# An Improved Minimax-Q Algorithm Based on Generalized Policy Iteration to Solve a Chaser-Invader Game

Minsong Liu *†, Yuanheng Zhu *†, Dongbin Zhao *†
*The State Key Laboratory of Management and Control for Complex Systems, Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China
†School of Artificial Intelligence, University of Chinese Academy of Sciences, Beijing 100049, China
Email: {liuminsong2018, yuanheng.zhu, dongbin.zhao}@ia.ac.cn

***Abstract*—In this paper, we use reinforcement learning and zero-sum games to solve a Chaser-Invader game, which is actually a Markov game (MG). Different from the single agent Markov Decision Process (MDP), MG can realize the interaction of multiple agents, which is an extension of game theory to a MDP environment. This paper proposes an improved algorithm based on the classical Minimax-Q algorithm. First, in order to solve the problem where Minimax-Q algorithm can only be applied for discrete and simple environment, we use Deep Q-network instead of traditional Q-learning. Second, we propose a generalized policy iteration to solve the zero-sum game. This method makes the agent use linear programming method to solve the Nash equilibrium action at each moment. Finally, through comparative experiments, we prove that the improved algorithm can perform as well as Monte Carlo Tree Search in simple environments and better than Monte Carlo Tree Search in complex environments.**

***Index Terms*—Minimax-Q, Markov game, Generalized Policy Iteration, Deep Q-network**

## I. Introduction

With the continuous progress and development of reinforcement learning (RL) in recent years, the research on RL had gradually extended from single-agent RL to multi-agent RL. In single-agent RL learning, the goal of the agent is to learn an optimal policy through continuous interaction with the environment, so that the cumulative reward return is maximized [1]. However, the problems faced in reality are sometime multi-agent problems. In the single-agent RL, the environment of agent is stable. But in the multi-agent RL, the environment is complex and dynamic [2]. Therefore, it brings great difficulties to the learning process, and the research on multi-agent RL is extremely challenging.

Research on multi-agent RL mainly focuses on the cooperation and interaction between agents. The goal for agents is to learn how to interact with other agents while taking care of own interest. An important difference between multi-agent RL and single-agent RL is that the behavior of other agents may affect the decision of own agent. This is very similar to the idea of game theory. Therefore, combining game theory with RL can well address multi-agent cooperation and competition problems. Game theory provides a reliable mathematical framework for the study of multi-agent interactions [3]. Among them, stochastic game and Nash equilibrium are the basis for the study of multi-agent. Littman proposed the Minimax-Q algorithm [4], which combines game theory and RL to solve the two-player zero-sum game. During confrontation, the agent can learn to obtain high return and let opponent get low return. It uses the minimax method in game theory to modify the Q value update process of traditional Q-learning, so that the agent can adapt to the existence of another independent agent when making decisions. The algorithm uses linear programming to solve the Nash equilibrium policy of the stage game for each specific state [5]. It uses the TD method to iteratively learn the state value function or action-state value function [6]. Through the minimax operator, the agent can simulate the strongest opponent as much as possible, and then learn a most stable policy. Littman verified the effectiveness of the Minimax-Q algorithm in a football confrontation game. However, the Minimax-Q algorithm also has its own limitations. The Minimax-Q algorithm can find the Nash equilibrium policy of multi-agent RL. If the opponent plays a bad policy instead of Nash equilibrium policy, the current agent cannot find a better policy according to the opponent. At the same time, the Minimax-Q algorithm uses the traditional Q-learning method to update the Q-value table, so it can only be applied in a simple experimental environment with discrete states and discrete actions. Once encountering high-dimensional and continuous environments, the Minimax-Q algorithm cannot achieve good results.

In this paper, we propose an improved Minimax-Q algorithm based on generalized policy iteration to solve a Chaser-Invader game [7]. Compared with the traditional Minimax-Q algorithm, we use generalized policy iteration instead of value iteration. In the process of updating the agent's policy, the agent will consider the influences of the policy of the opponent agent, which could make the policy learned by our agent more effective. We use neural network as a value function to fit the Q value, so that the algorithm can be used in complex or continuous experimental environments.

