Nash Q-Learning for General-Sum Stochastic Games

Junling Hu, Michael P. Wellman

Karthik Valmeekam*

May 3, 2022

Abstract

This paper [1] extends Q-learning to multi-agent environments which are non-cooperative using the game-theoretic framework of general sum stochastic games. Q-functions here give the sum of discounted rewards for a state-action tuple. Agents maintain these Q-functions over joint actions and perform iterative Q updates where the agents assume that the other agents behave based on the Nash equilibrium over the current Q-values. The authors propose an algorithm for such an iterative update and prove the convergence of this iterative update under certain restrictive conditions. Numerical results are shown on a game that the authors had designed to showcase the convergence. Furthermore, a modern state-of-the-art domain is used to compare the proposed algorithm with multi agent deep learning algorithms.

1 Introduction

This paper generalizes single-agent Q-learning to general sum stochastic games by proposing a new algorithm incorporating game-theoretic concepts. A general sum stochastic game is a non-cooperative Nash game with discrete time, state space and action space where agents choose actions simultaneously. The term "non-cooperative" here means that an agents cannot jointly agree on the actions unless this is modeled in the game itself. Here the objective of the agents is to maximize their expected sum of rewards based on the given model. In Q-learning the objective is similar in the sense that the agents maximize the sum of discounted rewards but the rewards are the rewards they receive by acting in the environment without having a concrete model of the environment or the other agents' models that are acting in the environment.

In literature, up to the point of time where the paper was written, there was no work which had dealt with non-cooperative multi-agent scenarios with a general-sum stochastic game framework. Previous works had focused on either single agent Q learning algorithms, or under multi agent scenarios, they have considered zero sum games or cooperative games [2]. This paper, however, considers the problem of finding the equilibrium Q-values for the agents in a general-sum stochastic game. Q-learning essentially leverages the fact that the agent does not have a concrete model of the environment or, in the case of multi-agent environments, the model of other agents. The agent is expected to maximize the expected reward by repeatedly acting in the environment and observing its own state, own rewards, other agent's actions and other agent's rewards. In standard multi-agent systems, the issue of considering other agents to be rational and affecting the environment thereby making the environment non-stationary is not considered. Therefore, the framework of stochastic games is adopted to address this issue.

In this work, the authors propose an algorithm that finds the optimal Q-values for all agents. An optimal Q-value for an agent is the Q-values received in a Nash equilibrium. The goal of learning is to find these Nash-Q values through repeated play in the environment. The authors have proved the convergence of Nash-Q learning, albeit under highly restrictive conditions. The learning process converges if every stage game (which is essentially defined by Q-values) that arises in the learning process has a global optimum point and the agents update according to values at this point. The learning also converges if every stage game has a saddle point and the agents update their values based on these points. Further, they have also constructed a grid-world domain to test their algorithm.

*1st year PhD

The remainder of this report is organized as follows. Section 2 looks at the problem setup with the needed background. Section 3 describes the proposed algorithm in detail. Section 4 provides the convergence proof under restrictive conditions. Section 5 showcases the numerical results. Section 6 summarizes and concludes the report by describing possible future work.

2 Problem Setup

2.1 Markov Decision Processes

Q-learning is a reinforcement learning technique that is rooted in strong theoretical foundations pertaining to Markov Decision Processes.

Definition 1. *A* ***Markov Decision Process*** *is a tuple* $\langle S, A, r, p\rangle$*, where*

- S *is the discrete state space*
- A *is the discrete action space*
- $r : S \times A \to R$ *is the reward function of the agent*
- $p : S \times A \to \Delta(S)$ *is the transition function, where* $\Delta(S)$ *is the set of probability distributions over state space* S*.*

The objective of an MDP is to find a (Markov) strategy π so as to maximize the sum of discounted expected rewards,

$$v(s, \pi) = \Sigma_{t=0}^{\infty} \beta^t \mathbb{E}(r_t|\pi, s_0 = s) \tag{1}$$

where,

- s is a particular state, s_0 is the initial state
- r_t is the reward at time t
- $\beta \in [0, 1)$ is the discount factor
- $v(s, \pi)$ is the *value* for a state s under strategy π

The solution to an MDP is the fixed point of the following equation through an iterative search method,

$$v(s, \pi^*) = \max_a \{r(s, a) + \beta \Sigma_{s'} p(s'|s, a) v(s', \pi^*)\} \tag{2}$$

where,

- $r(s, a)$ is the reward for taking action $a \in A$ at state $s \in S$
- $s' \in S$ is the next state
- $p(s'|s, a)$ is the probability of transiting to state s' after taking action a in state s

A solution π^* that satisfies the above equation is guaranteed to be an optimal policy. Here the agent is aware of the model of the environment (rewards and state transition probabilities) which is not usually the case. Hence a problem arises when the agent does not know the model of the environment to learn the required actions. This is where Q-learning techniques provide a way to learn Q-functions by acting in the environment and updating these Q-functions based on observed state-action-state-reward tuple.

2.2 Q-learning

A Q-function is essentially the total discounted reward of taking an action in a state and then following the optimal policy thereafter.

$$\mathcal{Q}^*(s,a) = r(s,a) + \beta \Sigma_{s'} p(s'|s,a) \max_a \mathcal{Q}^*(s',a) \tag{3}$$

Now, if we know $\mathcal{Q}^*$, then we can find the optimal policy by finding the action that maximizes the Q-value under a certain state.

The technique of Q-learning provides a simple updating procedure to find the $\mathcal{Q}$-function which can then be used to find the optimal policy.

$$\mathcal{Q}_{t+1}(s_t,a_t) = (1-\alpha_t)\mathcal{Q}_t(s_t,a_t) + \alpha_t(r_t + \beta \max_a \mathcal{Q}_t(s_{t+1},a)) \tag{4}$$

where $\alpha_t \in [0,1)$ is the learning rate sequence. The sequence (4) is shown to converge to the optimal $\mathcal{Q}^*$ value provided that each state-action tuple is visited infinitely often. This proof is provided in [3].

2.3 Stochastic game

In a stochastic game, agents choose actions simultaneously. The state space and action space are assumed to be discrete.

