:

$$9)\ \frac{\partial}{\partial \pi_i(s_i)}\left(u_i(\tilde{\pi}) - \tau \sum_{s_i} \pi_i(s_i) ln(\pi_i(s_i))\right) = 0$$

Computing the derivatives of $u_i(\tilde{\pi})$ and $\tau \sum_{s_i} \pi_i(s_i) ln(\pi_i(s_i))$:

$$\frac{\partial u_i(\tilde{\pi})}{\partial \pi_i(s_i)} = \frac{\partial}{\partial \pi_i(s_i)} \sum_{s_i} \sum_{s_{-i}} \pi_i(s_i)\pi_{-i}(s_{-i}) u_i(s_i, s_{-i}) = \sum_{s_{-i}} \pi_{-i}(s_{-i}) u_i(s_i, s_{-i}) = u_i(s_i, \pi_{-i})$$

$$\frac{\partial}{\partial \pi_i(s_i)} (\tau \sum_{s_i} \pi_i(s_i) ln(\pi_i(s_i)) = \tau(ln\, \pi_i(s_i) + \frac{\pi_i(s_i)}{\pi_i(s_i)})$$

$$11)\ u_i(s_i, \pi_{-i}) - \tau(ln\, \pi_i(s_i) + \frac{\pi_i(s_i)}{\pi_i(s_i)}) = 0$$

$$12)\ ln\, \pi_i(s_i) = \frac{1}{\tau} u_i(s_i, \pi_{-i}) - 1$$

$$13)\ \pi_i(s_i) \propto e^{\frac{1}{\tau} u_i(s_i, \pi_{-i})}$$

We then normalize the function 13) to achieve a valid probability distribution and arrive at the final form of the policy function (Poupart, 2020):

$$14)\ \pi_i(s_i) = \frac{e^{\frac{1}{\tau} u_i(s_i, \pi_{-i})}}{\sum_{s_i} e^{\frac{1}{\tau} u_i(s_i, \pi_{-i})}}$$

We can see that the resulting maximum entropy policy is of the same form as the policy 1) of the quantal response equilibrium where $\lambda = \frac{1}{\tau}$. The parameter τ is called the temperature of the distribution and controls the amount of exploration of the agent. When $\tau \to 0$ the policy becomes deterministic and reverts to the hard-max operator. When $\tau \to +\infty$ the policy becomes a uniform distribution. The requirement of rationality is satisfied by this policy since higher value actions are played with higher probability. It implicitly takes into account the policy of the opponent through the expected value of playing a pure strategy.

3.2 SOFT POLICY ITERATION

Soft Policy Iteration (SPI) is a method used to solve single agent regularized MDPs. As previously defined, matrix games are a special case of an MDP when the MDP is stateless and when the strategy of the opponent doesn't change. We can then utilize this schema to find the optimal value function and optimal policy of the MDP under the assumption that in the short term the strategy of the opponent is stationary. SPI alternates between the soft policy evaluation phase and soft policy improvement phase. Since we do not know the strategy or the actions played by the opponent the agent cannot compute directly the expected value of the policy. Therefore the agent needs to revert to averaging the rewards to estimate the value of the policy. In the following paragraph we introduce the notation used to define the policy iteration scheme.

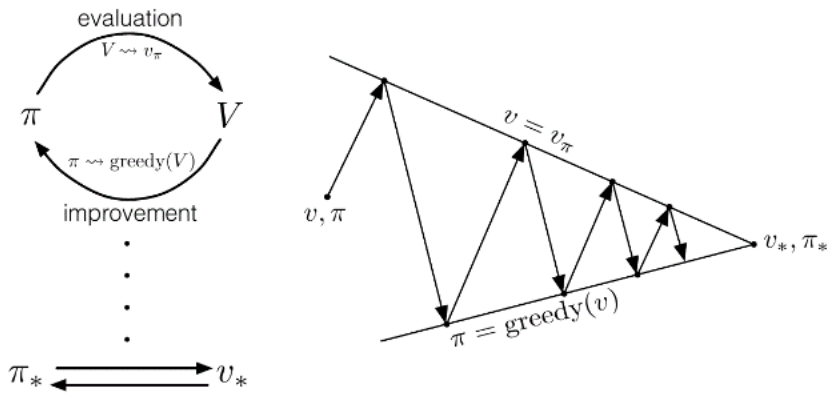


Figure 2 Policy Iteration (Sutton & Barto, 2018)

We now introduce the notation traditionally used in the reinforcement learning literature. We define the true value for the policy π_i of agent i playing against the fixed policy π_{-i} of agent -i as $v^{\pi_i,\pi_{-i}}$ and the optimal value for the optimal joint policy as v^*. $v^{\pi_i,\pi_{-i}}$ is the expected regularized utility of policy π_i: $v_i^{\pi_i,\pi_{-i}} = u_i^r(\pi_i, \pi_{-i})$. The estimate of $v_i^{\pi_i,\pi_{-i}}$ at time t is defined as ${}_tV_i^{\pi_i,\pi_{-i}}$ and the estimate of v_i^* as V_i^*. The estimator used to estimate ${}_iV_t^{\pi_i,\pi_{-i}}$ uses the estimate of the expected value of a pure strategy $Q_i^{\pi_{-i}}(s_i)$. The true value for a pure strategy is defined as $q_i^{\pi_{-i}}(s_i)$ and the optimal q-value q^*. $q_i^{\pi_{-i}}(s_i)$ is the expected regularized reward for playing the pure strategy s_i given the opponent's stationary strategy π_{-i}, therefore $q_i^{\pi_{-i}}(s_i) = u_i^r(s_i, \pi_{-i})$. The estimate of $q_i^{\pi_{-i}}(s_i)$ at time t is ${}_tQ_i^{\pi_{-i}}(s_i)$ and for $q_i{}^*$ *is* Q_i^* (Neto, 2005).

3.2.1 Soft Policy Evaluation: Incremental Average method

To estimate ${}_tQ_i^{\pi_{-i}}(s_i)$ the most straightforward method is to average the past rewards agent i has received up to time t: ${}_{t+1}Q_i^{\pi_{-i}}(s_i) = \frac{1}{t}\sum_{t=0}^{t} r_t^i(s_i) - \tau \sum_{s_i} \pi_i(s_i)\, ln(\pi_i(s_i))$, where $r_t^i(s_i)$ is the reward received by playing the pure strategy s_i. We can drop the regularization term because it doesn't depend on the actions taken and therefore can be treated as a constant. Adding a constant to all the q-values used in the SoftMax policy doesn't change the actions' probabilities. The new formulation is: ${}_{t+1}Q_i^{\pi_{-i}}(s_i) \propto \frac{1}{t}\sum_{t=0}^{t} r_t^i(s_i)$. To compute this average the agent would need to have perfect recall and this is not a realistic assumption for humans. A more efficient way to compute the average is by recursively updating the average estimate throughout time, which results to implementing a running average of the rewards:

$$ {}_{t+1}Q_i^{\pi_{-i}}(s_i) \;=\; \frac{1}{t}\sum_{t=0}^{t} r_t^i(s_i) = \frac{1}{t}(r_t^i(s_i) + \frac{t-1}{t-1}\sum_{t=0}^{t-1} r_t^i(s_i)) = $$

$$ = \frac{1}{t}(r_t^i(s_i) + (t-1)\ {}_tQ_i^{\pi_{-i}}(s_i)) = \frac{1}{t}(r_t^i(s_i) + t\ {}_tQ_i^{\pi_{-i}}(s_i) - \ {}_tQ_i^{\pi_{-i}}(s_i)) $$

$$ = \ {}_tQ_i^{\pi_{-i}}(s_i) \ + \frac{1}{t}(r_t^i(s_i) - \ {}_tQ_i^{\pi_{-i}}(s_i)) $$

Where $(r_t^i(s_i) - \ {}_{t-1}Q_i^{\pi_{-i}}(s_i))$ is the reward prediction error. More generally the update function takes the form ${}_{t+1}Q_i^{\pi_{-i}}(s_i) = \ {}_tQ_i^{\pi_{-i}}(s_i) + \alpha(r_t^i(s_i) - \ {}_{t-1}Q_i^{\pi_{-i}}(s_i))$ where $\alpha \in (0,1]$ is a step-size parameter. This is also called the update rule since it updates the estimate of the Q-value for an action taken at time t. This formulation of the expected value is more robust to non-stationary environments since it can put more weights to recent observations instead of long ones by increasing the parameter α therefore biasing the average towards those observations (Sutton & Barto, 2018). Only the Q-values of the action taken are updated.

3.2.2 Soft Policy Improvement

We now show that the policy has higher value after the update, thereby having monotonic improvements for a stationary opponent strategy π_{-i}. We need to show that the estimates of the value function at time t for the updated policy π_i^{t+1} is greater than the same estimate at time t for the policy π_i^t: ${}_tV_i^{\pi_i^{t+1},\pi_{-i}} \geq {}_tV_i^{\pi_i^t,\pi_{-i}}$. By the definition of ${}_tV_i^{\pi_i^{t+1},\pi_{-i}}$ and ${}_tV_i^{\pi_i^t,\pi_{-i}}$ the inequality becomes:

$$15)\ \mathbb{E}_{s_i,s_{-i}\sim\pi_{-i},\pi_i^{t+1}}[\ {}_tQ_i^{\pi_{-i}}(s_i)\ + \tau\Omega(\pi_i^{t+1})] \geq \mathbb{E}_{s_i,s_{-i}\sim\pi_{-i},\pi_i^t}[\ {}_tQ_i^{\pi_{-i}}(s_i) + \tau\Omega(\pi_i^t)]$$

$$16)\quad \Leftrightarrow \quad \sum_{s_i}\pi_i^{t+1}(s_i)\ {}_tQ_i^{\pi_{-i}}(s_i) - \tau\sum_{s_i}\pi_i^{t+1}(s_i)ln(\pi_i^{t+1}(s_i)) \geq \sum_{s_i}\pi_i^t(s_i)\ {}_tQ_i^{\pi_{-i}}(s_i)\ - \tau\sum_{s_i}\pi_i^t(s_i)ln(\pi_i^t(s_i))$$

We notice that ${}_tQ_i^{\pi_{-i}}$ can be substituted in 16) by rearranging the terms of the policy π_i^{t+1} derived in 14), writing it in terms of the updated policy:

$$\pi_i^{t+1} = \frac{e^{\frac{1}{\tau}\,{}_tQ_i^{\pi_{-i}}(s_i)}}{\sum_{s_i} e^{\ {}_tQ_i^{\pi_{-i}}(s_i)}}$$

$${}_tQ_i^{\pi_{-i}}(s_i) = \tau ln(\pi_i^{t+1}(s_i)\sum_{s_i} e^{\frac{1}{\tau}\,{}_tQ_i^{\pi_{-i}}(s_i)})$$

We now focus on the right side of the inequality 16) and substitute ${}_tQ_i^{\pi_{-i}}(s_i)$ in the equation:

$$17)\qquad \sum_{s_i}\pi_i^t(s_i)\,\tau ln(\pi_i^{t+1}(s_i)\sum_{s_i} e^{\frac{1}{\tau}\,{}_tQ_i^{\pi_{-i}}(s_i)}) - \tau\sum_{s_i}\pi_i^t(s_i)ln(\pi_i^t(s_i)) =$$

$$18)\ = \sum_{s_i}\tau\pi_i^t(s_i)\,[ln(\pi_i^{t+1}(s_i))\ + ln(\sum_{s_i} e^{\frac{1}{\tau}\,{}_tQ_i^{\pi_{-i}}(s_i)})] - \tau\sum_{s_i}\pi_i^t(s_i)ln(\pi_i^t(s_i))$$

$$19) = \sum_{s_i}\tau\pi_i^t(s_i)\,ln(\pi_i^{t+1}(s_i))\ + \tau\pi_i^t(s_i)ln(\sum_{s_i} e^{\frac{1}{\tau}\,{}_tQ_i^{\pi_{-i}}(s_i)}) - \tau\sum_{s_i}\pi_i^t(s_i)ln(\pi_i^t(s_i))$$

We notice that $\sum_{s_i}\tau\pi_i^t(s_i)\,ln(\pi_i^{t+1}(s_i)) - \tau\sum_{s_i}\pi_i^t(s_i)ln(\pi_i^t(s_i))$ can be rewritten as $\tau\sum_{s_i}\pi_i^t(s_i)ln(\frac{\pi_i^{t+1}(s_i)}{\pi_i^t(s_i)})$, which is the negative KL-divergence of policy $\pi_i^t(s_i)$ from policy