This work was supported in part by the National Key Research and Development Program of China under Grants 2018AAA0101005 and 2018AAA0102404.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

The paper is structured as follows. In Section II, we introduce the MDP, MG, generalized policy iteration, DQN algorithm, and Minimax-Q algorithm. In Section III, we introduce an improved Minimax-Q algorithm based on generalized policy iteration. Section IV describes the Chaser-Invader game and the experiment. Finally, the conclusion and future work are summarized.

## II. Background

### A. Markov Decision Process and Generalized Policy Iteration

The MDP is a memoryless random process. A complete MDP should satisfy Markov property: a certain state information contains all relevant history. As long as the current state is known [8], all historical information is no longer needed. If a process is a Markov decision process, it includes five elements: $M = \langle S, A, P, R, \gamma \rangle$ , where $S$ is a finite set of states, $A$ is a limited set of actions, $P$ is a state transition matrix, $R$ is a reward function, and $\gamma$ is a discount factor.

Planning problems based on MDP can be solved by dynamic programming methods. The policy iteration in the dynamic programming method is the idea used to solve the optimal policy [9]. The process of policy iteration includes two steps: policy evaluation and policy improvement. Policy evaluation is a process of solving the state value function corresponding to the current policy when any policy is given. After the policy evaluation finds the value function of the current policy, it is necessary to carry out the policy improvement. The purpose of policy improvement is to find a better policy. Policy iteration is the process of repeating the policy evaluation and policy improvement. It continuously improves the policy until the policy converges to the optimal policy.

In the process of policy iteration, the two processes of policy evaluation and policy improvement alternate with each other, and one is started only after the other is completed. Generalized policy iteration (GPI) refers to the general concept of the interaction of policy evaluation and policy improvement, without relying on the granularity and other details of the two processes.

Compared with policy iteration, GPI replaces the stable value function of infinite iterations with a value function of finite iterations during the policy evaluation process, which may improves the efficiency of searching. The state value function and action-state value function for the policy evaluation of GPI are:

$$V_i^T(s_t) = \sum_{a_t} \pi^T(a_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t)[r_{a_t}^{s_{t+1}} + \gamma * V_{i-1}^T(s_{t+1})], i = 1, \cdots, N \tag{1}$$

$$Q_i^T(s_t, a_t) = \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t)[r_{a_t}^{s_{t+1}} + \gamma * V_i^T(s_{t+1})], \quad i = 1, \cdots, N \tag{2}$$

After the policy evaluation obtains the action-state value function by $N$ iterations, we need to improve the policy. The formula for policy improvement is:

$$\pi^{T+1}(s) = \text{argmax}_a \, Q_N^T(s, a) \tag{3}$$

where $T$ is the number of iterations of the GPI, $i$ is the number of iterations of value function during policy evaluation, $N$ is a finite number, $\pi^T$ represents the current policy and $\pi^{T+1}$ indicates the policy to be updated, $p(s_{t+1}|s_t, a_t)$ represents a state transition function, $V_i^T(s_t)$ represents the value function obtained by $i$ iterations using $\pi^T(a_t|s_t)$, $Q_i^T(s_t, a_t)$ represents the state-action value function obtained by $i$ iterations under the current policy $\pi^T(a_t|s_t)$.

### B. Markov Game

Matrix games are represented by tuples $\langle n, A_i, R_i \rangle$ [10], where $n$ is the number of players participating in the game, $A_i$ is the action set of the player $i$, and $R_i$ is the reward value of the player $i$. In order to maximize their reward, players choose the corresponding action from the action set according to their respective policies.

A Markov game(MG), also called stochastic game, is an extension of game theory to MDP-like environments. Markov game can be expressed as tuple $N = \langle S, A_i, P, R_i, \gamma \rangle$, where

- $S$ is a set of joint state , which indicates the state that all agents currently perform together.
- $A_i$ is the set of finite action for the agent $i$, where $A$ is a joint behavior space. $A$ can be expressed as: $A_1 \times A_2 \times \cdots \times A_n$
- $P$ is a state transition matrix. $p(s'|s, a_1, a_2, ...a_n)$ represents the probability of transition to state $s'$ after performing actions $a_1, a_2, ...a_n$ in state $s$.
- $R$ is a reward function. $r(s'|s, a_1, a_2, ...a_n)$ indicates that the agent is in the state $s$, and the immediate reward value obtained after the state transition to $s'$ by performing actions $a_1, a_2, ...a_n$.
- $\gamma$ is a discount factor used to indicate how much attention is paid to future rewards.