Definition 2. *An n-player **stochastic game** Γ is a tuple $\langle S, A^1, ..., A^n, r^1, ..., r^n, p\rangle$*

- *S is the state space*
- *A^i is the action space of player $i(i = 1, ..., n)$*
- *r^i is the payoff function for player i*
- *$p : S \times A^1 \times ... \times A^n \to \Delta(S)$ is the transition probability map*
- *$\Delta(S)$ is the set of probability distributions over state space S.*

In a *discounted stochastic game*, the objective of each player is to maximize the discounted sum of rewards.

$$v^i(s, \pi^1, ..., \pi^n) = \Sigma_{t=0}^{\infty} \beta^t \mathbb{E}(r_t^1 | \pi^1, ..\pi^n, s_0 = s)$$

Definition 3. *In a stochastic game Γ, **a Nash Equilibrium point** is a tuple of n strategies $(\pi_*^1, ..., \pi_*^n)$ such that $\forall s \in S$ and $i = 1, ..., n$*

$$v^i(s, \pi_*^1, ..., \pi_*^n) \geq v^i(s, \pi_*^1, ..., \pi_*^{i-1}, \pi^i, \pi_*^{i+1}, ..., \pi_*^n) \quad \forall \pi^i \in \Pi^i$$

where Π^i is the set of strategies available to agent i

An important point here is that these Nash equilibrium strategies can be Markov or non-Markov. A Markov strategy depends only on the current state and not on the history of states that have been visited. We focus primarily on markov strategies as non-markov strategies are even more complex and less studied in the framework of general sum stochastic games. The following theorem proved in [4] shows that there always exists a Nash equilbirum in stationary strategies

Theorem 4 (c.f. [4])**.** *Every n-player discounted stochastic game possesses at least one Nash equilibrium point in stationary (markov) strategies.*

3 Extending Q-learning to multi-agent scenarios

To extend single-agent Q-learning to multi-agent scenarios using the framework of stochastic games, we need to first extend Q-functions to consider joint actions rather than just individual agent's actions. Therefore for an n-agent system, the Q-function will become $\mathcal{Q}(s, a^1, a^2, ..., a^n)$ which depends on joint actions of the agents instead of $\mathcal{Q}(s, a)$ which depends on just the individual actions of an agent.

3.1 Nash Q-values

Using the extended notion of Q-values, we can now talk about Nash equilibrium as a solution concept for the stochastic game. Recall that the future rewards in the single-agent case depends on assuming the the agent acts optimally from the next state onwards. This assumption when extended to a multi-agent scenario becomes a Nash equilibrium assumption where the agents are expected to follow specified Nash-equilbirum strategies from the next state onwards. Now we define an agent's Nash-Q agent representing the above,

Definition 5. *Agent i's Nash Q-function*

$$\mathcal{Q}^i_*(s, a^1, ..., a^n) = r^i(s, a^1, ..., a^n) + \beta\Sigma_{s'\in S} p(s'|s, a^1, ..., a^n) v^i(s', \pi^1_*, ..., \pi^n_*)$$

- $(\pi^1_*, ..., \pi^n_*)$ *is the joint Nash equilibrium strategy*
- $r^i(s, a^1, ..., a^n)$ *is agent i's one-period reward in state s under joint action*
- $v^i(s', \pi^1_*, ..., \pi^n_*)$ *is agent i's total discounted reward over infinite periods from state* s' *given all agents follow NE strategy.*

3.2 Nash-Q algorithm

We need to also distinguish between a stochastic game and a stage game. A stage game is essentially a one-period game whereas a stochastic-game is an n-period game.

Definition 6. *An* ***n-player stage game*** *is defined as* $(M^1, ..., M^n)$*, where for* $i = 1, ..., n$*,* M^i *is agent i's payoff function over the space of joint actions.*

Definition 7. *A joint strategy* $(\sigma^1, ..., \sigma^n)$ *constitutes a Nash equilibrium for the stage game* $(M^1, ..., M^n)$ *if, for* $i = 1, ..., n$

$$\sigma^i\sigma^{-i}M^i \geq \hat{\sigma}^i\sigma^{-i}M^i \quad \forall\hat{\sigma}^i \in \sigma(A^i)$$

Now for an agent to figure out the Nash equilibrium values at a stage game it also needs to know the payoffs for the other agents. In order to achieve this an agent needs to observe not only its own reward but the rewards of other agents too. Therefore, we consider games with perfect but incomplete information. The agents are able to observe the actions and obtained rewards of the other agents. Now, the Q-values can be initialized to zero and then at a time t, an agent i observes the current state s and takes an action a. After the agent takes its own action, it observes the actions and rewards of the other agents and the new state s'. Then the agent calculates the Nash Equilibrium based on the Q values of the other agents $(\mathcal{Q}^1_t(s'), ..., \mathcal{Q}^n_t(s'))$ and updates it's own Q-values through the following iterative update,

$$\mathcal{Q}^i_{t+1}(s, a^1, ..., a^n) = (1 - \alpha_t)\mathcal{Q}^i_t(s, a^1, .., a^n) + \alpha_t \left[r^i_t(s, a^1, ..., a^n) + \beta \text{Nash}\mathcal{Q}^j_i(s')\right] \tag{5}$$

where,

$$\text{Nash}\mathcal{Q}^j_i(s') = \pi^1(s') \cdots \pi^n(s') \cdot \mathcal{Q}^i_t(s') \tag{6}$$

- $\pi^1(s') \cdots \pi^n(s')$ is the Nash equilibrium for the stage game $(\mathcal{Q}^1_t(s'), ..., \mathcal{Q}^n_t(s'))$

Note that $\alpha_t = 0$ for $(s, a^1, ..., a^n) \neq (s_t, a^1_t, ..., a^n_t)$. It updates only the entry corresponding to the current state and the agents chosen by the agents. The algorithm is described in Algorithm 1.

4 Convergence Proof

In this section, we give the convergence proof for the Q-values $\mathcal{Q}^i_t$ to an equilibrium $\mathcal{Q}^i_*$ for the learning agent i. The value $\mathcal{Q}^i_*$ is dependent on the joint strategies of all agents. Our agent has to also learn the agents Q-values and derive strategies from them. Therefore the learning objective is $(\mathcal{Q}^1_*, \mathcal{Q}^2_*, ..., \mathcal{Q}^n_*)$ and we have to showcase the convergence to the same. We begin by making 3 major assumptions.

Algorithm 1 Nash Q Learning

Let $t = 0$, get the initial state s_0
Let the learning agent be indexed by i
$\forall s \in S$ and $a^j \in A^j$, $j = 1, ..., n$, let $\mathcal{Q}_t^j(s, a^1, .., a^n) = 0$
while $t \neq$ max steps **do**
  Choose action a_t^i
  Observe $r_t^1, ..., r_t^n; a_t^1, ..., a_t^n$ and $s_{t+1} = s'$
  Update $\mathcal{Q}_t^j$ for $j = 1, ..., n$
  $\mathcal{Q}_{t+1}^j(s, a^1, ..., a^n) = (1 - \alpha_t)\mathcal{Q}_t^j(s, a^1, .., a^n) + \alpha_t \left[r_t^j + \beta \text{Nash}\mathcal{Q}_t^j(s') \right]$
  where $\alpha_t \in (0, 1)$ is the learning rate, and $\text{Nash}\mathcal{Q}_t^k(s')$ is defined earlier.
  Let $t := t + 1$
end while

Assumption 8 (Infinite Sampling)**.** *Every state $s \in S$ and action $a^k \in A^k$ for $k = 1, ..., n$ are visited infinitely often.*