$\pi_i^{t+1}(s_i)$ multiplied by the Lagrange multiplier. The KL divergence is a measure of dissimilarity between two probability distributions: $D_{KL}(\pi_i^t||\pi_i^{t+1})$. We then rewrite the expression 19) in terms of the D_{KL}:

$$20) \ -\tau D_{KL}(\pi_i^t||\pi_i^{t+1}) + \sum_{s_i} \tau\pi_i^{t+1}(s_i) ln(\sum_{s_i} e^{\frac{1}{\tau}\ {}_tQ_i^{\pi_{-i}}(s_i)})$$

Since the negative KL-divergence is always negative, dropping it will make the resulting expression grater:

$$21) \quad 20) \leq \tau \sum_{s_i} \tau\pi_i^{t+1}(s_i) ln(\sum_{s_i} e^{\frac{1}{\tau}\ {}_tQ_i^{\pi_{-i}}(s_i)})$$

Since by definition: $ln(\sum_{s_i} e^{\frac{1}{\tau}\ {}_tQ_i^{\pi_{-i}}(s_i)}) = ln(e^{\frac{1}{\tau}\ {}_tQ_i^{\pi_{-i}}(s_i)}) - ln(\pi_i^{t+1}(s_i))$, we then substitute it into 22) resulting in:

$$23) \qquad \sum_{s_i} \pi_i^{t+1}(s_i)\tau[ln(e^{\frac{1}{\tau}\ {}_tQ_i^{\pi_{-i}}(s_i)}) \ - \ ln(\pi_i^{t+1}(s_i)))]$$

$$= \ \sum_{s_i} \pi_i^{t+1}(s_i) \ {}_tQ_i^{\pi_{-i}}(s_i) - \tau\pi_i^{t+1}(s_i) ln\left(\pi_i^{t+1}(s_i)\right)$$

$$= \ \mathbb{E}_{s_i\sim\pi_{-i},\pi_i^{t+1}}[\ {}_tQ_i^{\pi_{-i}}(s_i)\ ] + \tau H(\pi_i^{t+1}) = \ {}_tV_i^{\pi_i^{t+1},\pi_{-i}}$$

Thereby proving the inequality (Poupart, 2020).

3.3 PRACTICAL IMPLEMENTATION

To satisfy the requirement of stationarity we linearly decrease the exploration parameter τ of agents by ϵ = 0.001 after each policy update up until an optimal value. We update the policy after taking the action. We don't change the α parameter. The optimal value of the parameters was found by grid-search.

3.4 COMPARISON WITH THE ROTH & EREV MODEL

The model developed by (Erev & Roth, 1998) is the standard model of reinforcement. The basic model of reinforcement has similar components to the one discussed in this thesis. The main concept the model is that each agent n has initial propensities corresponding to each of his kth pure strategies at time t defined as $q_{nk}(t)$. The concept of propensity is similar to the Q-value for the pure strategy s_i: ${}_tQ_i^{\pi-i}(s_i)$ where $s_i = k$ and n=i. The propensities are then updated after taking actions k and receiving the positive payoff x by the reinforcement function R(x) = x – $x_{min,}$ where x_{min} is the smallest possible payoff. The update of the propensity of each pure strategy k is:

$$q_{nj}(t+1) = \begin{cases} q_{nj}(t) + R(x) & if\ j = k \\ q_{nj}(t) & otherwise. \end{cases}$$

thereby only updating the action taken.

The probability of choosing a kth pure strategy of player n at time t is: $p_{nk}(t) = \frac{q_{nk}(t)}{\sum_j q_{nj}(t)}$.

The limitation of this probability function is that it cannot accept negative q-values. The policy in this thesis however does not have this limitation (Leslie & Collins, 2005). The more advanced form of $q_{nj}(t+1)$ implements the "forgetting" property controlled by the parameter ∅. The update takes the form:

$$q_{nj}(t+1) = (1-\emptyset)q_{nj}(t) + E(j, R(x))$$

where $E(j, R(x))$ is a function that determines how the experience of playing strategy k and receiving reward R(x) is generalized to update each strategy. This is similar to the incremental average approach of this thesis because α has the same role of ∅.

4 STATISTICAL ANALYSIS OF PLAYERS STRATEGIES

4.1 EXPERIMENTAL DESIGN

4.2 PROCEDURE

The procedure of the experiments is the same as the one in (Selten & Chmura, 2008). For each game, n agents play the role of player role player1 and n play the role of player2. The agents are randomly matched at every iteration of the playing period. Randomly matching players is a standard experimental design procedure which has the effect of reducing the correlation between agents' strategies, thereby avoiding a correlated equilibrium. The agents play for a total of 200 iterations. The experimental data were obtained in 54 sessions with 16 subjects each. In each session there were 2 groups of 8 subjects divided in 4 subjects in the role of player1 and 4 in the role of player2.

4.3 GAMES

The matrix games used to evaluate the algorithm are 2x2 normal form games. The first 6 games are constant sum games, where the total amount of reward for the players is constant. This type of games are competitive games since a player gains what the opponent loses. The other 6 games are non-constant sum games. These games are derived by adding a constant to the rewards of player1 in the column of player2 and by adding a constant to the rewards of player2 in the column of player1. The games form a pair of constant and non-constant games. The paired games have the same best response structure.

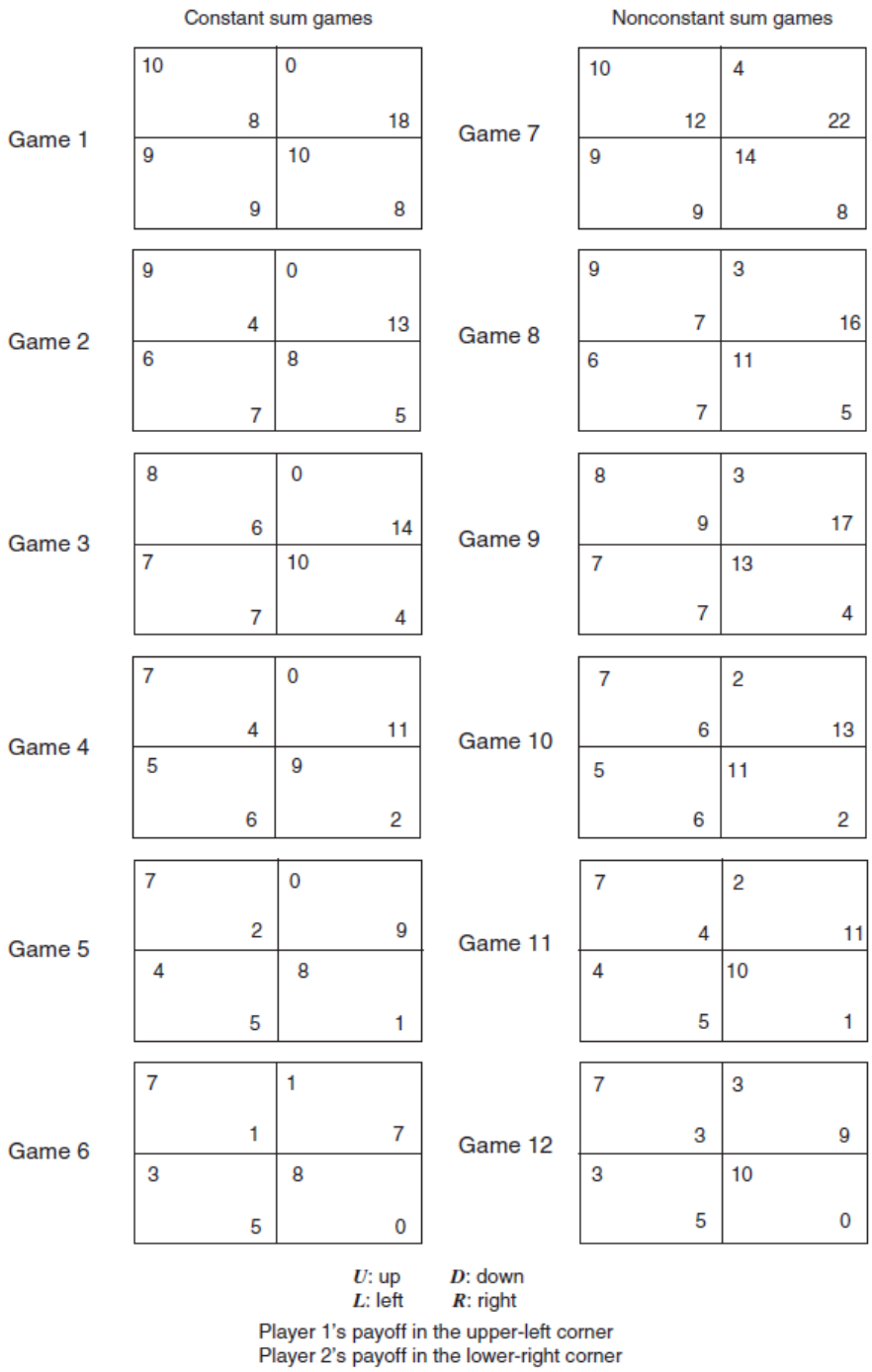


Figure 3 Matrix Games used for evaluation from figure 5 of (Selten & Chmura, 2008)

TABLE 1—FIVE STATIONARY CONCEPTS TOGETHER WITH THE OBSERVED RELATIVE FREQUENCIES FOR EACH OF THE EXPERIMENTAL GAMES

		Nash equilibrium	Quantal response equilibrium	Action-sampling equilibrium	Payoff-sampling equilibrium	Impulse balance equilibrium	Observed average of 12 observations
Game 1	U	0.091	0.070	0.057	0.071	0.088	0.079
	L	0.909	0.882	0.664	0.645	0.580	0.690
Game 2	U	0.182	0.172	0.185	0.184	0.172	0.217
	L	0.727	0.711	0.619	0.569	0.491	0.527
Game 3	U	0.273	0.250	0.137	0.152	0.161	0.198
	L	0.909	0.898	0.753	0.773	0.765	0.793
Game 4	U	0.364	0.348	0.286	0.285	0.259	0.286
	L	0.818	0.812	0.679	0.726	0.710	0.736
Game 5	U	0.364	0.354	0.286	0.307	0.297	0.327
	L	0.727	0.721	0.679	0.654	0.628	0.664
Game 6	U	0.455	0.449	0.448	0.427	0.400	0.445
	L	0.636	0.634	0.613	0.597	0.600	0.596
		Nash equilibrium	Quantal response equilibrium	Action-sampling equilibrium	Payoff-sampling equilibrium	Impulse balance equilibrium	Observed average of 6 observations
Game 7	U	0.091	0.070	0.057	0.056	0.104	0.141
	L	0.909	0.882	0.664	0.691	0.634	0.564
Game 8	U	0.182	0.172	0.185	0.222	0.258	0.250
	L	0.727	0.711	0.619	0.601	0.561	0.586
Game 9	U	0.273	0.250	0.137	0.154	0.188	0.254
	L	0.909	0.898	0.753	0.767	0.764	0.827
Game 10	U	0.364	0.348	0.286	0.308	0.304	0.366
	L	0.818	0.812	0.679	0.731	0.724	0.699
Game 11	U	0.364	0.354	0.286	0.339	0.354	0.331
	L	0.727	0.721	0.679	0.651	0.646	0.652
Game 12	U	0.455	0.449	0.448	0.405	0.466	0.439
	L	0.636	0.634	0.613	0.600	0.604	0.604

Figure 4 Games equilibrium table 1 of (Selten & Chmura, 2008)