MG combines the characteristics of two methods of MDP and matrix game. It extends the MDP problem from a single agent to multiple agents. The behaviors of multiple agents affect each other, so that the decision-making behavior of a single agent needs to rely on the actions of other agents to achieve the interaction and collaboration of multiple agents [11]. If the total number of agents is one, the problem is a single agent problem. If the total number of agents is two, the problem is a zero-sum game. The research in this article is based on a zero-sum game.

Unlike the single agent, the state value function and state action value function of the zero-sum game are affected by the states and actions of the two agents at the same time. For a zero-sum game, the state value function can be expressed as:

$$V(s_t) = \sum_{a_t, b_t} \pi(a_t, b_t|s_t) \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t, b_t)[r_{a_t, b_t}^{s_{t+1}} + \gamma * V(s_{t+1})] \tag{4}$$

The state action value function can be expressed as:

$$Q(s_t, a_t, b_t) = \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t, b_t)\left[r_{a_t,b_t}^{s_{t+1}} + \gamma * V(s_{t+1})\right] \quad (5)$$

where $V(s_t)$ is the state value function, $Q(s_t, a_t, b_t)$ is a function of the state value of the joint action, $\pi(a_t, b_t|s_t)$ is the joint policy of two agents, which means the probability that agent A and agent B choose action $a$ and action $b$ in state $s$, $p(s_{t+1}|s_t, a_t, b_t)$ is a transition function, which represents the probability of reaching state $s_{t+1}$ with actions $a_t$ and $b_t$ in state $s_t$, $r_{a_t,b_t}^{s_{t+1}}$ is the reward function.

### C. Minimax-Q

In the Markov game, because of the other dynamic agents in the environment [12], using the single-agent value function update method cannot solve the problem well. Littman proposed the Minimax-Q algorithm combined with the game theory minimization algorithm, which only needs a state-action value function to solve the problem. The goals of two agents in the game are to maximize and minimize the state-action value function. The linear programming method is used to solve the state-action value function, and then the Nash equilibrium policy is found.

Linear programming is to find an optimal policy to maximize or minimize an objective function under given constraints. We can simplify the above game problem. First, we define a value function:

$$Q = \begin{bmatrix} Q_{11} & Q_{12} & \cdots & Q_{1j} \\ Q_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ Q_{i1} & \cdots & \cdots & Q_{ij} \end{bmatrix} \quad (6)$$

where $Q$ is a state-action value function, and $Q_{ij}$ represents the state-action value obtained when the agent chooses action $i$ and another agent chooses action $j$. Then we can write the following linear programming:

$$\begin{aligned} &\max_{p,\cdots,p_i} V \\ &s.t. \quad p_1 Q_{11} + p_2 Q_{21} + \cdots + p_i Q_{i1} \geq V \\ &\quad\; p_1 Q_{12} + p_2 Q_{22} + \cdots + p_i Q_{i2} \geq V \\ &\quad\; \vdots \\ &\quad\; p_1 Q_{1j} + p_2 Q_{2j} + \cdots + p_i Q_{ij} \geq V \\ &\quad\; p_1 + p_2 + \cdots + p_i = 1 \\ &\quad\; p_k \geq 0, k = 1, \cdots, i \end{aligned}.$$

where $p_k$ $(k = 1, \cdots, i)$ is the probability that the agent chooses action $k$. $V$ is the state value function of the agent. Solving the above formula can get the Nash equilibrium policy.

For agent A and agent B in zero-sum game, referring to Equation (4), the joint policy function is divided into two parts, where agent B's policy is to minimize Q function, and agent A's policy is to maximize Q function. For a state $s$, the state-value function of agent A under policy $\pi(s_t, \cdot)$ can be expressed as:

$$V(s_t) = \max_{\pi(s_t,\cdot)} \min_{b_t \in A_b} \sum_{a_t \in A_a} Q(s_t, a_t, b_t)\pi(s_t, a_t) \quad (7)$$

The action-state value function in zero-sum game is:

$$Q(s_t, a_t, b_t) = R(s_t, a_t, b_t) + \gamma \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t, b_t) V(s_{t+1}) \quad (8)$$