Assumption 9 (Learning rate Decay)**.** *The learning rate α_t satisfies the following conditions $\forall s, t, a^1, .., a^n$.*

1. $0 \leq \alpha_t(s, a^1, .., a^n) < 1$, $\Sigma_{t=0}^{\inf} \alpha_t(s, a^1, .., a^n) = \infty$, $\Sigma_{t=0}^{\inf} \left[\alpha_t(s, a^1, .., a^n) \right]^2 < \infty$ *and the latter two hold uniformly and with probability 1.*
2. $\alpha_t(s, a^1, .., a^n) = 0$ *if* $(s, a^1, .., a^n) \neq (s_t, a_t^1, .., a_t^n)$

Assumption 10 (Global or Saddle point Consistency)**.** *One of the following conditions holds during learning*

A. *Every stage game $(\mathcal{Q}_t^1(s), ..., \mathcal{Q}_t^n(s))$, $\forall t, s$, has a global optimal point and agent's payoffs in this equilibrium are used to update their $\mathcal{Q}$-functions.*

B. *Every stage game $(\mathcal{Q}_t^1(s), ..., \mathcal{Q}_t^n(s))$, $\forall t, s$, has a saddle point and agent's payoffs in this equilibrium are used to update their $\mathcal{Q}$-functions.*

Theorem 11 (Nash Q-value Convergence)**.** *Under Assumptions 1-3, the sequence $\mathcal{Q}_t = (\mathcal{Q}_t^1, ..., \mathcal{Q}_t^n)$, updated by*

$$\mathcal{Q}_{t+1}^k(s, a^1, ..., a^n) = (1 - \alpha_t)\mathcal{Q}_t^k(s, a^1, ..., a^n) + \alpha_t \left(r_t^k(s, a^1, ..., a^n) + \beta\pi^1(s')...\pi^n(s')\mathcal{Q}_t^k(s') \right)$$

for $k = 1, ..., n$ where $\pi^1(s'), ..., \pi^n(s')$ is the appropriate type of Nash equilibrium solution for the stage game $(\mathcal{Q}_t^1(s'), ..., \mathcal{Q}_t^n(s'))$, converges to the Nash $\mathcal{Q}$-value $\mathcal{Q}_* = (\mathcal{Q}_*^1, ..., \mathcal{Q}_*^n)$

Proof. We showcase the outline for the proof and the give a detailed description.

$$\underbrace{\mathcal{Q}_{t+1}^k(s, a^1, ..., a^n)}_{\mathcal{Q}_{t+1}} = (1 - \alpha_t) \underbrace{\mathcal{Q}_t^k(s, a^1, ..., a^n)}_{\mathcal{Q}_t} + \alpha_t \left(\underbrace{r_t^k(s, a^1, ..., a^n) + \beta\pi^1(s')...\pi^n(s')}_{P_t} \underbrace{\mathcal{Q}_t^k(s')}_{\mathcal{Q}_t} \right)$$

$$\mathcal{Q}_{t+1} = (1 - \alpha_t)\mathcal{Q}_t + \alpha_t(P_t\mathcal{Q}_t)$$

1. Show that P_t is a (pseudo) contraction operator which makes the above equation converge to $\mathcal{Q}_*$.
2. Show the fixed-point condition $\mathbb{E}[P_t\mathcal{Q}_*] = \mathcal{Q}_*$
3. Show that a Nash solution for the converged set of $\mathcal{Q}$-values corresponds to a Nash equilibrium point for the overall stochastic game. This is proven in Theorem 4.6.5 in [5].

□

4.1 Showing P_t is a contraction operator

Definition 12. *A joint strategy* $(\sigma^1, ..., \sigma^n)$ *of the stage game* $(M^1, ..., M^n)$ *is a global optimal point if every agent receives its highest payoff at this point. For all k,*

$$\sigma M^k \geq \hat{\sigma} M^k \quad \forall \hat{\sigma} \in \sigma(A)$$

A global optimal point is always a Nash equilibrium. It is easy to show that all global optima have equal values.

Definition 13. *A joint strategy* $(\sigma^1, ..., \sigma^n)$ *of the stage game* $(M^1, ..., M^n)$ *is a saddle point if (1) it is a Nash equilibrium, and (2) each agent would receive a higher payoff when at least one other agent deviates. For all k,*

$$\sigma^k \sigma^{-k} M^k \geq \hat{\sigma}^k \sigma^{-k} M^k \quad \forall \hat{\sigma}^k \in \sigma(A^k)$$
$$\sigma^k \sigma^{-k} M^k \geq \sigma^k \hat{\sigma}^{-k} M^k \quad \forall \hat{\sigma}^{-k} \in \sigma(A^{-k})$$

All saddle points of a stage game are equivalent in their values.

Lemma 14. *Let* $\sigma = (\sigma^1, ..., \sigma^n)$ *and* $\delta = (\delta^1, ..., \delta^n)$ *be saddle points of the n-player stage game* $(M^1, ..., M^k)$*. Then for all k,* $\sigma M^k = \delta M^k$

Proof. By definition of a saddle point, for every k = 1,..,n

$$\sigma^k \sigma^{-k} M^k \geq \delta^k \sigma^{-k} M^k \tag{7}$$
$$\delta^k \delta^{-k} M^k \leq \delta^k \sigma^{-k} M^k \tag{8}$$

Combining (7) and (8), we get

$$\sigma^k \sigma^{-k} M^k \geq \delta^k \delta^{-k} M^k \tag{9}$$
$$\delta^k \delta^{-k} M^k \geq \sigma^k \sigma^{-k} M^k \tag{10}$$

Therefore,

$$\sigma^k \sigma^{-k} M^k = \delta^k \delta^{-k} M^k$$

□

Assumption 3, as mentioned earlier, requires that all the stage games encountered during learning have global optima, or, they all have saddle points. We now define the distance between two Q-functions.

Definition 15. *For* $\mathcal{Q}$, $\hat{\mathcal{Q}}$,

$$\begin{aligned} ||\mathcal{Q} - \hat{\mathcal{Q}}|| &= \max_j \max_s ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)||_{(j,s)} \\ &= \max_j \max_s \max_{a^1,..,a^n} |\mathcal{Q}^j(s, a^1, .., a^n) - \hat{\mathcal{Q}}^j(s, a^1, ..., a^n)| \end{aligned}$$

Given Assumption 3, we can show that P_t is a contraction mapping operator.