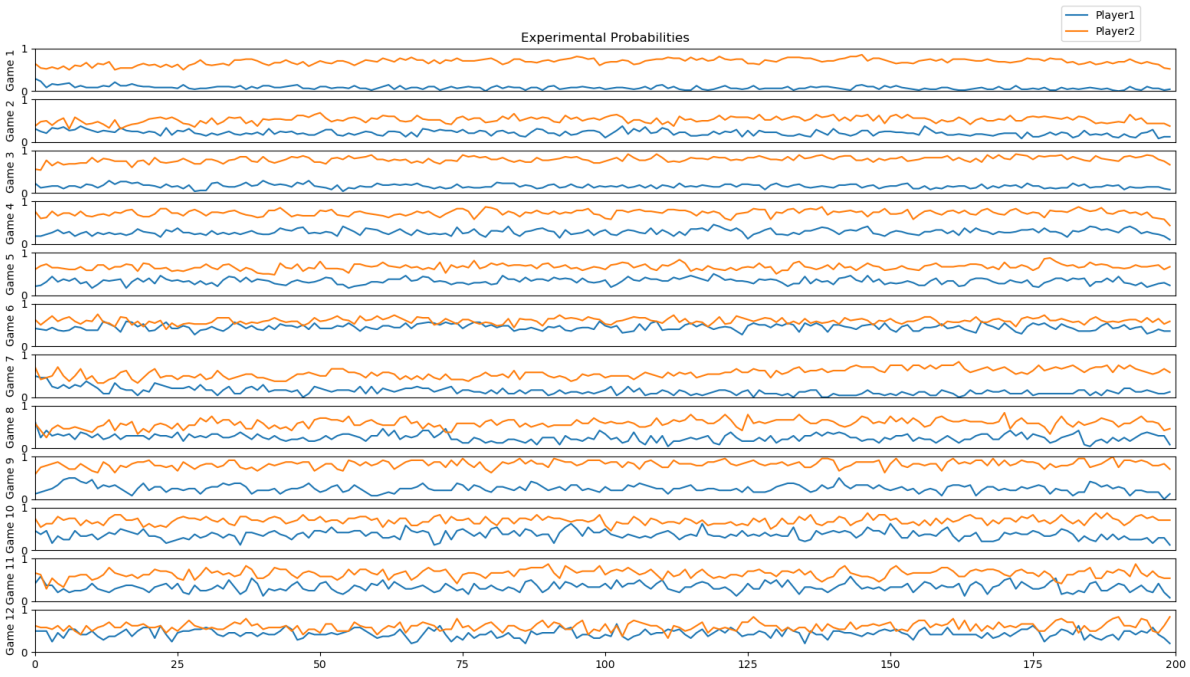


Figure 5 Experimental human strategies

4.4 MEAN SQUARE DEVIATION SCORE

We evaluate the simulation results with the Mean Square Distance (MSD) measure from (Selten & Chmura, 2008), which measures the distance between the expected agent trajectory and the experimental data expected trajectory. Formally, the MSD for a game G and a simulation run R (which can be considered an independent subject group) is:

$$MSD_G = \frac{1}{R}\sum_{r}^{R}(\bar{\pi}_{r,G}(s_i) - \bar{\pi}_{expG}(s_i))^2$$

where $\bar{\pi}_{r,G}(s_i)$ is the average of the average player strategy over a time period: $\bar{\pi}_{r,G}(s_i) = \frac{1}{T}\sum_{t}^{T}\frac{1}{|agents|}\sum_{i\in Agents}\pi_{i,r,G}^{t}(s_i)$ and $\bar{\pi}_{expG}(s_i)$ is the experimental players average strategy for game G, which is calculated with the same formula as $\bar{\pi}_{i,r,G}(s_i)$. The total MSD is the average of MSD_G for games G: $MSD = \frac{1}{|G|}\sum_{G} MSD_G$. The MSD can be decomposed into sampling variance (S) and theory specific component (T):

$$S_G = \frac{1}{2}\sum_{s_i}\frac{1}{R}\sum_{r}^{R}(\bar{\pi}_{r,G}(s_i) - \bar{\pi}_G(s_i))^2$$

$$T_G = (\bar{\pi}_G(s_i) - \bar{\pi}_{exp,G}(s_i))^2$$

Where $\bar{\pi}_G(s_i)$ and $\bar{\pi}_{exp,G}(s_i)$ are the average run game probabilities for simulated agents and experimental players respectively: $\bar{\pi}_G(s_i) = \frac{1}{R}\sum_r \bar{\pi}_{rG}(s_i)$.

We follow the approach of (Selten & Chmura, 2008) and divide S and T by the first 6 constant sum games (MSD_C) and the other nonconstant games (MSD_N):

$$MSD_C = S_C + T_C$$

$$MSD_N = S_N + T_N$$

$$MSD = S + T$$

4.5 GRID SEARCH OF OPTIMAL PARAMETERS

We performed grid search to find the parameters that minimize the Mean Square Deviation of the simulated players trajectories and the experimental trajectories. The results of the parameters are shown in figure7: the optimal values were $\tau_{t=0}$ =1.8 for the initial τ, $\tau_{t=T} = 0.55$ for final τ, α =0.33 and ϵ = 0.001, for a MSD_{tot}=0.11273. The alpha parameter mean that on average the agent looks at just 3 previous time steps to estimate the expected value of an action. We searched over $\tau_{t=0} = (0.5, 1.8)$, $\tau_{t=T} = (0.1,1)$, $\epsilon = (0.001,0.01)$, α = (0.05, 0.65) by subdividing each axis into equally spaced intervals. The main drawback of having to search by grid-search over many parameters is the computational time complexity that is exponential in the number of variables to estimate.

4.6 SIMULATION RESULTS AND COMPARISON TO HUMAN PLAYERS TRAJECTORIES

The following plots refer to the averaged 8 simulation runs with 4 agents per player type.

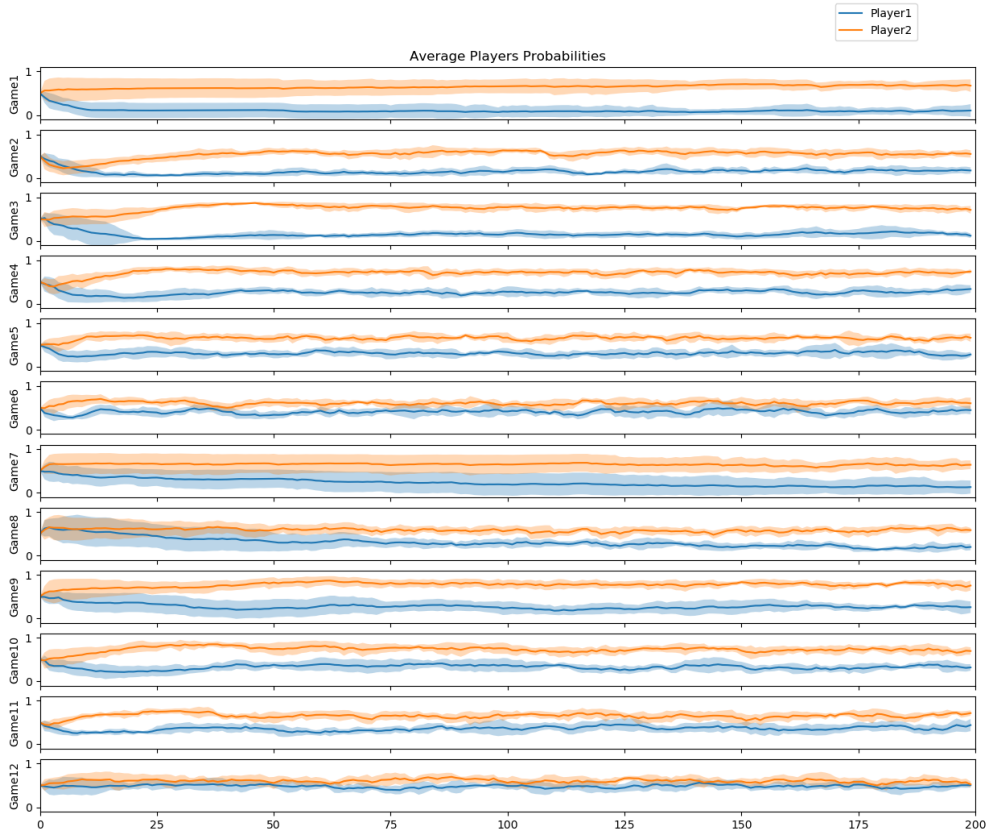


Figure 6 Average Mean and Variance over 8 runs of 4 simulated agents per player's probabilities trajectories.