The Minimax-Q learning update formula is:

$$\begin{aligned} Q(s_t, a_t, b_t) \leftarrow & Q(s_t, a_t, b_t) + \alpha[r_{t+1} + \\ & \gamma V(s_{t+1}) - Q(s_t, a_t, b_t)] \end{aligned} \quad (9)$$

where agent B is the opponent of agent A. $a$ is the action of agent A and $b$ is the action of agent B. If $Q(s_t, a_t, b_t)$ is known, we can directly solve the Nash equilibrium policy at state $s$ by linear programming [4]. However, in multi-agent RL, $Q(s_t, a_t, b_t)$ is unknown, so the excellent TD in Q-learning is used to update the action-state value function.

### D. DQN

Traditional Q-learning is a model-free RL method. The goal is to learn the policy of choosing the optimal action in a given limited MDP. The core of the Q-learning algorithm is to use the difference between the current Q value and the target Q value to iteratively update the value. The updated formula is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \quad (10)$$

where $Q(s, a)$ is the action-state value function of state $s$ and action $a$. But traditional Q-learning cannot solve the problems of large dimensions and complex state space. DQN is a method combining deep learning and RL [13]. DQN uses multi-layer neural network to fit Q function. It calculates a loss function based on the difference between the current Q value and the target Q value fitted by the neural network [14]. We can use the loss function to improve the network weights of the deep learning model [15]. The updated formula of loss function is:

$$\mathcal{L}(\omega) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a', \omega^-) - Q(s, a, \omega))^2] \quad (11)$$

where $\omega^-$ represents the neural network parameter of the target Q value, and $\omega$ represents the neural network parameter of the current Q value. $\omega^-$ will only be updated with the value of $\omega$ after a fixed step size.

## III. Improved Algorithm

Minimax-Q is an online learning method. It can get good effect on Markov game [16]. However, Minimax-Q algorithm also has its own limitations. Minimax-Q algorithm uses value iteration to update Q function. The convergence speed of value iteration is slow. In addition, Minimax-Q algorithm cannot be applied to the problem of large state space [17]. In response to these problems, we improve the Minimax-Q algorithm and propose an improved Minimax-Q algorithm based on generalized policy iteration (M2GPI).

### A. Fitted Q Function

We use neural network to fit the Q function to replace the Q value table in the process of GPI. In order to fit the Q function, we design three fully-connected neural networks with the same structure: current $Q$ network, target $Q_1$ network, and target $Q_2$ network. Current $Q$ network is used to fit the current Q value. Target $Q_1$ network is used to fit the agent A's target Q value. Target $Q_2$ network is used to fit the agent B's target Q value.

### B. GPI

Referring to the definition of GPI for a single agent, we redefine the GPI of zero-sum games. For the update of agent A, first, we need to perform the policy evaluation. In the policy evaluation process, we need to iterate the value function $N$ times according to the current policy of agent A. During the $i$ iteration of the policy evaluation, we use the target $Q_2$ network of agent B's $i-1$ iteration and the current policy of agent A to calculate the state value function and action-state value function. From (4) and (5), we can find:

$$V(s_t) = \sum_{a_t, b_t} \pi(a_t, b_t|s_t)\, Q(s_t, a_t, b_t) \tag{12}$$

For agent A and agent B with zero-sum game, in the case of fixed agent A's policy, agent B's policy is to choose action $b$ that minimizes the value of the agent B's action-state value function. Refer to (1) and (2). The formula for the value function is:

$$V_i^T(s_t) = \sum_{a_t} \pi^T(a_t|s_t, \omega_1) \min_{b_t} Q_{i-1}^T(s_t, a_t, b_t, \omega_2), \quad i = 1, \cdots, N \tag{13}$$

The action-state value function of angen A and agent B is:

$$Q_i^T(s_t, a_t, b_t, \omega_j) = \sum_{s_{t+1}} p(s_{t+1}|s_t, a_t, b_t)[r_{a_t, b_t}^{s_{t+1}} + \gamma V_i^T(s_{t+1})], i = 1, \cdots, N, j = 1, 2 \tag{14}$$

After the policy evaluation obtains the action-state value function by $N$ iterations, we need to improve the policy. In the process of policy improvement, linear programming is adopted to solve the Nash equilibrium policy by using the action-state value function obtained after $N$ iterations of policy evaluation. The formula for policy improvement is:

$$\pi^{T+1}(s, \omega_1) = \arg\max \min Q_N^T(s, a, b, \omega_1) \tag{15}$$

After one iteration is completed, we will get a new policy. Through continuous iteration, the policy will converge to a stable value. The action-state value function will also converge to the optimal action-state value function.