Lemma 16. $||P_t\mathcal{Q} - P_t\hat{\mathcal{Q}}|| \leq \beta||\mathcal{Q} - \hat{\mathcal{Q}}||$ *for all* $\mathcal{Q}, \hat{\mathcal{Q}}$

Proof.

$$\begin{aligned} ||P_t\mathcal{Q} - P_t\hat{\mathcal{Q}}|| &= \max_j ||P_t\mathcal{Q}^j - P_t\hat{\mathcal{Q}}^j||_{(j)} \\ &= \max_j \max_s |\beta\pi^1(s)\dots\pi^n(s)\mathcal{Q}^j(s) - \beta\hat{\pi}^1(s)\dots\hat{\pi}^n(s)\hat{\mathcal{Q}}^j(s)| \\ &= \max_j \beta|\pi^1(s)\dots\pi^n(s)\mathcal{Q}^j(s) - \hat{\pi}^1(s)\dots\hat{\pi}^n(s)\hat{\mathcal{Q}}^j(s)| \end{aligned}$$

Now, we show that $|\pi^1(s)\dots\pi^n(s)\mathcal{Q}^j(s) - \hat{\pi}^1(s)\dots\hat{\pi}^n(s)\hat{\mathcal{Q}}^j(s)| \leq ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)||$. To simplify the notation we use σ^j to represent $\pi^j(s)$, and $\hat{\sigma}^j$ to represent $\widehat{\pi^j(s)}$. Now we want to prove,

$$|\sigma^j\sigma^{-j}\mathcal{Q}^j(s) - \hat{\sigma}^j\hat{\sigma}^{-j}\hat{\mathcal{Q}}^j(s)| \leq ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)||$$

- Case 1 (Global optimal points): If $\sigma^j \sigma^{-j} \mathcal{Q}^j(s) \geq \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s)$

$$\begin{aligned}
\sigma^j \sigma^{-j} \mathcal{Q}^j(s) - \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s) &\leq \sigma^j \sigma^{-j} \mathcal{Q}^j(s) - \sigma^j \sigma^{-j} \hat{\mathcal{Q}}^j(s) \\
&= \Sigma_{a^1,..,a^n} \sigma^1(s^1) \dots \sigma^n(a^n)(\mathcal{Q}^j(s, a^1, .., a^n) - \hat{\mathcal{Q}}^j(s, a^1, .., a^n)) \\
&\leq \Sigma_{a^1,..,a^n} \sigma^1(s^1) \dots \sigma^n(a^n) ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)|| \\
&= ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)||
\end{aligned}$$

  If $\sigma^j \sigma^{-j} \mathcal{Q}^j(s) \leq \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s)$, then a similar proof is shown.

- Case 2 (Saddle Point): If $\sigma^j \sigma^{-j} \mathcal{Q}^j(s) \geq \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s)$

$$\begin{aligned}
\sigma^j \sigma^{-j} \mathcal{Q}^j(s) \geq \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s) &\leq \sigma^j \sigma^{-j} \mathcal{Q}^j(s) \geq \sigma^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s) \\
&\leq \sigma^j \hat{\sigma^{-j}} \mathcal{Q}^j(s) \geq \sigma^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s) \\
&\leq ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)||
\end{aligned}$$

  A similar proof applies if $\sigma^j \sigma^{-j} \mathcal{Q}^j(s) \leq \hat{\sigma}^j \hat{\sigma}^{-j} \hat{\mathcal{Q}}^j(s)$

Finally,

$$\begin{aligned}
||P_t \mathcal{Q} - P_t \hat{\mathcal{Q}}|| &\leq \max_j \max_s |\beta \pi^1(s) \dots \pi^n(s) \mathcal{Q}^j(s) - \beta \hat{\pi}^1(s) \dots \hat{\pi}^n(s) \hat{\mathcal{Q}}^j(s)| \\
&\leq \max_j \max_s \beta ||\mathcal{Q}^j(s) - \hat{\mathcal{Q}}^j(s)|| \\
&= \beta ||\mathcal{Q} - \hat{\mathcal{Q}}||
\end{aligned}$$

□

4.2 Fixed point condition

Lemma 17. *For an n-player stochastic game* $\mathbb{E}[P_t, \mathcal{Q}_*] = \mathcal{Q}_*$ *where* $Q_* = (Q_*^1, ..., Q_*^n)$

Proof. We know that from [5], given optimal Q-values it corresponds to Nash equilibrium $\pi_*^1(s'), ..., \pi_*^n(s') \mathcal{Q}_*^k(s')$. Now we have,

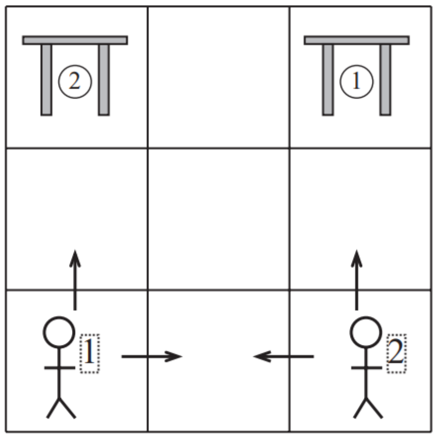
$$\begin{aligned}
\mathcal{Q}_*^i(s, a^1, ..., a^n) &= r^i(s, a^1, ..., a^n) + \beta \Sigma_{s' \in S} p(s'|s, a^1, ..., a^n) v^i(s', \pi_*^1, ..., \pi_*^n) && (11) \\
&= \Sigma_{s' \in S} p(s'|s, a^1, ..., a^n)(r^i(s, a^1, ..., a^n) + \beta \pi_*^1(s'), ..., \pi_*^n(s') \mathcal{Q}_*^k(s')) && (12) \\
&= \mathbb{E}\left[P_t^k \mathcal{Q}_*^k(s, a^1, ..., a^n)\right] && (13)
\end{aligned}$$

for all $s, a^1, ..., a^n$ and for all k. Thus $\mathcal{Q}_* = \mathbb{E}[P_t \mathcal{Q}_*]$ □

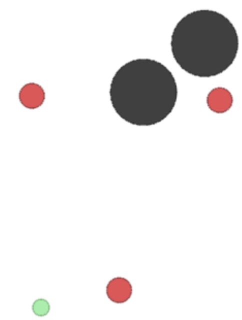
By proving Lemma 16 and Lemma 17, we show that under Assumptions 1-3, Theorem 11 is proved. Hence the convergence is proved.

5 Simulation Results

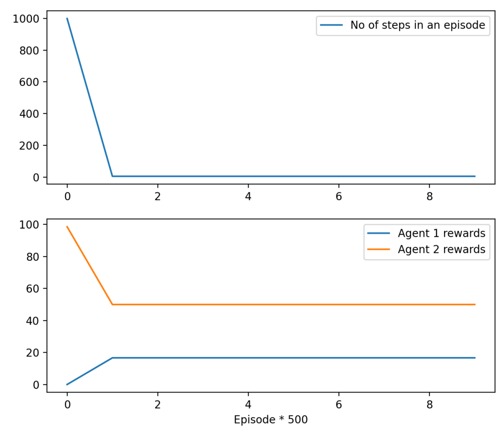
For the purpose of testing the algorithm, the authors perform grid world experiments (shown in Figure 1a). Additionally, I have chosen a modern domain where state-of-the-art multi agent deep learning agents have been tested on. This game is called simple tag game from PettingZoo [6] (shown in Figure 1b), which is a testbed framework for reinforcement learning agents.