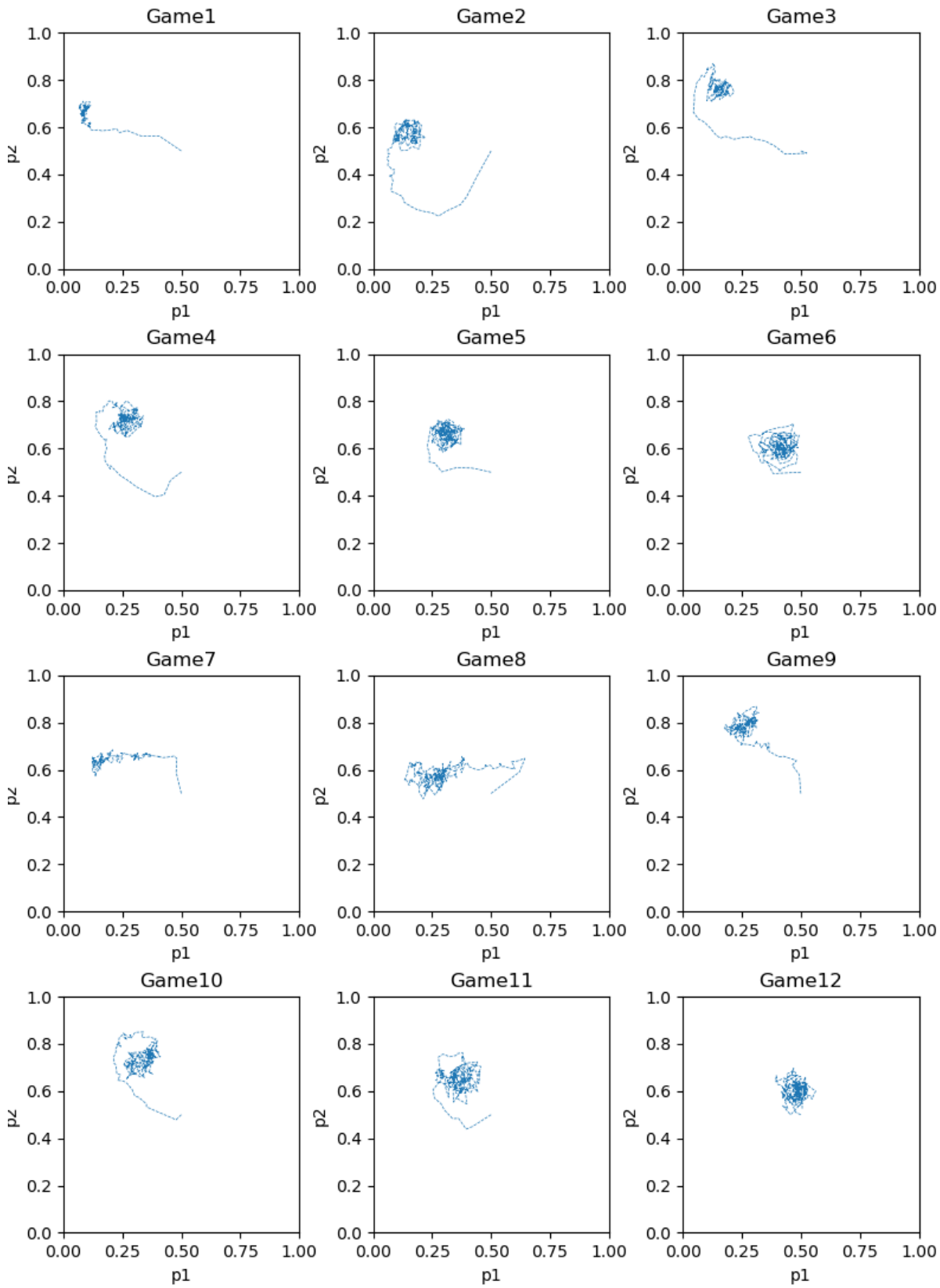


Figure 7 Players Dynamics plotted in phase space

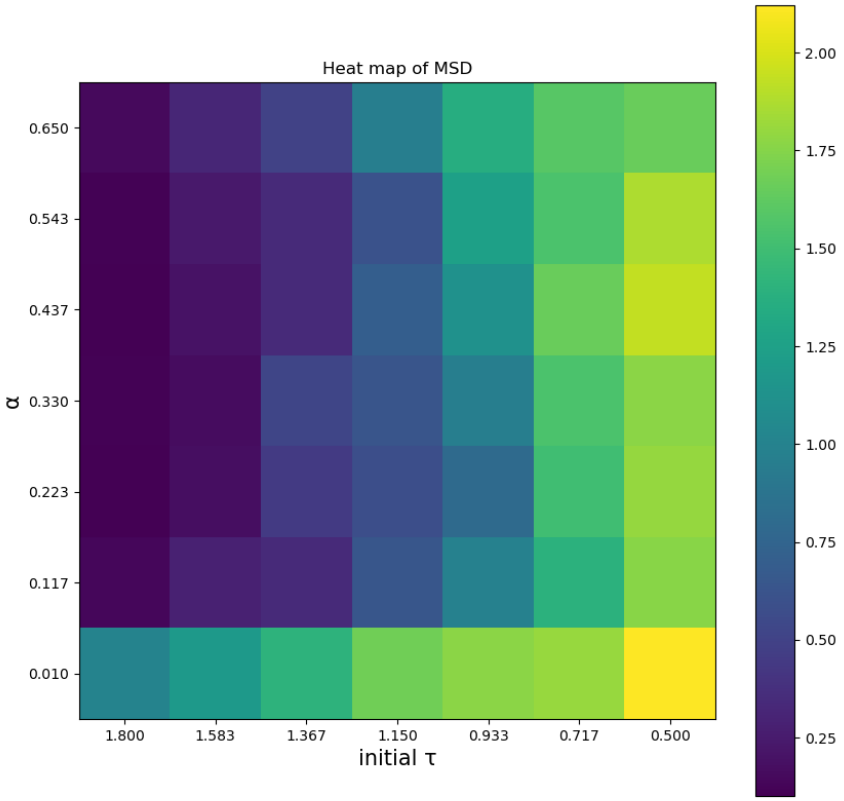
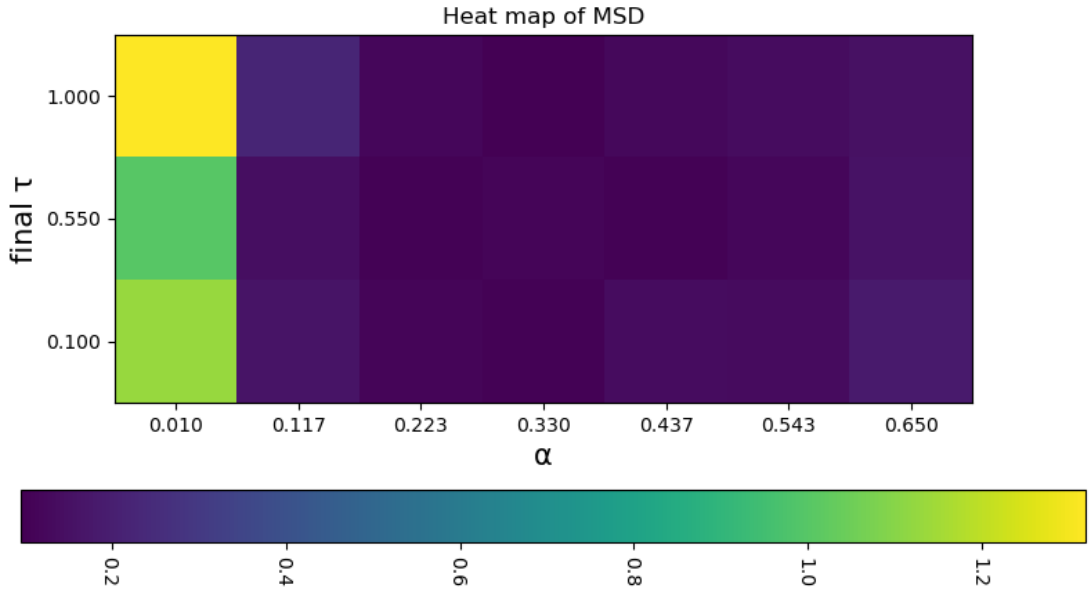


Figure 8 MSD for initial τ_T and α



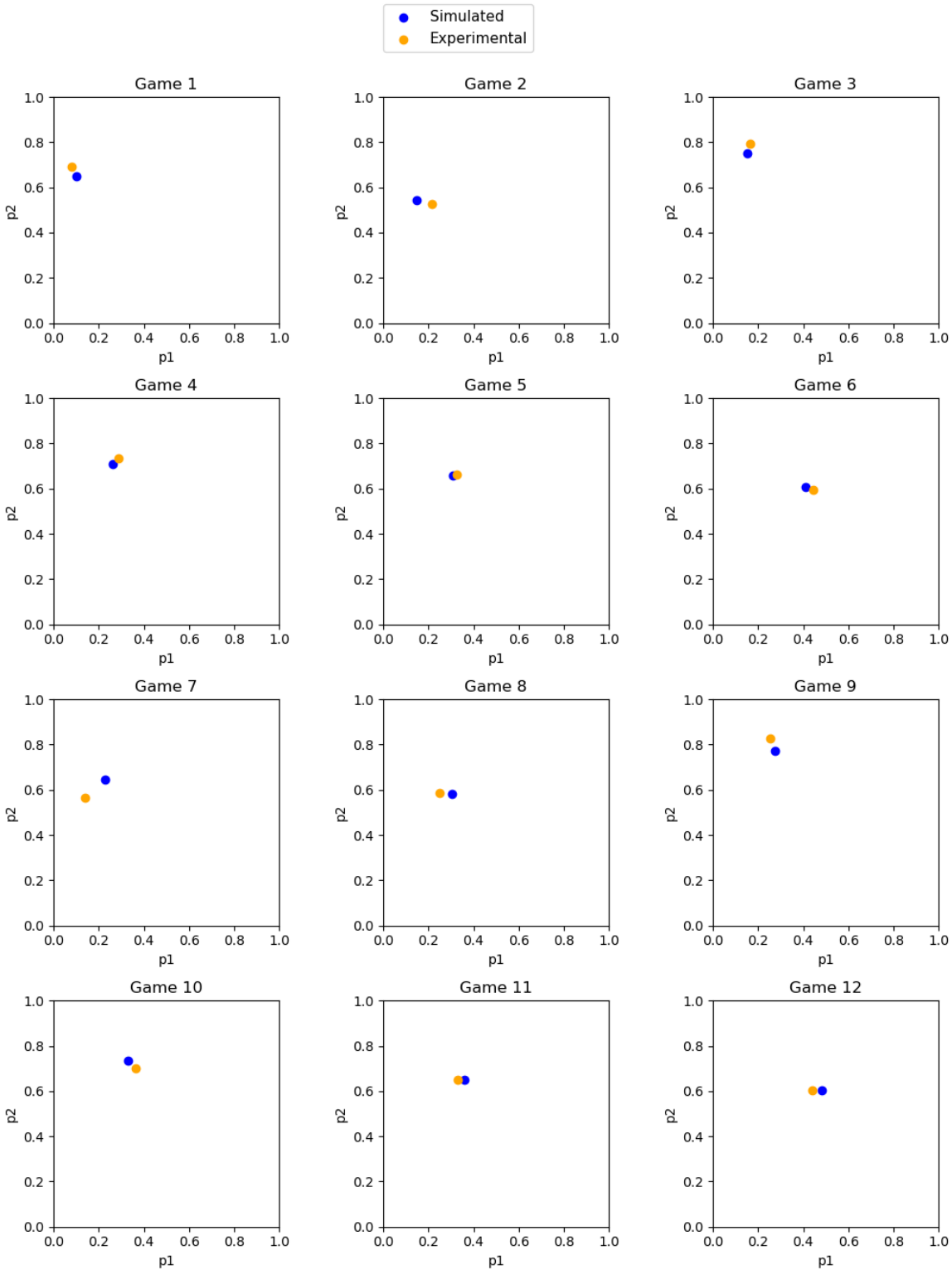


Figure 9 Expected Trajectory Equilibrium for Simulated and Experimental players of last 50 timesteps

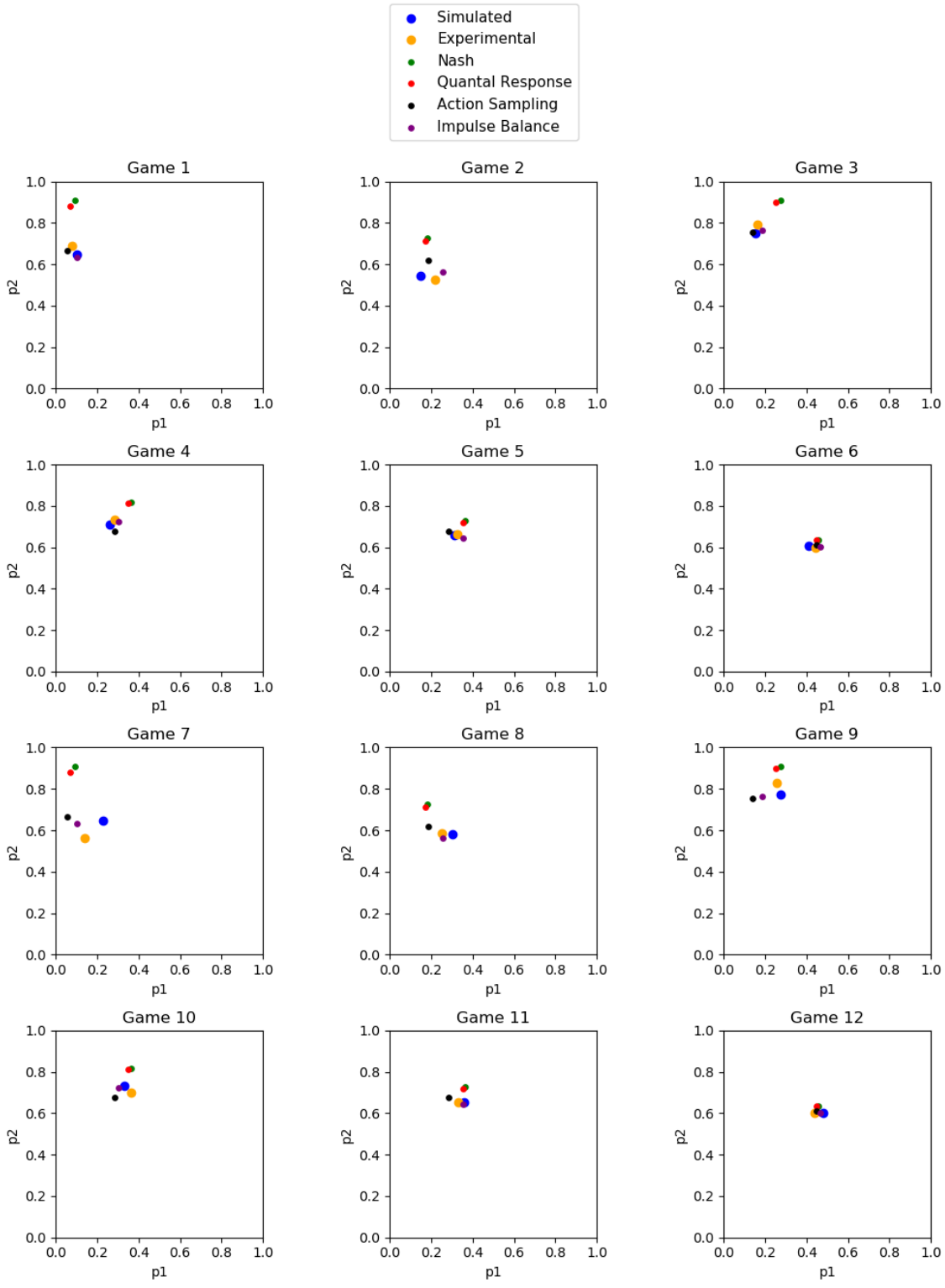


Figure 10 Comparison with other theories

	Player1 simulated	Player1 experimental	Player2 simulated	Player2 experimental
game1	0.095	0.079	0.685	0.69
game2	0.172	0.217	0.561	0.527
game3	0.17	0.164	0.762	0.793
game4	0.282	0.286	0.704	0.736
game5	0.322	0.327	0.655	0.664
game6	0.419	0.445	0.608	0.596
game7	0.142	0.141	0.626	0.564
game8	0.199	0.25	0.581	0.586
game9	0.276	0.254	0.787	0.827
game10	0.318	0.366	0.713	0.699
game11	0.371	0.331	0.65	0.652
game12	0.474	0.439	0.588	0.604