### C. Update

During the update of the neural network, we use the error of the current Q value and the target Q value to establish a loss function. Gradient back propagation updates all parameters of the fitted Q-network and updates the neural network. With the continuous iteration of the optimal policy, the fitted Q-network can also stabilize. The error loss function is:

$$\mathcal{L}(\omega) = \mathbb{E}_{(s_t, a_t, b_t, r_t, s_{t+1}) \sim B}[(Q(s_t, a_t, b_t, \omega) - r_t - \gamma \sum_{s_{t+1}} \pi(a_{t+1}|s_{t+1}, \omega_1) \min_{b_{t+1}} Q(s_{t+1}, a_{t+1}, b_{t+1}, \omega_2))^2] \tag{16}$$

where $B$ is the collection of historical experience data, which is used for the experience replay, $\omega$ represents the parameter of the current $Q$ network, $\omega_1$ represents the parameter of the target $Q_1$ network, $\omega_2$ represents the parameter of the target $Q_2$ network, $\omega$ is update all the time, and $\omega_1$ is updated with $\omega$ every $C_1$ iterations, and $\omega_2$ is updated with $\omega$ every $C_2$ iterations, $C_1$, $C_2$ are constants. When $\mathcal{L}(\omega)$ is small enough, we consider that the neural network has converged to a stable state.

## IV. Experiments

### A. Chaser-Invader Game

The game is played on a grid map. As shown in the Fig.1, two players, player A and player B occupy different squares. Player A and Player B can choose one of five actions at each step: N, S, E, W, and Stand. Player A is Chaser. Player B is Invader. Goal is the target point. Chaser's goal is to catch Invader. If the Invader is located in a 3 * 3 grid centered on the Chaser after an action step, we think the Chaser catch the Invader. Invader's goal is to avoid Chaser's pursuit and reach the target point. When Invader reaches the target point or Chaser catches Invader, the game resets and the positions of Chaser and Invader are randomly generated. If Chaser catches the Invader, Chaser gets 1 point. If Invader reaches Goal, Chaser receives -10 points. If Chaser does not catch Invader and Invader does not reach Goal, the reward is calculated based on the distance between Invader and Goal and the distance between Invader and Chaser. The detailed reward rules are:

$$r = \begin{cases} 1 & \text{Chaser catches Invader} \\ -10 & \text{Invader reaches Goal} \\ D(P_A, P_B, P_{A'}, P_{B'}, P_G) & \text{others} \end{cases}. \tag{17}$$

where $D(P_A, P_B, P_{A'}, P_{B'}, P_G)$ is the function that calculates the reward in the case of Chaser does not catch Invader and Invader does not reach Goal, $P_A$, $P_B$, $P_G$ are the positions of player A ,player B, and Goal position. $P_{A'}$, $P_{B'}$ are the positions for next state. The specific form of the reward function $D$ is:

$$D = 0.1[(P_A - P_B) - (P_{A'} - P_{B'}) + (P_{B'} - P_G) - (P_B - P_G)] \tag{18}$$

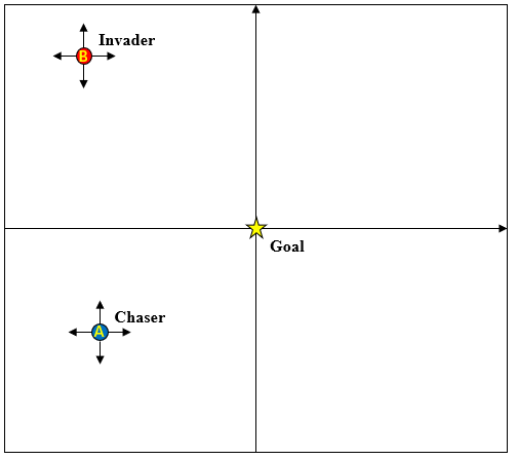


Fig. 1. The Chaser-Invader Game

### B. Experiment

We design a learning experiment for the M2GPI algorithm. The experimental parameters are set as follows. During the M2GPI iterative policy evaluation phase, the iteration number of the value function is one. In order to fit the Q function, we design three fully-connected neural networks with the same structure: current Q network, target $Q_1$ network for Chaser, and target $Q_2$ network for Invader. The number of the neural network layer is 3. The number of nodes in the hidden layer is 15. The network input is state $s$. The network output is $Q(s, a, b)$. The current $Q$ network is constantly updating parameters. Target $Q_1$ network updates network parameters every 100 steps. Target $Q_2$ network updates parameters every 1000 steps. The learning step is 250,000. The learning rate is 0.0005. The size of the experience replay is 10,000. The size of minibatch is 16. The discount factor is 0.95.

In the learning experiment, agent A and agent B both use the M2GPI algorithm. The two agents share and update the same network. When agent A makes a decision, it uses linear programming method to select the policy that maximizes the worse output of target $Q_1$ network. When agent B makes a decision, in combination with agent A's policy, it chooses the policy that minimizes the output of target $Q_2$ network.

*1) 10 * 10 Chaser-Invader Experiment*

In order to show the effect of the M2GPI algorithm and the changes in the learning process, we design three sets of comparative experiments in a 10 * 10 Chaser-Invader grid map. The first set of experiments is to use the optimal policy learned by the M2GPI algorithm to compare against Monte Carlo Tree Search(MCTS). The second set of experiments uses MCTS to compare against MCTS. The third set of experiments extracts network parameters every 10,000 steps iteration in the M2GPI learning process. We compare the policies in the learning process with MCTS. The results are in Table I. MCTS is mainly used to solve the Combinatorial Game [18] [19]. MCTS explores and uses the tree structure through the four steps of Selection, Expansion, Simulation, and Back-propagation to find a search policy that approximates Nash equilibrium [20]. M2GPI also learns policy for approaching Nash equilibrium. So comparing M2GPI and MCTS can well verify the effect of M2GPI.

*2) 200 * 200 Chaser-Invader Experiment*

In order to verify the performance of M2GPI in a large space environment, we conduct three sets of comparative experiments in a 200 * 200 Chaser-Invader grid map.

Befor the comparative experiments, we learn four different algorithms. They are M2GPI algorithm, Minimax-Q algorithm (M2Q), DQN algorithm, and DQNSelfplay algorithm. For M2GPI's policy learning, we set the M2GPI algorithm to Chaser, the Invader also uses the M2GPI algorithm. Chaser and Invader share and update the same network. For the M2Q, we set the M2Q algorithm to Chaser and Invader to be a random policy opponent. For the DQN algorithm, we set the DQN algorithm to Chaser and Invader to be random policy opponent. For the DQNSelfplay algorithm, we set the DQNSelfplay algorithm to Chaser and Invader also adopts the DQN algorithm. Chaser and Invader use and update the same network. In addition, We use four DQN networks to fight the optimal policies learned by the M2GPI, M2Q, DQN, and DQNSelfplay algorithms. The challenge DQN network is Invader, the optimal policies of M2GPI, M2Q, DQN, and DQNSelfplay algorithms are Chaser. With the four challenege DQN networks, we can learn four challenger policies of the four optimal policies.

In the first set of experiments, we compare the optimal policies trained by M2Q, DQN, and DQNSelfplay algorithms with the optimal policy trained by M2GPI. The optimal policy trained by M2GPI is Invader. The optimal policies trained by the other three algorithms are Chaser for each experiment. The results are in Table II. In the second set of experiments, we compare the optimal policies trained by the M2GPI, M2Q, DQN, and DQNSelfplay algorithms with their respective challenger policies. The optimal policies trained by the four algorithms are Chaser. The four challenger policies are Invader. The results are in Table III. In the third set of experiments, we compare the optimal policies trained by M2GPI, M2Q, and DQNSelfplay with MCTS. Among them, three tested policies are Chaser, the MCTS is Invader. The results are in Table IV.

### C. Results and Evaluation

During the game, players from both sides use their own policies to play according to preset allocations. There are 100 games in each experiment. Each step has a probability of 1% being judged as a draw. We use this method to simulate the discount factor.