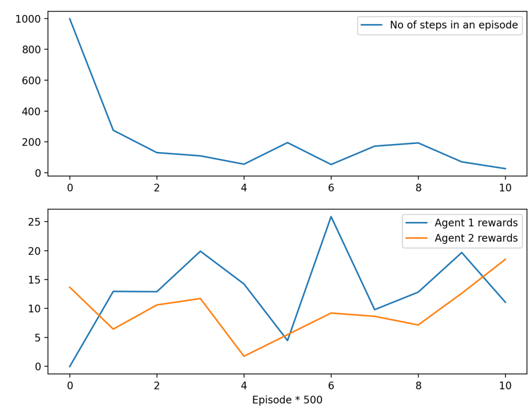
(a) Grid Game 1



(b) Multi Particle Environment - Simple Tag game (c.f. [6])



(a) Grid Game 1 with each of the agents choosing First Nash

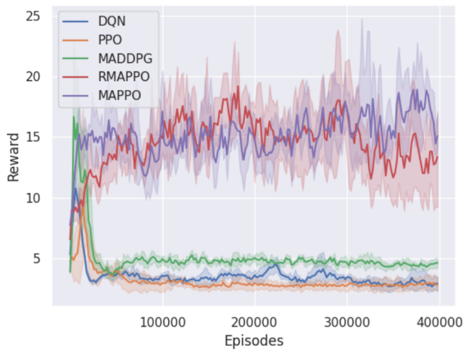


(b) Grid Game 1 with each of the agents choosing Second Nash

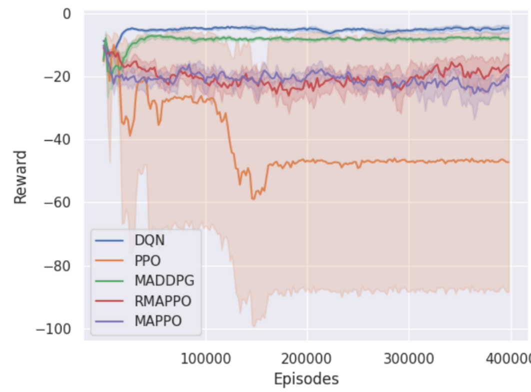
5.1 Grid World Experiments

In this game, as shown in Figure 1a, the objective of both the agents is to reach the mentioned goal with minimum number of moves. The rules of the game are as follows, (1) An agent can move only one cell at a time and in four possible directions (Left, Right, Up, Down), (2) If two agents attempt to move into the same cell (excluding either of the goal cells), they revert back to their original positions. Reaching the goal earns a reward of 100. In case of a collision, the reward is -1 and in case of just a movement without collisons or reaching the goal the rewrd is 0. The actions are deterministic and both the agents start at the position shown.

Based on the Nash equilbirum values there are multiple paths which are NE strategies. I have trained the agents for 5000 episodes with each episode having a maximum of 1000 steps. I did 5 trials of such training. Each trial took 5 hours. While choosing the Nash equilibrium, I had two types of agents: (1) an agent that chooses the First Nash and (2) an agent that chooses the second Nash. As shown in Figure 2a, when both the agents choose First Nash, it converges to the Nash Equilibrium whereas when both the agents choose Second Nash, it does not converge to a Nash equilibrium (as shown in Figure 2b). For all the 5 trials, the same result was obtained.



(a) Predator in Simple Tag game based on various Multi Agent Deep Learning algorithms (f.f.[7])



(b) Prey in Simple Tag game based on various Multi Agent Deep Learning algorithms (f.f.[7])

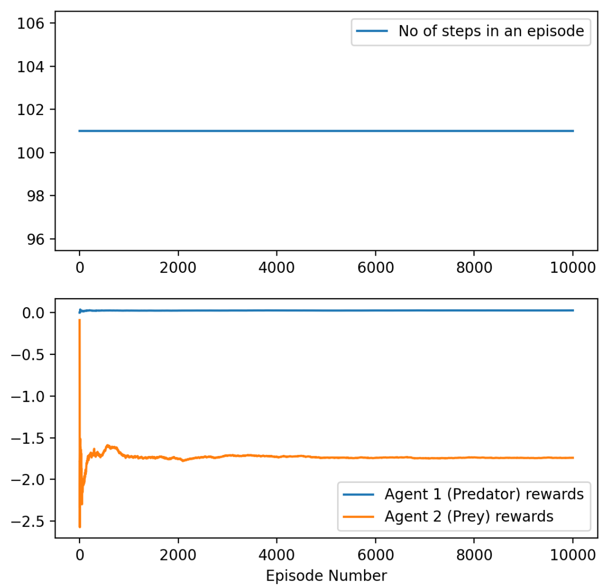


Figure 4: Predator and Prey rewards using Nash-Q Algorithm

5.2 Simple Tag Game

In this game, there are predators and preys. Both can move in four directions or choose to not move at all. Good agents (preys) are faster and receive a negative reward for being hit by adversaries (predators) (-10 for each collision). Adversaries are slower and are rewarded for hitting good agents (+10 for each collision). Obstacles (large black circles) block the way. In our game, there is 1 prey, 1 predator and 2 obstacles. I have trained for 10000 episodes with each episode having 100 cycles or 200 steps. This is similar to the experiments conducted in [7], with the difference being that they have trained the agents for 400000 episodes (as shown in Figure 3a and 3b). Compared to the deep learning agents, the Nash Q learning agent performed poorly, due to a couple major reasons. This result can be seen in Figure 4.

1. The Q-values are stored for a particular state and the state here is comprised of float values. This makes the state-action pairs unlikely to be visited again unless it is trained for huge number of episodes.

2. The exploration of the agent is not enough as every time it is exploring just enough to never get hit or hit a predator or a prey.

6 Conclusions and Future Work

The paper focuses on generalizing single-agent Q-learning to stochastic games by replacing the expected Q-value with a equilibrium operator which is based on the joint actions of all the agents. The authors provide a neat convergence proof under restrictive conditions. The authors use a gridworld domain to showcase the computational results of the algorithm. In this write-up, I have additionally tested the algorithm on a modern domain and compared the performance of the proposed algorithm to the current state-of-the-art deep learning methods. One possible future direction is to use function approximators (deep neural networks) to approximate the Q-values at each stage and use those values to find the Nash equilibrium for a stage game. This would be an interesting direction to pursue and a comparison can then be fairly made with the current deep learning approaches.

References

[1] J. Hu and M. P. Wellman, "Nash q-learning for general-sum stochastic games," *Journal of machine learning research*, vol. 4, no. Nov, pp. 1039–1069, 2003.

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

[3] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[4] A. M. Fink, "Equilibrium in a stochastic n-person game," *Journal of science of the hiroshima university, series ai (mathematics)*, vol. 28, no. 1, pp. 89–93, 1964.