Figure 11 Equilibrium Results averaged over the last 50 timesteps

(Selten & Chmura, 2008)

TABLE 2—SQUARED DISTANCES OF THE FIVE THEORIES

	Nash equilibrium	Quantal response equilibrium	Action-sampling equilibrium	Payoff-sampling equilibrium	Impulse balance equilibrium	Sampling variance
Game 1	0.0572	0.0460	0.0103	0.0112	0.0213	0.00909
Game 2	0.0483	0.0428	0.0164	0.0098	0.0102	0.00693
Game 3	0.0321	0.0250	0.0087	0.0057	0.0073	0.00523
Game 4	0.0169	0.0137	0.0072	0.0041	0.0054	0.00403
Game 5	0.0149	0.0136	0.0115	0.0100	0.0117	0.00953
Game 6	0.0042	0.0039	0.0027	0.0028	0.0045	0.00246
Game 7	0.1237	0.1082	0.0189	0.0253	0.0081	0.00178
Game 8	0.0298	0.0269	0.0106	0.0063	0.0060	0.00531
Game 9	0.0212	0.0192	0.0332	0.0276	0.0224	0.01409
Game 10	0.0208	0.0196	0.0134	0.0109	0.0111	0.00665
Game 11	0.0098	0.0084	0.0059	0.0032	0.0036	0.00307
Game 12	0.0045	0.0042	0.0033	0.0047	0.0039	0.00317

	MSD	T	S
game1	0.01816	0.00028	0.01788
game2	0.00572	0.00315	0.00258
game3	0.00655	0.00099	0.00555
game4	0.00388	0.00099	0.00289
game5	0.00155	0.0001	0.00145
game6	0.00224	0.00079	0.00144
game7	0.04869	0.00385	0.04484
game8	0.00431	0.00266	0.00165
game9	0.00787	0.0021	0.00577
game10	0.00414	0.00244	0.00171
game11	0.00678	0.00161	0.00517
game12	0.00285	0.00149	0.00135

Table 12 Table of MSD, Theory component (T) and Sampling variance(S) over the last 50 timesteps

	MSD	T	S
Constant	0.05786	0.01103	0.04683
Nonconstant	0.12381	0.02542	0.09839
Total	0.18487	0.03817	0.1467

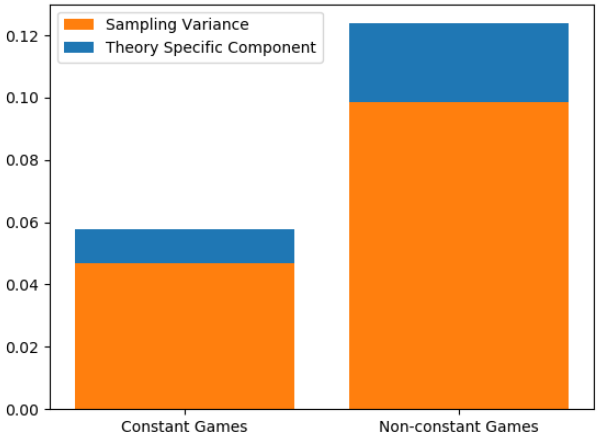


Figure 13 MSD of 200 timesteps for Constant, Nonconstant and Total games

	MSD	T	S
Constant	0.03586	0.00551	0.03035
Nonconstant	0.07402	0.01344	0.06058
Total	0.11273	0.02045	0.09228

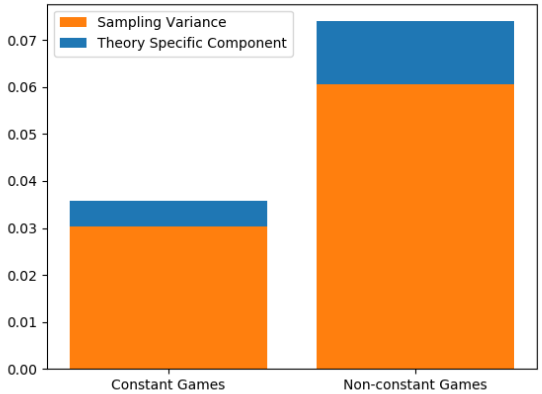


Figure 14 MSD of the last 50 timesteps for Constant, Nonconstant and Total Games

5 CONCLUSIONS

The aim of this thesis was to investigate whether the framework of reinforcement learning can approximate human learning in games. We introduced the Entropy Regularized Reinforcement Learning framework and showed that it can be applied to stateless Markov Games. We then showed the relation of this framework to the Quantal Best Response equilibrium by deriving the same softmax policy. We then implemented the algorithm in python and search the optimal parameters by grid search that best approximate humans' trajectories. The simulation results showed that this framework can approximate with reasonable accuracy the gameplays of human players and that it's competitive with other frameworks. The high degrees of freedom, namely the beginning and ending exploration parameter, the learning rate and the decay of exploration, of this framework allow fitting the experimental data in an accurate manner. The exploration decay parameter ϵ was essential to make the algorithm converge, which means that also humans explore less over time because they are more certain over the expected reward of each action. One drawback of the developed model is that the exploration is not a function of the variance of the expected value of an action which would make it more dynamical and can be an unexplored question for further research.

6 REFERENCES

Asadi, K., & Littman, M. L. (2017). An Alternative Softmax Operator for Reinforcement Learning. *Proceedings of the 34th International Conference on Machine Learning, 70*, pp. 243-252.

Bowling, M., & Veloso, M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence, 136*(2), pp. 215-250. doi:10.1016/S0004-3702(02)00121-2

Brown, G. (1951). Iterative solution of games by Fictitious Play. *Activity Analysis of Production and Allocation*, pp. 374-376.

Busoniu, L., Babuska, R., & Schutter, B. D. (2010). Multi-agent Reinforcement Learning: An Overview. In *Innovations in Multi-Agent Systems and Applications - 1* (pp. pp. 183-221).

Cesa-Bianchi, N., Gentile, C., Lugosi, G., & Neu, G. (2017). Boltzmann Exploration Done Right. *Proceedings of the 31st International Conference on Neural Information Processing*, pp. 6287-6296.

Erev, I., & Roth, A. E. (1998). Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria. *American Economic Review, 88*(4), pp. 848-881.

Fudenberg, D., & Levine, D. K. (2007). An economist's perspective on multi-agent learning. *Artificial Intelligence, Volume 171*(Issue 7), Pages 378-381. doi:10.1016/j.artint.2006.11.006

Haarnoja, T., Tang, H., Abbeel, P., & Levine, S. (2017). Reinforcement learning with deep energy-based policies. *Proceedings of the 34th International Conference on Machine Learning, 70*, pp. 1352-1361.

Hernandez-Leal, P., Kaisers, M., & Baarslag, T. (2019, 03 11). *A Survey of Learning in Multiagent Environments: Dealing with Non-Stationarity.* Retrieved from arxiv.org: arXiv:1707.09183v2

Hu, J., & Wellman, M. P. (2003). Nash Q-Learning for General-Sum Stochastic Games. *The Journal of Machine Learning Research, 4*, pp. 1039-1069.

Kapoor, S. (2018). *Multi-Agent Reinforcement Learning: A Report on Challenges and Approaches.* Retrieved from https://arxiv.org: /abs/1807.09427

Leslie, D. S., & Collins, E. J. (2005). Individual Q-Learning in Normal Form Games. *SIAM Journal on Control and Optimization, 44*(2), 495-514. doi:10.1137/S0363012903437976

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *In Proceedings of the Eleventh International Conference on Machine Learning* (pp. pp. 157-163). Morgan Kaufmann.

McKelvey, R. D., & Palfrey, T. R. (1995). Quantal Response Equilibria for Normal Form Games. *Games and Economic Behavior, 10*(1), pp. 6-38. doi:10.1006/game.1995.1023

Nachum, O., Norouzi, M., Xu, K., & Schuurmans, D. E. (2017). Bridging the gap between value and policy based reinforcement learning. *Proceedings of the 31st International Conference on Neural Information Processing*, pp. 2772-2782.

Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences, 36*(1), pp. 48-49.

Neto, G. (2005). *From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods.*

Neumann, J. v., & Morgenstern, O. (1944). *Theory of games and economic behavior.* Princeton University Press.

Poupart, P. (2020). *CS885 Spring 2020 - Reinforcement Learning.* Retrieved from https://cs.uwaterloo.ca/: ~ppoupart/teaching/cs885-spring20/slides/cs885-module2.pdf

Selten, R., & Chmura, T. (2008). Stationary Concepts for Experimental 2x2-Games. *The American Economic Review, 98*(3), pp. 938–966.

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal, 27*(3), pp. 379-423.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

Van Der Wal, J. (1891). Stochastic dynamic programming. *Mathematical Centre Tracts*.

7 APPENDIX

7.1 SOFTWARE AND IMPLEMENTATION DETAILS

The pseudocode for the algorithm is as follow:

```
1) Initialize agents Q-values and Game Matrices:
for player in |players|:
        player_Q_values = [0,0]
for i in |game_matrice|:
        game_matrix = game_matrix(player, action1, action2) ∈ Game Matrices
2) For each game and period make agents play
for game_matrix in game_matrices:
        for i = 1…200 do:
                agents_list = random_matching()
                for player1, player2 in agents_list:
                        action_player1 = player1.take_action()
                        action_player1 = player1.take_action()
                        reward_player1 = game_matrix(player1, action_player1, action_player2)
                        reward_player1 = game_matrix(player2, action_player1, action_player2)
                        player1.update_q_values(reward_player1)
                        player1.update_q_values(reward_player2)
                        player1.update_policy()
                        player2.update_policy()
```

We implemented the algorithm in python3 programming language utilizing the mathematical library NumPy and the plotting library Matplotlib. The code consists in two main classes, the agent class that implements the agent logic and the environment class that implements the game play logic. The main.py file is used to implement the main logic and run the program. The analyzer.py file contains the grid search function used to find the optimal parameters that minimize the MSD and some other plotting functions. In the following pages we paste the code produced.