*1) 10 * 10 Chaser-Invader Experiment*

In the environment with 10 * 10 Chaser-Invader grid map, the results of the comparative experiment are shown in Table I. We can see that the results of the optimal policy trained by M2GPI against MCTS are basically the same as the results of MCTS self-confrontation. M2GPI has a 50% win rate with MCTS. The experimental results of comparing policies

in the learning process with MCTS are shown in Fig. 2. Every 10,000 steps in learning step, we extract the currently learned policy and test it with MCTS. The abscissa indicates the number of tests. The ordinate indicates the M2GPI win rate. At the beginning, the M2GPI's win rate is very low. It is almost impossible to beat MCTS. However, with the continuous increase of the number of iterations, the M2GPI's win rate is more than 50% in the 100,000 steps. In the final stage of iterative learning, the winning percentage of M2GPI fluctuates around 50%. These results show that M2GPI can reach the same level as MCTS in the ability to approach Nash equilibrium.

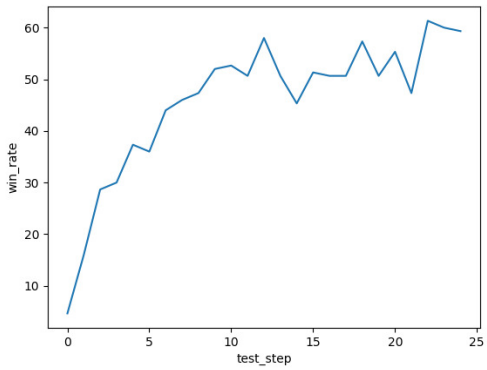


Fig. 2. M2GPI Policies in Learning Process vs MCTS in 10*10 Map

TABLE I
M2GPI vs MCTS in Small-Size

| | Chaser win | Invader win |
|---|---|---|
| M2GPI vs MCTS | 48% | 47% |
| MCTS vs MCTS | 47% | 47% |

TABLE II
M2GPI vs M2Q, DQN, DQNSelfplay in Large-Size

| | Chaser win | Invader win |
|---|---|---|
| M2Qvs M2GPI | 26% | **68**% |
| DQN vs M2GPI | 31% | **63**% |
| DQNSelfplay vs M2GPI | 31% | **64**% |

TABLE III
Challengers vs M2GPI, M2Q, DQN, DQNSelfplay in Large-Size

| | Chaser win | Invader win |
|---|---|---|
| M2GPI vs Challenger | **52**% | 40% |
| M2Q vs Challenger | 22% | 73% |
| DQN vs Challenger | 8% | 84% |
| DQNSelfplay vs Challenger | 11% | 83% |

*2) 200 * 200 Chaser-Invader Experiment*

In the environment with 200 * 200 Chaser-Invader grid map, the results of the first experiment are shown in Table II. Compared with M2GPI and M2Q, M2GPI gets the higher win rate. Because M2Q adopts the method of Q value table to update and iterate, it can not achieve good results in the face of complex experimental environments. When M2GPI compare with the optimal policies of DQN and DQNSelfplay algorithms, the optimal policy trained by M2GPI shows a higher win rate. DQN algorithm can train better results for specific policies, but it is difficult to show good results with M2GPI algorithm without training.

TABLE IV
MCTS vs M2GPI, M2Q, DQNSelfplay in Large-Size

| | Chaser win | Invader win |
|---|---|---|
| M2GPI vs MCTS | **52**% | 42% |
| MCTS vs MCTS | 41% | 41% |
| M2Q vs MCTS | 19% | 73% |
| DQNSelfplay vs MCTS | 5% | 88% |

In the second set of experiments, from Table III, we can see that except M2GPI algorithm, the challengers trained by the DQN algorithm can beat the opponent with a large score advantage. This is related to the advantages of the DQN algorithm. DQN can learn effective countermeasures based on the determined policies. The opponent simulated by M2GPI is a perfect opponent. What M2GPI can learn is the policy that can obtain the maximum profit in the face of a perfect opponent. But the DQN Challenger is not a perfect opponent. Therefore, the DQN challenger does not defeat the M2GPI algorithm.