[5] W. Szczechla, S. Connell, J. Filar, and O. Vrieze, "On the puiseux series expansion of the limit discount equation of stochastic games," *SIAM journal on control and optimization*, vol. 35, no. 3, pp. 860–875, 1997.

[6] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sulivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh, R. Sullivan, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," *arXiv preprint arXiv:2009.14471*, 2020.

[7] K. M. Lee, S. G. Subramanian, and M. Crowley, "Investigation of independent reinforcement learning algorithms in multi-agent environments," *arXiv preprint arXiv:2111.01100*, 2021.

7 Appendix

```python
# Implement the Nash Q Learning algorithm
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import time
import sys
import os
import math
import copy
from env import GridWorld
from nashq_agent import NashQLearner
from tqdm import tqdm

"""
TODO:
"""


def run_episode(env, episode, agents, learning=True, max_steps=1000):
    """Run single episode of the game"""
    state = env.state
```

```python
    for step in range(max_steps):
        actions = [agent.choose_action(state[agent.id], learning) for agent in agents]
        state, rewards, done, observations = env.step(tuple(actions))
        agents[0].learn(state[0], rewards[0], rewards[1], actions[1], learning)
        agents[1].learn(state[1], rewards[1], rewards[0], actions[0], learning)
        if not learning and episode !=0 and episode % 1000 == 0:
            visualize(env, 'test'+str(episode), step, state)
        if done:
            break

    average_rewards = []
    average_rewards.append(np.mean(agents[0].reward_list))
    average_rewards.append(np.mean(agents[1].reward_list))
    return step, average_rewards

def visualize(env,episode, iteration, state):
    if (os.path.isdir(str(episode)) == False):
        os.mkdir(str(episode))
    board = np.zeros(env.board.shape[:2])
    for ind, i in enumerate(state):
        index = np.unravel_index(i, board.shape)
        board[index] += ind+1
    # create a figure with size 6x6 inches, and 100 dots-per-inch and save it as a PNG file
    plt.figure(figsize=(6, 6), dpi=100)
    plt.imshow(board, cmap='hot', interpolation='nearest')
    plt.savefig(str(episode)+'/nash_q_learning_'+ str(iteration) + '.png')
    plt.close()

if __name__ == '__main__':
    nb_episodes = 5000
    max_steps = 1000
    actions = 4
    env = GridWorld(goal_pos=[(0,2),(0,0)])
    init_state = env.state
    agent1 = NashQLearner(0,init_state[0],actions)
    agent2 = NashQLearner(1,init_state[1],actions)
    action_history = []
    reward_history = {0:[],1:[]}
    #Train
    for episode in range(nb_episodes+1):
        print("Episode: {}".format(episode))
        env.reset(goal_pos=[(0,2),(0,0)])
        run_episode(env, episode, [agent1, agent2], learning=True, max_steps=max_steps)
        if episode % 500 == 0:
            env.reset(goal_pos=[(0,2),(0,0)])
            agent1.reset(env.state[0])
            agent2.reset(env.state[1])
            step, rewards = run_episode(env, episode, [agent1, agent2], learning=False,
    max_steps=max_steps)
            reward_history[0].append(rewards[0])
            reward_history[1].append(rewards[1])
            action_history.append(step)
            print("--------------------------------------------------------")
            print(f"{episode}th episode, step: {step}, a0:{rewards[0]}, a1:{rewards[1]}")
            print("--------------------------------------------------------")

    plt.figure(figsize=(12, 8))
    plt.subplot(3, 1, 1)
    plt.plot(np.arange(len(action_history)), action_history, label="step")
    plt.legend()
    plt.subplot(3, 1, 2)
    reward_history["0"] = np.array(reward_history[0])
    reward_history["1"] = np.array(reward_history[1])
    print(action_history)
    print(reward_history["0"])
    print(reward_history["1"])
    plt.plot(np.arange(len(reward_history["0"])),
             reward_history["0"], label="reward_history0")
```

```python
    plt.plot(np.arange(len(reward_history["1"])),
             reward_history["1"], label="reward_history1")
    plt.xlabel('Episode * 500')
    plt.legend()
    plt.savefig("result.png")
    plt.show()
```

Listing 1: Learning and testing loop for gridworld

```python
from pettingzoo.mpe import simple_tag_v2
# Implement the Nash Q Learning algorithm
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import time
import sys
import os
import math
import copy
from env import GridWorld
from nashq_agent import NashQLearner
from tqdm import tqdm

"""
TODO:
"""


def run_episode(env, episode, agents, agents_names, learning=True, max_steps=1000):
    """Run single episode of the game"""
    state = env.state
    prev_observations = [None, None]
    prev_rewards = [None, None]
    prev_actions = [None, None]
    step = 0
    for agent in env.agent_iter():
        observation, reward, done, infos = env.last()
        curr_agent = agents_names.index(agent)
        prev_observations[curr_agent] = observation
        prev_rewards[curr_agent] = reward
        action = agents[curr_agent].choose_action(tuple(observation), learning)
        prev_actions[curr_agent] = action
        if done:
            action = None
        env.step(action)
        # print("Curr agent",curr_agent,"\n")
        if curr_agent == 1:
            agents[0].learn(tuple(prev_observations[0]), prev_rewards[0], prev_rewards[1],
    prev_actions[1], learning)
            agents[1].learn(tuple(prev_observations[1]), prev_rewards[1], prev_rewards[0],
    prev_actions[0], learning)
        step+=1
    average_rewards = []
    average_rewards.append(np.mean(agents[0].reward_list))
    average_rewards.append(np.mean(agents[1].reward_list))
    return step//2, average_rewards

def visualize(env,episode, iteration, state):
    if (os.path.isdir(str(episode)) == False):
        os.mkdir(str(episode))
    board = np.zeros(env.board.shape[:2])
    for ind, i in enumerate(state):
        index = np.unravel_index(i, board.shape)
        board[index] += ind+1
    # create a figure with size 6x6 inches, and 100 dots-per-inch and save it as a PNG file
    plt.figure(figsize=(6, 6), dpi=100)
    plt.imshow(board, cmap='hot', interpolation='nearest')
```

```
    plt.savefig(str(episode)+'/nash_q_learning_'+ str(iteration) + '.png')
    plt.close()

if __name__ == '__main__':
    nb_episodes = 500000
    max_steps = 1000

    env = simple_tag_v2.env(num_good = 1, num_adversaries = 1, num_obstacles = 2, max_cycles
    =100)
    env.reset()
    agents_names = env.agents

    actions = 5
    init_state = tuple(env.state())
    agent1 = NashQLearner(0,init_state,actions) #Adversary
    agent2 = NashQLearner(1,init_state,actions) #Good
    action_history = []
    reward_history = {0:[],1:[]}
    #Train
    for episode in range(nb_episodes):
        env.reset()
        print("Episode: ",episode,"\n")
        step, rewards = run_episode(env, episode, [agent1, agent2], agents_names, True,
    max_steps)
        reward_history[0].append(rewards[0])
        reward_history[1].append(rewards[1])
        action_history.append(step)
        if episode % 500 == 0:
            f = open("nash_q_learning_rewards.txt", "w")
            f.write(f"Episode: {episode}\n")
            f.write(f"Action History: {action_history}\n")
            f.write(f"Adversary Reward History: {reward_history[0]}\n")
            f.write(f"Good Reward History: {reward_history[1]}\n")
            f.close()


    plt.figure(figsize=(12, 8))
    plt.subplot(3, 1, 1)
    plt.plot(np.arange(len(action_history)), action_history, label="step")
    plt.legend()
    plt.subplot(3, 1, 2)
    reward_history["0"] = np.array(reward_history[0])
    reward_history["1"] = np.array(reward_history[1])
    # print(action_history)
    # print(reward_history["0"])
    # print(reward_history["1"])
    plt.plot(np.arange(len(reward_history["0"])),
             reward_history["0"], label="reward_history0")
    plt.plot(np.arange(len(reward_history["1"])),
             reward_history["1"], label="reward_history1")
    plt.xlabel('Episode')
    plt.legend()
    plt.savefig("result_simple_tag.png")
    plt.show()
```