7.2 MAIN FILE

```
import matplotlib.pyplot as plt
import numpy as np
from src.agent import Agent
from src.environment import Environment
from src.analyzer import (
    averages,
    plot_data,
    grid_search,
    mean_squared_error,
    show_params_space,
    plot_avg_distance,
    plot_average_run,
    open_results,
    load_obj,
)
import argparse
import os
import pickle

parser = argparse.ArgumentParser()
parser.add_argument("--simulate", type=bool, help="Sun simulation :bool",
default=False)
parser.add_argument("--iter", type=int, help="number of iterations :int",
default=200)
parser.add_argument(
    "--agents", type=int, help="number of agents per player type :int",
default=2
)
parser.add_argument(
    "--strategy",
    type=str,
    help='strategy of agent: "max"| "boltzmann" | "entropy" :string',
    default="entropy",
)
parser.add_argument(
    "--d_expl", type=bool, help="Decrease Exploration :bool", default=True
)
parser.add_argument(
    "--exp",
    type=bool,
    help="Whether to decrease exploration exponentially or linearly :bool",
    default=False,
)
parser.add_argument(
    "--decay_rate",
    type=float,
    help="Rate of exploration decrease of exponential or linear decay
:float",
    default=0.005,
)
parser.add_argument(
    "--save", type=bool, help="save game history plots :bool",
default=False
)
parser.add_argument("--runs", type=int, help="number of simulation runs",
default=1)
parser.add_argument(
    "--save_fig",
    type=bool,
    help="whether to save simulations images :bool",
```

```
    default=False,
)
parser.add_argument(
    "--show_fig",
    type=bool,
    help="whether to show simulations images :bool",
    default=False,
)
parser.add_argument(
    "--sum_rewards",
    type=bool,
    help="whether to plot sum of rewards or rewards per iteration :bool",
    default=False,
)
parser.add_argument(
    "--grid_search",
    type=bool,
    help="grid search of optimal parameter :bool",
    default=False,
)
parser.add_argument(
    "--plot_exp",
    type=bool,
    help="whether to show experimental games players' probabilities :bool",
)
parser.add_argument(
    "--t", type=float, help="temperature parameter of agents :float",
default=2
)
parser.add_argument(
    "--a", type=float, help="alpha parameter of agents :float", default=0.1
)
parser.add_argument(
    "--open",
    type=int,
    help="show best parameter runs in folder n of games histories :int",
)
parser.add_argument("--intervals", type=int, help="search intervals :int",
default=10)
parser.add_argument(
    "--begin", type=int, help="beginning time from which to calculate MSD",
default=0
)
parser.add_argument(
    "--mse",
    type=int,
    help="Calculate mean square deviation of histories in file config.pkl
in folder n :int",
)
parser.add_argument(
    "--from_game", type=int, help="If mse then begin from this game",
default=0
)
parser.add_argument(
    "--to_game", type=int, help="If mse then end to this game :int",
default=11
)
parser.add_argument(
    "--equilibria", type=bool, help="Show all equilibrium in plot :bool",
default=False
)
```

```
parser.add_argument(
    "--compare",
    type=int,
    help="MSD comparison of constant and non-constant sum games of file
config.pkl in folder n :int",
    default=False,
)
parser.add_argument("--plot_avg", type=int, help="Plot equilibrium
results")

args = parser.parse_args()
simulate = args.simulate
iterations = args.iter
agents = args.agents
strategy = args.strategy
d_exp = args.d_expl
exp_decay = args.exp
decay_rate = args.decay_rate
save = args.save
runs = args.runs
save_fig = args.save_fig
show_fig = args.show_fig
grid = args.grid_search
sum_rewards = args.sum_rewards
experimental = args.plot_exp
open_ = args.open
intervals = args.intervals
begin = args.begin
mse = args.mse
from_game = args.from_game
to_game = args.to_game
equilibria = args.equilibria
compare_games = args.compare
plot_avg = args.plot_avg
t = args.t if args.t > 0 else 1.6
a = args.a if 0 < args.a < 1 else 0.4

strategy_ = {"entropy": 2, "boltzmann": 1, "max": 0}
strategy = strategy_[strategy]


def main():
    games = [
        [[[10, 0], [9, 10]], [[8, 18], [9, 8]]],
        [[[9, 0], [6, 8]], [[4, 13], [7, 5]]],
        [[[8, 0], [7, 10]], [[6, 14], [7, 4]]],
        [[[7, 0], [5, 9]], [[4, 11], [6, 2]]],
        [[[7, 0], [4, 8]], [[2, 9], [5, 1]]],
        [[[7, 1], [3, 8]], [[1, 7], [5, 0]]],
        [[[10, 4], [9, 14]], [[12, 22], [9, 8]]],
        [[[9, 3], [6, 11]], [[7, 16], [7, 5]]],
        [[[8, 3], [7, 13]], [[9, 17], [7, 4]]],
        [[[7, 2], [5, 11]], [[6, 13], [6, 2]]],
        [[[7, 2], [4, 10]], [[4, 11], [5, 1]]],
        [[[7, 3], [3, 10]], [[3, 9], [5, 0]]],
    ]
    # add Bimatrix Games
    env = Environment()
    env.iterations = iterations
    for game in games:
        env.add_game(np.asarray(game))
```

```
    if save:
        n = 1
        folder = os.path.join(os.getcwd(), "data", "game_histories")
        directory = os.path.join(folder, str(n))
        path = os.path.exists(directory)
        while path:
            n += 1
            directory = os.path.join(folder, str(n))
            path = os.path.exists(directory)
        os.mkdir(directory)
        print("saving directory: {}".format(directory))
    if grid:
        if exp_decay:
            range_decay = (0.9999, 0.94)
        else:
            range_decay = (0.001, 0.01)
        t_init = (0.5, 1.8)
        t_final = (0.1, 1)
        points_t_final = 3
        points_t_init = intervals
        points_decay = 1
        a_range = (0.01, 0.65)
        best_fit, parameter_space = grid_search(
            t_init,
            points_t_init,
            a_range,
            intervals,
            range_decay,
            points_decay,
            t_final,
            points_t_final,
            env,
            runs,
            agents,
            kwargs_agent={
                "d_exp": d_exp,
                "exponential_decay": exp_decay,
                "strategy": strategy,
            },
            kwargs_averages={"begin": begin},
        )
        mse = best_fit["mse"]
        print("Mean Squared Error {}".format(mse))
        print("params {}".format(best_fit["params"]))
        histories = best_fit["history"]
        if save:
            with open(os.path.join(directory, "config.pkl"), "wb") as f:
                pickle.dump(best_fit, f, pickle.HIGHEST_PROTOCOL)
            np.save(os.path.join(directory, "params.npy"), parameter_space)
            print("history saved to {}".format(os.path.join(directory,
"params.npy")))
            open_results(directory)
        if show_fig:
            plot_average_run(histories, save_fig, directory)
            plt.show()
    elif simulate:
        for _ in range(agents):
            env.add_player(
                Agent(
                    strategy=strategy,
```

```
                    t=t,
                    alpha=a,
                    player_type=0,
                    exponential_decay=exp_decay,
                    d_exp=d_exp,
                    decay_rate=decay_rate,
                )
            )
            env.add_player(
                Agent(
                    strategy=strategy,
                    t=t,
                    alpha=a,
                    player_type=1,
                    exponential_decay=exp_decay,
                    d_exp=d_exp,
                    decay_rate=decay_rate,
                )
            )
        histories = []
        for _ in range(runs):
            history = env.play()
            histories.append(history)
        mse = mean_squared_error(histories)
        print("MSD {}".format(mse))
        if show_fig:
            plot_average_run(histories, save_fig, directory)
            plt.show()
    # save history
        if save:
            np.save(os.path.join(directory, "histories.npy"), histories)
            print("history saved to {}".format(os.path.join(directory,
"histories.npy")))
        if save_fig:
            directory = os.path.join(os.getcwd(), "figures")

    if experimental:
        plot_data()

    if save:
        return directory


if __name__ == "__main__":
    print("START")
    directory = main()
    if open_:
        open_results(open_, compare=equilibria)
        print("open")

    if plot_avg:
        directory = os.path.join("data", "game_histories", str(plot_avg))
        histories = load_obj(directory)["history"]
        if save_fig:
            save_dir = directory
        else:
            save_dir = None
        plot_avg_distance(histories, compare=equilibria, save_dir =
save_dir)

    if mse:
```

```
        import pandas as pd
        from pandas.plotting import table

        directory = os.path.join("data", "game_histories", str(mse))
        histories = load_obj(directory)["history"]
        mse_mat = np.empty((12,3))
        for i in range(12):
            if i == 11:
                j = None
            else:
                j=i+1
            mse, T, S = mean_squared_error(histories, from_game=i,
to_game=j, begin=begin)
            mse_mat[i] = [mse, T, S]
        df1 = pd.DataFrame(data = mse_mat, index=["game"+str(i+1) for i in
range(12)], columns=["MSE","T", "S"])
        print(df1.head())
        print("mse game{}: {:.4f} {:.4f} {:.4f}".format(i, mse, T, S))
        if save_fig:
            df1 = df1.round(5)
            fig, ax = plt.subplots(figsize=(5, 5))  # set size frame
            ax.xaxis.set_visible(False)  # hide the x axis
            ax.yaxis.set_visible(False)  # hide the y axis
            ax.set_frame_on(False)  # no visible frame, uncomment if size
is ok
            tabla = table(ax, df1, loc='center', colWidths=[0.17] *
len(df1.columns))  # where df is your data frame
            tabla.auto_set_font_size(False)  # Activate set fontsize
manually
            tabla.set_fontsize(12)  # if ++fontsize is necessary
++colWidths
            tabla.scale(1.2, 1.2)  # change size table
            plt.savefig(os.path.join(directory, "games_total_MSD_begin-
{}.png".format(begin)))
            print("MSD figure saved to {}".format(os.path.join(directory,
"games_total_MSD_begin-{}.png".format(begin))))
            np.save(os.path.join(directory, "games_MSD.npy"), df1.to_numpy)
        if show_fig:
            plt.show()

    if compare_games:
        import pandas as pd
        from pandas.plotting import table

        directory = os.path.join("data", "game_histories",
str(compare_games))
        histories = load_obj(directory)["history"]
        mse_c, T_c, S_c = mean_squared_error(
            histories, from_game=0, to_game=5, begin=begin
        )
        mse_n, T_n, S_n = mean_squared_error(
            histories, from_game=5, to_game=11, begin=begin
        )
        mse_t, T_t, S_t = mean_squared_error(histories, begin=begin)
        plt.figure()
        tcn = plt.bar([0,1], [S_c+T_c, S_n+T_n])
        scn = plt.bar([0,1], [S_c, S_n])
        plt.xticks([0,1], ("Constant Games", "Non-constant Games"))
        plt.legend([scn, tcn],("Sampling Variance", "Theory Specific
Component"))
        if save_fig:
```

```
            plt.savefig(os.path.join(directory, "games_const_non_bar_begin-
{}.png".format(begin)))

        df1 = pd.DataFrame({"MSD":[mse_c, mse_n, mse_t],
                            "T":[T_c, T_n, T_t],
                            "S":[S_c, S_n, S_t]})
        df1 = df1.round(5)
        df1.index = ["Constant","Nonconstant","Total"]
        fig, ax = plt.subplots(figsize=(5, 2))  # set size frame
        ax.xaxis.set_visible(False)  # hide the x axis
        ax.yaxis.set_visible(False)  # hide the y axis
        ax.set_frame_on(False)  # no visible frame, uncomment if size is ok
        tabla = table(ax, df1, loc='upper right', colWidths=[0.17] *
len(df1.columns))  # where df is your data frame
        tabla.auto_set_font_size(False)  # Activate set fontsize manually
        tabla.set_fontsize(12)  # if ++fontsize is necessary ++colWidths
        tabla.scale(1.2, 1.2)  # change size table
        if save_fig:
            plt.savefig(os.path.join(directory, "games_MSD_begin-
{}.png".format(begin)))
            print("MSD figure saved to {}".format(os.path.join(directory,
"games_MSD_begin-{}.png".format(begin))))

    if show_fig:
        plt.show()
    print("END")
```

7.3 Agent Class

```
import numpy as np


class Agent:
    def __init__(
        self,
        strategy=0,
        epsilon=0.9,
        decay_rate=0.99,
        alpha=0.3,
        t=0.5,
        final_t=0.1,
        player_type=0,
        d_exp=False,
        exponential_decay=False,
    ):
        self.reward_history = []
        self.action_history = []
        self.probability_history = []
        self.R = 0
        self.N = 0
        self.epsilon = (
            epsilon if 0 < epsilon and epsilon < 1 else 1
        )  # exploration parameter
        self.alpha = (
            alpha if 0 < alpha and alpha < 1 else 0.6
        )  # learning rate parameter
        self.original_alpha = alpha
        self.t = t  # boltzmann exploration temperature parameter
        self.original_t = t
```

```
        self.lastAction = 0
        self.decay_rate = decay_rate
        self.strategy = strategy
        self.player_type = player_type
        self.Qhistory = []
        # self.Qvalue = np.random.uniform(0,1,2)
        self.Qvalue = np.zeros(2)
        self.probabilities = self.calcProbabilities()
        self.d_exp = d_exp
        self.exponential_decay = exponential_decay
        self.final_t = final_t

    def reset(self):
        # self.Qvalue = np.zeros(2)
        self.Qvalue = np.zeros(2)
        # self.Qvalue = np.random.uniform(0,1,2)
        self.probabilities = self.calcProbabilities()
        self.epsilon = 1
        self.alpha = self.original_alpha
        self.t = self.original_t
        self.N = 0
        self.rewardHistory = []
        self.actionHistory = []
        self.probability_history = []
        self.Qhistory = []

    def decreaseEpsylon(self):
        if self.d_exp > 0.05:
            if self.epsilon * self.decay_rate - 0.1 > 0.1:
                self.epsilon = self.epsilon * self.decay_rate + 0.1

    def decreaseT(self):
        if self.exponential_decay:
            if self.t * self.decay_rate > self.final_t:
                self.t = self.t * self.decay_rate
        else:
            if self.t - self.decay_rate > self.final_t:
                self.t = self.t - self.decay_rate

    """
    def calcProbabilities(self):
        return (np.exp(self.Qvalue/ self.t) / np.sum(np.exp(self.Qvalue/
self.t)))
    """

    def calcProbabilities(self):
        # Numerically stable softmax
        # https://stackoverflow.com/a/39558290
        z = self.Qvalue.reshape(1, 2) / self.t
        assert len(z.shape) == 2
        s = np.max(z, axis=1)
        s = s[:, np.newaxis]
        e_x = np.exp(z - s)
        div = np.sum(e_x, axis=1)
        div = div[:, np.newaxis]
        div = e_x / div
        return div.flatten()

    def randomExploration(self):
        if np.random.ranf() < self.epsilon:
            action = np.random.choice(2)
```

```
            if self.d_exp:
                self.decreaseEpsylon()
        else:
            action = np.argmax(self.Qvalue)
        self.lastAction = action
        return self.lastAction

    def boltzmanExploration(self):
        if np.random.ranf() < self.epsilon:
            action = np.random.choice(a=[0, 1], p=self.calcProbabilities())
            if self.d_exp:
                self.decreaseEpsylon()
        else:
            action = np.argmax(self.Qvalue)
        self.lastAction = action
        return self.lastAction

    def noExploration(self):
        self.lastAction = np.random.choice(a=[0, 1],
p=self.calcProbabilities())
        if self.d_exp:
            self.decreaseT()
        return self.lastAction

    def take_action(self):
        self.probabilities = self.calcProbabilities()
        self.probability_history.append(self.probabilities[0])
        self.Qhistory.append(self.Qvalue)
        if self.strategy == 0:
            return self.boltzmanExploration()
        elif self.strategy == 1:
            return self.randomExploration()
        elif self.strategy == 2:
            return self.noExploration()

    def incremental_avg_update(self, R, a):
        return self.alpha * (R - self.Qvalue[a])

    def updateQ(self, R, a):
        self.N += 1
        self.Qvalue[a] += self.incremental_avg_update(R, a)

    def get_reward_for_action(self, R):
        self.updateQ(R, self.lastAction)
        self.reward_history.append(R)
        self.action_history.append(self.lastAction)
```