In the third set of experiments, the optimal policies learned by the M2GPI, M2Q, and DQNSelfplay algorithms are compared with MCTS. Unlike the results in Table I, the state space is larger in the second experimental environment. The search speed and results of MCTS are affected by the larger state space. In Table IV, we can see that M2GPI performs better than MCTS. M2GPI's ability to search for optimal policy will not change much due to the changes in state space. But it is difficult for MCTS to approach Nash equilibrium policy in a large state space. Compare MCTS with M2Q, M2Q performs much worse than MCTS in large state space. DQN works well for specific policies, but it is far less than the approximates Nash equilibrium policy of MCTS.

## V. Conclusion and Future Work

In this paper, an improved Minimax-Q algorithm based on generalized policy iteration is proposed. First, we use GPI instead of value iteration, so that each decision of the current agent will make corresponding adjustments based on the decision of the other agent, which improves the effectiveness of the algorithm. Secondly, the Deep Q Network is used to fit the Q function. The method of small batches of experience replay is used to train network parameters to reduce the impact of correlation between data on the experiment, so that the M2GPI algorithm can adapt to more complex and larger dimensional experimental environments. By comparing the M2GPI learning process with MCTS in 10*10 grid map, it proves that M2GPI is constantly optimizing its own policy, and

finally achieves the same effect as MCTS. The comparisons of M2GPI with M2Q, DQN, DQNSelfplay, and MCTS in 200*200 grid map proves that M2GPI can achieve better results in complex environments. There are still shortcomings in this work, which can be further studied and improved. For example, convolutional neural networks can be used to replace fully connected neural networks to solve more complex problems in continuous environments.

## References


[1] D. Zhao, Y. Zhu, L. Lv, Y. Chen, and Q. Zhang, "Convolutional fitted Q iteration for vision-based control problems," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 4539–4544.

[2] A. Majumdar, P. Benavidez, and M. Jamshidi, "Multi-agent exploration for faster and reliable deep Q-learning convergence in reinforcement learning," in *2018 World Automation Congress (WAC)*, 2018, pp. 1–6.

[3] S. Li et al., "Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[4] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine learning proceedings*, pp. 157–163. 1994.

[5] F. A. Dahl, A. Fredrik, and O. M. Halck, "Minimax TD-learning with neural nets in a markov game," in *European Conference on Machine Learning*, 2000, pp. 117–128.

[6] K. Shao, Y. Zhu, and D. Zhao, "Starcraft micromanagement with reinforcement learning and curriculum transfer learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 3, no. 1, pp. 73–84, 2018.

[7] A. T. Bilgin and E. Kadioglu-Urtis, "An approach to multi-agent pursuit evasion games using reinforcement learning," in *2015 International Conference on Advanced Robotics (ICAR)*, 2015, pp. 164–169.

[8] A. Greenwald, K. Hall, and R. Serrano, "Correlated Q-learning," in *ICML*, 2003, vol. 3, pp. 242–249.

[9] Y. Zhu, D. Zhao, and H. He, "Invariant adaptive dynamic programming for discrete-time optimal control," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.

[10] W. Li, Y. Zhu, and D. Zhao, "Multi-agent reinforcement learning based on clustering in two-player games," in *Symposium Series on Computational Intelligence (SSCI)*, 2019.

[11] M. L. Littman, "Friend-or-foe Q-learning in general-sum games," in *ICML*, 2001, vol. 1, pp. 322–328.

[12] R. B. Diddigi, C. Kamanchi, and S. Bhatnagar, "Solution of two-player zero-sum game by successive relaxation," *arXiv preprint arXiv:1906.06659*, 2019.

[13] V. François-Lavet et al., "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018.

[14] V. Mnih et al., "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[15] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529, 2015.

[16] G. Neto and P. Lima, "Minimax value iteration applied to robotic soccer," in *Proceedings of the IEEE ICRA 2005 Workshop on Cooperative Robotics*, 2005.

[17] S. Mukhopadhyay, O. Tilak, and S. Chakrabarti, "Reinforcement learning algorithms for uncertain, dynamic, zero-sum games," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, pp. 48–54.

[18] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484, 2016.

[19] D. Silver et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[20] T. Anthony, R. Nishihara, P. Moritz, T. Salimans, and J. Schulman, "Policy gradient search: Online planning and expert iteration without search trees," *arXiv preprint arXiv:1904.03646*, 2019.