Listing 2: Learning and testing loop for simple tag

```
# Implement the Nash Q Learning algorithm
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import time
import sys
import os
import math
import copy

```

```python
from tqdm import tqdm

"""
TODO:
"""


class GridWorld:
    def __init__(self, goal_pos = [(0, 1),(0, 1)]):
        super().__init__()
        self.agents = [1, 2]
        self.board = np.array([[(0, 0), (0, 0), (0, 0)], [(0, 0), (0, 0), (0, 0)], [(1, 0),
    (0, 0), (0, 1)]])
        self.state = self.get_state(self.board)
        self.n_states = (self.board.shape[0] * self.board.shape[1]) ** len(self.agents)
        # print("------------Number of states: ", self.n_states, "------------")
        self.n_actions = 4
        self.actions = ["up", "down", "left", "right"]
        self.goal = goal_pos
        self.observations = {1: {'actions': [], 'rewards': []}, 2: {'actions': [], 'rewards'
    : []}}
        self.observed_states = []

    # (2,2) => 8
    def get_agent_index_ravel(self, agent_index, board):
        return np.ravel_multi_index(agent_index, board.shape[:2])
    # 8 => (2,2)
    def get_agent_index_unravel(self, agent_index, board):
        return np.unravel_index(agent_index, board.shape[:2])

    def get_state(self, board):
        # Get the state of the environment
        state = []
        for index, agent in enumerate(self.agents):
            for x, i1 in enumerate(board):
                for y, j in enumerate(i1):
                    if j[index] == 1:
                        state.append(np.ravel_multi_index((x, y), board.shape[:2]))
        return tuple(state)

    def reset(self, goal_pos = [(0, 1),(0, 1)]):
        self.agents = [1, 2]
        self.board = np.array([[(0, 0), (0, 0), (0, 0)], [(0, 0), (0, 0), (0, 0)], [(1, 0),
    (0, 0), (0, 1)]])
        self.state = self.get_state(self.board)
        self.n_states = (self.board.shape[0] * self.board.shape[1]) ** len(self.agents)
        # print("------------Number of states: ", self.n_states, "------------")
        self.n_actions = 4
        self.actions = ["up", "down", "left", "right"]
        self.goal = goal_pos
        self.observations = {1: {'actions': [], 'rewards': []}, 2: {'actions': [], 'rewards'
    : []}}
        self.observed_states = []

    def get_new_pos(self, action, agent):
        # Search for agent in the board
        # Get the position of the agent in board
        agent_pos = self.get_agent_index_unravel(self.state[agent - 1], self.board)
        opponent_pos = self.get_agent_index_unravel(self.state[agent%2], self.board)

        # print("------------Agent position: ", agent_pos, "------------")
        # print("------------Action: ", self.actions[action], "------------")
        # Update the board with action
        if action == 0:
            if agent_pos[0] - 1 >= 0:
                return tuple((agent_pos[0] - 1, agent_pos[1]))
        elif action == 1:
            if agent_pos[0] + 1 < self.board.shape[0]:
```

```python
                return tuple((agent_pos[0] + 1, agent_pos[1]))
        elif action == 2:
            if agent_pos[1] - 1 >= 0:
                return tuple((agent_pos[0], agent_pos[1] - 1))
        elif action == 3:
            if agent_pos[1] + 1 < self.board.shape[1]:
                return tuple((agent_pos[0], agent_pos[1] + 1))

        return agent_pos

    def change_board(self, agent_pos):
        # print("------------Agent position: ", agent_pos, "------------")
        temp_board = np.zeros(self.board.shape)
        for index, i in enumerate(agent_pos):
            temp_board[i][index] = 1
        self.board = temp_board

    def get_reward(self, agent_pos):
        # print("------------Agent position: ", agent_pos, "------------")
        # print("Goal: ", self.goal==agent_pos)
        reward = []
        for index,i in enumerate(agent_pos):
            if i == self.goal[index]:
                reward.append(100)
            else:
                if agent_pos[0]==agent_pos[1] and i!=self.goal[(index+1)%2]:
                    reward.append(-1)
                else:
                    reward.append(0)
        return reward

    # Action is a vector of length 2 with first element corresponding to agent 1 and second element corresponding to agent 2
    def step(self, action):
        # Based on the action, update the state of the environment
        new_agent_pos = []  # Unraveled index
        prev_agent_pos = [self.get_agent_index_unravel(i, self.board) for i in self.state]
        for index, i in enumerate(self.agents):
            new_agent_pos.append(self.get_new_pos(action[index], i))
            self.observations[self.agents[index]]['actions'].append(action[index])
        # If both the agents are in the same position, then the state is the same
        reward = self.get_reward(new_agent_pos)
        # print("1",new_agent_pos, new_agent_pos[0] == new_agent_pos[1])
        if sum(reward) == -2:
            new_agent_pos[0] = self.get_agent_index_unravel(self.state[0], self.board)
            new_agent_pos[1] = self.get_agent_index_unravel(self.state[1], self.board)
        # print("2",new_agent_pos, new_agent_pos[0] == new_agent_pos[1])
        for index, i in enumerate(self.agents):
            self.observations[i]['rewards'].append(reward[index])
        self.change_board(new_agent_pos)
        self.state = self.get_state(self.board)
        self.observed_states.append(self.state)
        if sum(reward) == 200:
            return self.state, reward, True, self.observations
        else:
            return self.state, reward, False, self.observations
```