7.4 Environment Class

```
import numpy as np


class Environment:
    def __init__(self, games=[], agents=[], players1=[], players2=[],
iterations=200):
        self.players1 = players1
        self.players2 = players2
        self.agents = agents
        self.games = games
        self.agentsRewards = []
        self.agentsActions = []
        self.agentsProbabilities = []
        self.iterations = iterations

    def reset_agents(self):
        self.players1 = []
        self.players2 = []

    def reset_games(self):
        self.games = []

    def add_game(self, game):
        self.games.append(game)

    def add_player(self, player):
        if player.player_type == 0:
            self.players1.append(player)
        elif player.player_type == 1:
            self.players2.append(player)

    def reward(self, game, player, action_player1, action_player2):
        r = game[player, action_player1, action_player2]
        return r

    def random_matching(self):
        if len(self.players1) == len(self.players2):
            players2idx = [i for i in range(len(self.players2))]
            np.random.shuffle(players2idx)
            playersList = []
            for i in range(len(self.players1)):
                playersList.append((self.players1[i],
self.players2[players2idx[i]]))
            return playersList

    def play2_agents(self, game, player1, player2):
        action1 = player1.take_action()
        action2 = player2.take_action()
        R1 = self.reward(game, 0, action1, action2)
        R2 = self.reward(game, 1, action1, action2)
        player1.get_reward_for_action(R1)
        player2.get_reward_for_action(R2)

    def step(self, game):
        random_matched_pair_list = self.random_matching()
        for player1, player2 in random_matched_pair_list:
            self.play2_agents(game, player1, player2)
```

```
    def play(self):
        # np.random.seed(1)
        games_history = []
        if self.games and self.players1 and self.players2:
            for game in self.games:
                agents_rewards = []
                agents_actions = []
                agents_probabilities = []
                agentsQ = []
                for _ in range(self.iterations):
                    self.step(game)
                for i, j in zip(self.players1, self.players2):
                    i.final_t = i.t
                    j.final_t = j.t
                    agents_rewards.append((i.reward_history,
j.reward_history))
                    agents_actions.append((i.action_history,
j.action_history))
                    agents_probabilities.append(
                        (i.probability_history, j.probability_history)
                    )
                    agentsQ.append((i.Qhistory, j.Qhistory))
                    i.reset()
                    j.reset()
                games_history.append(
                    (agents_actions, agents_rewards, agents_probabilities,
agentsQ)
                )
        return games_history
```