Listing 3: GridWorld Environment

```python
import numpy as np
import random
import nashpy as nash
class NashQLearner:
    def __init__(self, id, init_state, actions, epsilon=1, alpha=0.2, gamma=0.9):
        self.id = id
        self.Q = {}
        self.opponent_Q = {}
        self.actions = actions
```

```
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.prev_state = init_state
        self.prev_action = 0
        self.reward_list = []
        self.n = {}
        self.check_new_state(init_state)

    def reset(self, init_state):
        self.prev_state = init_state
        self.prev_action = 0
        self.reward_list = []
        self.check_new_state(init_state)

    def update_epsilon(self):
        self.epsilon *= self.epsilon * 0.999
        if self.epsilon < 0.01:
            self.epsilon = 0.01

    def get_pi(self, state):
        pi, _ = self.compute_pi(state)
        return pi
    def choose_action(self, state, training=True):
        self.check_new_state(state)
        pi_nash = self.compute_pi(state)

        action = None
        if training:
            if np.random.random() < self.epsilon:
                action = random.randint(0, self.actions-1)
            else:
                # a = max(self.Q[state], key=self.Q[state].get)
                action = random.choice(np.flatnonzero(pi_nash[0] == pi_nash[0].max()))
        else:
            # a = max(self.Q[state], key=self.Q[state].get)
            action = random.choice(np.flatnonzero(pi_nash[0] == pi_nash[0].max()))

        self.prev_action = action
        return action

    def check_new_state(self, state):
        if state not in self.Q:
            self.Q[state] = {}
            self.opponent_Q[state] = {}
            for i in range(self.actions):
                for j in range(self.actions):
                    self.Q[state][(i, j)] = 0
                    self.opponent_Q[state][(i, j)] = 0
                    self.n[(state, i, j)] = 0


    def compute_pi(self, state):
        pi = []
        pi_opponent = []
        for i in range(self.actions):
            row_q = []
            row_opponent_q = []
            for j in range(self.actions):
                row_q.append(self.Q[state][(i, j)])
                row_opponent_q.append(self.opponent_Q[state][(i, j)])
            pi.append(row_q)
            pi_opponent.append(row_opponent_q)
        # Compute Nash Equilibrium using nashpy with Lemke Howson algorithm
        nash_game = nash.Game(pi, pi_opponent)
        equilibria = nash_game.lemke_howson_enumeration()
        pi_nash = None
        try:
```

```
            pi_nash_list = list(equilibria)
        except:
            pi_nash_list = []
        for ind, eq in enumerate(pi_nash_list):
            if eq[0].shape == (self.actions,) and eq[1].shape == (self.actions,):
                if any(np.isnan(eq[0]))==False and any(np.isnan(eq[1]))==False:
                    # For First Nash
                    # pi_nash = (eq[0], eq[1])
                    # break
                    #For Second Nash
                    if ind !=0:
                        pi_nash = (eq[0], eq[1])
                        break
        if pi_nash is None:
            pi_nash = (np.ones(self.actions)/self.actions, np.ones(self.actions)/self.
    actions)
        return pi_nash



    def getNashQ(self, state, pi, pi_opponent, opponent=False):
        #return max of Q[state]
        nashq = 0
        for a1 in range(self.actions):
            for a2 in range(self.actions):
                if not opponent:
                    nashq += pi[a1] * pi_opponent[a1] * self.Q[state][(a1, a2)]
                else:
                    nashq += pi[a1] * pi_opponent[a1] * self.opponent_Q[state][(a1, a2)]
        return nashq

    def getOpponentQ(self, state):
        #return max of Q[state]
        return self.opponent_Q[state][max(self.opponent_Q[state], key=self.opponent_Q[state
    ].get)]

    def update_alpha(self, state, action_opponent):
        self.alpha = 1/(self.n[(state, self.prev_action, action_opponent)])
        if self.alpha < 0.01:
            self.alpha = 0.01

    def learn(self, state, reward, reward_opponent, action_opponent,  training=True):
        self.check_new_state(state)
        pi, pi_opponent = self.compute_pi(state)
        if training:
            # print("Learning")
            # print("Q", self.Q)
            self.n[(state, self.prev_action, action_opponent)] += 1
            nashq = self.getNashQ(state, pi, pi_opponent)
            opponentq = self.getNashQ(state, pi_opponent, pi, True)
            action = tuple((self.prev_action, action_opponent))
            action_o = tuple((action_opponent, self.prev_action))
            self.update_alpha(state, action_opponent)
            self.Q[self.prev_state][action] = self.Q[self.prev_state][action] + self.alpha *
     (reward + (self.gamma * nashq) - self.Q[self.prev_state][action])
            self.opponent_Q[self.prev_state][action_o] = self.opponent_Q[self.prev_state][
    action_o] + self.alpha * (reward_opponent + (self.gamma * opponentq) - self.opponent_Q[
    self.prev_state][action_o])
            self.update_epsilon()
        self.reward_list.append(reward)
        self.prev_state = state
```

Listing 4: Nash Q Learner

```
import matplotlib.pyplot as plt
import numpy as np
import json
reward_history = {}
```

```
f = open("nash_q_learning_rewards.txt", "r")
for line in f:
    if "Episode" in line:
        episode = int(line.split(":")[1])
    elif "Action History" in line:
        a3 = json.loads(line.split(":")[1])
    elif "Adversary Reward History" in line:
        a1 = json.loads(line.split(":")[1])
    elif "Good Reward History" in line:
        a2 = json.loads(line.split(":")[1])
# First Nash
# a1 = [0,16.66668,16.66668,16.66668,16.66668,16.66668,16.66668,16.66668,16.66668,16.66668,]
# a2 = [98.6,50.0,50.0,50.0,50.0,50.0,50.0,50.0,50.0,50.0]
# a3 = [999,5,5,5,5,5,5,5,5,5]
# Second Nash
# a3 = [999, 275, 130, 109, 55, 195, 53, 172, 193, 70, 26]
# a1 = [-0.039,12.94565217, 12.89312977, 19.89090909, 14.23214286,  4.44897959,25.87037037,
     9.79190751, 12.81443299, 19.66197183, 11.07407407]
# a2 = [13.661,6.42391304, 10.60305344, 11.70909091,  1.73214286, 5.46938776,  9.2037037,
    8.63583815,  7.1443299,  12.61971831, 18.48148148]
reward_history["0"] = a1
reward_history["1"] = a2
plt.figure(figsize=(12, 8))
plt.subplot(3, 1, 1)
plt.plot(np.arange(len(a3)), a3, label="No of steps in an episode")
plt.legend()
plt.subplot(3, 1, 2)
plt.plot(np.arange(len(reward_history["0"])),
         reward_history["0"], label="Agent 1 (Predator) rewards")
plt.plot(np.arange(len(reward_history["1"])),
         reward_history["1"], label="Agent 2 (Prey) rewards")
#x axis title
plt.xlabel('Episode Number')
#x axis ticks each tick label represent 500 episodes
plt.legend()
plt.savefig("result_ag.png")
plt.show()
```

Listing 5: Extra graph creation code