7.5 Analyzer Class

```
import numpy as np
import matplotlib.pyplot as plt
import os
from data.load_csv import return_data
import tqdm

data_path = os.path.join(os.getcwd(), "data", "numpy_data.npy")


def averages(
    histories, what, begin=0, end=None, average_runs=True, from_game=0,
to_game=None
):
    """
      histories: list of games history from Environment.play()
      what: 0|1|2|3 = actions|rewards|probabilities|Qvalues
      begin: begin iteration index
      end: end iteration index
      average_runs: True|False, returns average of runs
   """
    average_runs1, average_runs2 = [], []
    for run in histories:
        gamePlayersAvg1, gamePlayersAvg2 = [], []
        for game in run[from_game:to_game]:
            avgPlayers1, avgPlayers2 = [], []
            for player in game[what]:
                avgPlayers1.append(player[0][begin:end])
                avgPlayers2.append(player[1][begin:end])
```

```
            gamePlayersAvg1.append(np.mean(avgPlayers1, axis=0))
            gamePlayersAvg2.append(np.mean(avgPlayers2, axis=0))
        average_runs1.append(gamePlayersAvg1)
        average_runs2.append(gamePlayersAvg2)
    variance_runs1 = np.std(average_runs1, axis=0)
    variance_runs2 = np.std(average_runs2, axis=0)
    if average_runs:
        average_runs1 = np.mean(average_runs1, axis=0)
        average_runs2 = np.mean(average_runs2, axis=0)
    return average_runs1, average_runs2, variance_runs1, variance_runs2


def expected_trajectory(histories, **kwargs):
    avg = averages(histories, what=2, **kwargs)
    return np.mean(avg, axis=1)


def mean_squared_error(histories, from_game=0, to_game=None, **kwargs):
    data_games = return_data()[from_game:to_game]
    avg_data_games = np.mean(data_games, axis=2)[np.newaxis, :]
    avg1, avg2, _, _ = averages(
        histories,
        from_game=from_game,
        to_game=to_game,
        what=2,
        average_runs=False,
        **kwargs
    )
    runs = len(avg1)
    avg1 = np.mean(avg1, axis=2)
    avg2 = np.mean(avg2, axis=2)
    avg_run1, avg_run2, _, _ = averages(
        histories,
        from_game=from_game,
        to_game=to_game,
        what=2,
        average_runs=True,
        **kwargs
    )
    avg_run1 = np.mean(avg_run1, axis=1)[np.newaxis, :]
    avg_run2 = np.mean(avg_run2, axis=1)[np.newaxis, :]

    var_theory_spec1 = np.square(avg_run1 - avg_data_games[:, :,
0]).flatten()
    var_theory_spec2 = np.square(avg_run2 - avg_data_games[:, :,
1]).flatten()

    sampling_variance1 = np.square(avg1 - avg_run1).flatten()
    sampling_variance2 = np.square(avg2 - avg_run2).flatten()

    T = np.sum((var_theory_spec1, var_theory_spec2))
    S = np.sum((sampling_variance1, sampling_variance2)) / runs
    mse = T + S
    return mse, T, S


def grid_search(
    range_t,
    points_t,
    range_a,
    points_a,
```

```
    range_decrease,
    points_decrease,
    range_final,
    points_final,
    environment,
    runs,
    agents,
    kwargs_agent={},
    kwargs_averages={},
):
    from src.agent import Agent
    from time import time

    best_fit = {
        "mse": 100000,
        "T": 100000,
        "S": 0,
        "history": None,
        "params": {},
        "agents": agents,
        "range_t": range_t,
        "range_a": range_a,
        "points_a": points_a,
        "points_t": points_t,
        "points_decrease": points_decrease,
        "range_decrease": range_decrease,
        "range_final": range_final,
        "points_final": points_final,
        "runs": runs,
        "kwargs_agent": kwargs_agent,
        "kwargs_averages": kwargs_averages,
    }
    interval_a = np.linspace(range_a[0], range_a[1], points_a)
    interval_t = np.linspace(range_t[0], range_t[1], points_t)
    interval_e = np.linspace(range_decrease[0], range_decrease[1],
points_decrease)
    interval_final_t = np.linspace(range_final[0], range_final[1],
points_final)
    parameter_space = np.empty((points_t, points_a, points_decrease,
points_final))
    begin_time = time()
    progress = tqdm.tqdm(
        total=points_a * points_t * points_decrease * points_final * runs
    )
    for i, t in enumerate(interval_t):  # initial t
        for j, a in enumerate(interval_a):  # alpha
            for z, e in enumerate(interval_e):  # exploration decay
                for k, final_t in enumerate(interval_final_t):  # final t
                    for _ in range(agents):
                        environment.add_player(
                            Agent(
                                player_type=0,
                                decay_rate=e,
                                alpha=a,
                                t=t,
                                final_t=final_t,
                                **kwargs_agent
                            )
                        )
                        environment.add_player(
                            Agent(
```

```
                                    player_type=1,
                                    decay_rate=e,
                                    alpha=a,
                                    t=t,
                                    final_t=final_t,
                                    **kwargs_agent
                                )
                            )
                        histories = []
                        for _ in range(runs):
                            history = environment.play()
                            histories.append(history)
                            progress.update(n=1)
                        mse, T, S = mean_squared_error(histories,
**kwargs_averages)
                        parameter_space[i, j, z, k] = mse
                        if best_fit["T"] > T:
                            best_fit["mse"] = mse
                            best_fit["T"] = T
                            best_fit["S"] = S
                            best_fit["history"] = histories
                            best_fit["params"] = {
                                "t": t,
                                "a": a,
                                "e": e,
                                "final_t": final_t,
                                "ijzk": [i, j, z, k],
                            }
                        environment.reset_agents()

    progress.close()
    end_time = time()
    delta = end_time - begin_time
    print("\nTime elapsed {:.2f} s".format(delta))
    return best_fit, parameter_space


def plot_data():
    data = return_data()
    for i in range(12):
        p = plt.subplot(12, 1, i + 1)
        plt.plot(np.arange(200), data[i, 0, :], np.arange(200), data[i, 1,
:])
        plt.xlim(0, 200)
        plt.ylim(0, 1)
        if i != 11:
            p.set_xticks([])
        plt.ylabel("Game {}".format(i + 1))
        if i == 0:
            plt.title("Experimental Probabilities")
            plt.legend(
                bbox_to_anchor=(0.9, 2),
                loc="upper left",
                borderaxespad=0.0,
                labels=["Player1", "Player2"],
            )


def show_params_space(
    directory,
    range_x,
```

```
    range_y,
    labels=["\u03C4", "\u03B1"],
    slice_=[0, slice(0, None), 0, slice(0, None)],
):
    img = np.load(os.path.join(directory, "params.npy"))[slice_]
    x_ticks = np.linspace(range_x[0], range_x[1], img.shape[0])
    x_ticks = ["%.3f" % x for x in x_ticks]
    y_ticks = np.linspace(range_y[0], range_y[1], img.shape[1])
    y_ticks = ["%.3f" % y for y in y_ticks]
    plt.figure(figsize=(10, 10))
    plt.imshow(img[:, ::-1].T)
    plt.colorbar()
    plt.title("Heat map of MSD")
    plt.xlabel(labels[0], fontsize=15)
    plt.xticks(np.arange(img.shape[0]), x_ticks)
    plt.ylabel(labels[1], fontsize=15)
    plt.yticks(np.arange(img.shape[1])[::-1], y_ticks)
    plt.xlim()
    path = os.path.join(
        directory, "params_space" + str(labels[0]).replace(" ","") +
str(labels[1]).replace(" ","") + ".png"
    )
    plt.savefig(path)
    print("image saved to {}".format(path))


def plot_avg_distance(histories, save_dir=None, compare=True, **kwargs):
    avg1, avg2, _, _ = averages(histories, what=2, **kwargs)
    avg1 = np.mean(avg1, axis=1)
    avg2 = np.mean(avg2, axis=1)
    data_games = return_data()
    data_games = np.mean(data_games, axis=2)
    fg, ax_ls = plt.subplots(4, 3, figsize=(7 * 1.5, 9 * 1.5))
    NE = np.asarray(
        [
            [0.091, 0.909],
            [0.181, 0.727],
            [0.273, 0.909],
            [0.364, 0.818],
            [0.364, 0.727],
            [0.455, 0.636],
        ]
    )
    NE = np.vstack((NE, NE))
    QE = np.asarray(
        [
            [0.07, 0.882],
            [0.172, 0.711],
            [0.25, 0.898],
            [0.348, 0.812],
            [0.354, 0.721],
            [0.449, 0.634],
        ]
    )
    QE = np.vstack((QE, QE))
    AS = np.asarray(
        [
            [0.057, 0.664],
            [0.185, 0.619],
            [0.137, 0.753],
            [0.283, 0.679],
```

```
            [0.286, 0.679],
            [0.448, 0.613],
        ]
    )
    AS = np.vstack((AS, AS))
    IB = np.asarray(
        [
            [0.104, 0.634],
            [0.258, 0.561],
            [0.188, 0.764],
            [0.304, 0.724],
            [0.354, 0.646],
            [0.466, 0.604],
        ]
    )
    IB = np.vstack((IB, IB))

    for (
        i,
        ((ea1, ea2), (ne1, ne2), (qe1, qe2), (as1, as2), (ib1, ib2), a1,
a2, ax),
    ) in enumerate(zip(data_games, NE, QE, AS, IB, avg1, avg2,
ax_ls.flat)):
        ax.set_aspect("equal", "box")
        ax.set_title("Game {}".format(i + 1), fontsize=12)
        ax.set_xlim(0, 1)
        ax.set_ylim(0, 1)
        ax.scatter(a1, a2, color="blue", label="SE")
        ax.scatter(ea1, ea2, color="orange", label="EE")
        if compare:
            ax.scatter(ne1, ne2, color="green", s=14.5)
            ax.scatter(qe1, qe2, color="red", s=14.5)
            ax.scatter(as1, as2, color="black", s=14.5)
            ax.scatter(ib1, ib2, color="purple", s=14.5)
        ax.set_xlabel("p1", fontsize=10)
        ax.set_ylabel("p2", fontsize=10)

    ax_ls[0, 1].legend(
        loc="lower left",
        bbox_to_anchor=(0, 1.2),
        borderaxespad=0.0,
        labels=[
            "Simulated",
            "Experimental",
            "Nash",
            "Quantal Response",
            "Action Sampling",
            "Impulse Balance",
        ],
        fontsize=11,
    )
    fg.tight_layout()
    if save_dir:
        if compare:
            name = "avg_dist_compare.png"
        else:
            name = "avg_dist.png"
        plt.savefig(os.path.join(save_dir, name))
        print("figure saved in {}".format(os.path.join(save_dir, name)))
```

```
def plot_average_run(histories, save_fig, directory):
    avgP1, avgP2, varP1, varP2 = averages(histories=histories, what=2)
    # print(avgP1[0].shape)
    # avgR1, avgR2, varR1, varR2 = averages(histories = histories, what =
1, average_runs=True)

    fg, ax_ls = plt.subplots(len(histories[0]), 1, figsize=(15, 12))
    for index, axis in enumerate(ax_ls):
        if index < 11:
            axis.set_xticklabels([])

    ax_ls[0].set_title("Average Players Rewards")
    ax_ls[0].set_title("Average Players Probabilities")
    ax_ls[0].set_xlabel("iterations")
    ax_ls[0].set_xlabel("iterations")

    """
   #plot rewards
   for index, (game_p1, game_p2) in enumerate(zip(avgR1, avgR2)):
      if sum_rewards:
         ax_ls[index,0].plot(np.arange(len(game_p1)), np.cumsum(game_p1) ,
np.arange(len(game_p2)), np.cumsum(game_p2))
      else:
         ax_ls[index,0].plot(np.arange(len(game_p1)), game_p1,
np.arange(len(game_p2)), game_p2)
      ax_ls[index, 0].set_xlim(0,200)
      ax_ls[index,0].set_ylabel("Game{}".format(index+1))
   """
    # plot probabilities
    for index, (game_p1, game_p2, var1, var2) in enumerate(
        zip(avgP1, avgP2, varP1, varP2)
    ):
        ax_ls[index].set_xlim(0, 200)
        ax_ls[index].set_ylim(bottom=-0.1, top=1.1)
        ax_ls[index].plot(
            np.arange(len(game_p1)), game_p1, np.arange(len(game_p2)),
game_p2
        )
        ax_ls[index].fill_between(
            np.arange(len(game_p1)), game_p1 - var1, game_p1 + var1,
alpha=0.3
        )
        ax_ls[index].fill_between(
            np.arange(len(game_p1)), game_p2 - var2, game_p2 + var2,
alpha=0.3
        )
        ax_ls[index].set_ylabel("Game{}".format(index + 1))
    ax_ls[0].legend(
        bbox_to_anchor=(0.85, 1.5),
        loc="lower left",
        borderaxespad=0.0,
        labels=["Player1", "Player2"],
    )

    if save_fig and directory:
        plt.savefig(os.path.join(directory, "RP.png"))
    else:
        n = 0
        folder = os.path.join(os.getcwd(), "figures")
        path = os.path.exists(
            os.path.join(
```

```
                folder,
                "RP_run"
                + str(len(histories))
                + "_ag"
                + str(len(histories[0][0][0]))
                + "_"
                + str(n)
                + ".png",
            )
        )
        while path:
            n += 1
            path = os.path.exists(
                os.path.join(
                    folder,
                    "RP_run",
                    str(len(histories))
                    + "_ag"
                    + str(len(histories[0][0][0]))
                    + "_"
                    + str(n)
                    + ".png",
                )
            )
        plt.savefig(
            os.path.join(
                folder,
                "RP_run"
                + str(len(histories))
                + "_ag"
                + str(len(histories[0][0][0]))
                + "_"
                + str(n)
                + ".png",
            )
        )

    plt.figure(1)
    fg, ax_ls = plt.subplots(4, 3, figsize=(8, 11))
    # fg.suptitle("Probability Dynamics")
    fg.tight_layout()
    indexes = np.arange(12).reshape(4, 3)
    for i in range(4):
        for j in range(3):
            ax_ls[i, j].set_aspect("equal", "box")
            ax_ls[i, j].set_ylim(bottom=0, top=1)
            ax_ls[i, j].set_xlim(left=0, right=1)
            ax_ls[i, j].plot(
                avgP1[indexes[i, j]], avgP2[indexes[i, j]], "--",
linewidth=0.5
            )
            ax_ls[i, j].set_ylabel("p2")
            ax_ls[i, j].set_xlabel("p1")
            ax_ls[i, j].set_title("Game{}".format(indexes[i, j] + 1))
            # ax_ls[i, j].scatter(data[0,:], data[1,:], s=0.5, color="r")
            # ax_ls[i, j].plot(data[0,:], data[1,:], linewidth=0.5,
color="r")
            """
         length = len(avgP1[indexes[i,j]])
         avp1 = np.asarray(avgP1[indexes[i,j]]).reshape(-1,1)
         avp2 = np.asarray(avgP2[indexes[i,j]]).reshape(-1,1)
```

```
         for t, (avp1, avp2) in enumerate(zip(avp1, avp2)):
            ax_ls[i,j].scatter( avp1, avp2, c = np.asarray(t).reshape(1,),
s = 10, alpha = 1-t/lenght)
         """
    if save_fig and directory:
        plt.savefig(os.path.join(directory, "Dy.png"))
    else:
        n = 0
        folder = os.path.join(os.getcwd(), "figures")
        path = os.path.exists(
            os.path.join(
                folder,
                "Dy_run"
                + str(len(histories))
                + "_ag"
                + str(len(histories[0][0][0]))
                + "_"
                + str(n)
                + ".png",
            )
        )
        while path:
            n += 1
            path = os.path.exists(
                os.path.join(
                    folder,
                    "Dy_run"
                    + str(len(histories))
                    + "_ag"
                    + str(len(histories[0][0][0]))
                    + "_"
                    + str(n)
                    + ".png",
                )
            )
        plt.savefig(
            os.path.join(
                folder,
                "Dy_run"
                + str(len(histories))
                + "_ag"
                + str(len(histories[0][0][0]))
                + "_"
                + str(n)
                + ".png",
            )
        )


def load_obj(directory):
    import pickle

    with open(os.path.join(directory, "config.pkl"), "rb") as f:
        return pickle.load(f)


def open_results(open_, **kwargs):
    print("open")
    if type(open_) == int:
        directory = os.path.join("data", "game_histories", str(open_))
    else:
```

```
        directory = open_
    obj = load_obj(directory)
    print(
        "params {} mse {:.5f} T {:.5f} S {:.5f}  agents {} runs {}".format(
            obj["params"], obj["mse"], obj["T"], obj["S"], obj["agents"],
obj["runs"]
        )
    )
    histories = obj["history"]
    plot_avg_distance(histories, directory, **kwargs, begin=100)
    i = obj["params"]["ijzk"][0]
    j = obj["params"]["ijzk"][1]
    z = obj["params"]["ijzk"][2]
    k = obj["params"]["ijzk"][3]
    t = obj["params"]["t"]
    e = obj["params"]["e"]
    a = obj["params"]["a"]
    t_f = obj["params"]["final_t"]
    final_t = obj["params"]["final_t"]
    i_len = obj["points_t"]
    j_len = obj["points_a"]
    z_len = obj["points_decrease"]
    k_len = obj["points_final"]

    slice_init_t_alpha = (slice(0, None), slice(0, None), z, k)
    slice_alpha_final_t = (i, slice(0, None), z, slice(0, None))
    slice_e_final_t = (i, j, slice(0, None), slice(0, None))
    range_e = obj["range_decrease"]
    range_final_t = obj["range_final"]
    range_initial_t = obj["range_t"]
    range_a = obj["range_a"]
    range_t = obj["range_t"]
    labels_decay_final_t = ["decay coeff", "final \u03C4"]
    labels_t_init_a = ["initial \u03C4", "\u03B1"]
    labels_a_t_final = ["\u03B1", "final \u03C4"]
    show_params_space(
        directory,
        range_x=range_a,
        range_y=range_final_t,
        slice_=slice_alpha_final_t,
        labels=labels_a_t_final,
    )
    show_params_space(
        directory,
        range_x=range_initial_t,
        range_y=range_a,
        slice_=slice_init_t_alpha,
        labels=labels_t_init_a,
    )
    show_params_space(
        directory,
        range_x=range_e,
        range_y=range_final_t,
        slice_=slice_e_final_t,
        labels=labels_decay_final_t,
    )
    return histories
```

7.6 README

to install required packages with pip:

pip install -r requirements.txt

to run the program:

python3 main.py

optional commands:

--iter, type=int, help="number of iterations :int", default=200)

--simulate, type=bool, help="Sun simulation :bool", default=False

--strategy, type=str, help="strategy of agent: "max"| "boltzmann"| "entropy" :string", default="entropy")

--agents, type=int, help="number of agents per player type :int", default=2)

--d_exp, type=bool, help="Decrease Exploration :bool", default=True)

--exp, type=bool, Whether to decrease exploration exponentially or linearly :bool, default=False)

--decay_rate, type=float, Rate of exploration decrease of exponential or linear decay :float)

--save, type=bool, save game history plots :bool", default=False)

--runs, type=int, number of simulation runs", default=1)

--save_fig, type=bool, whether to save simulations images :bool", default=False)

--show_fig, type=bool, whether to show simulations images :bool", default=False)

--sum_rewards,type=bool, whether to plot sum of rewards or rewards per iteration :bool)

--grid_search, type=bool, grid search of optimal parameter :bool, default=False)

--plot_exp, type=bool, whether to show experimental games players' probabilities :bool")

--t, type=float, temperature parameter of agents :float, default=2)

--a, type=float, alpha parameter of agents :float, default=.1)

--open, type=int, show best parameter runs in folder n of games histories :int)

--intervals, type=int, search intervals :int, default=10)

--begin, type=int, help="beginning iteration to calculate MSD, default=0)

--mse, type=int, help="Calculate mean square deviation of histories in file config.pkl in folder n :int"

--from_game, type=int, help="If mse then begin from this game", default=0

--equilibria, type=bool, help="Show all equilibrium in plot :bool", default=False

--compare, type=int, help="MSD comparison of constant and non-constant sum games of file config.pkl in folder n :int",

default=False

--plot_avg, type=int, help="Plot equilibrium results"