

Basic Knowledge of Game Theory

I. 矩阵博弈

双智能体矩阵博弈：

行智能体 ① 列智能体 ② 奖励矩阵

$$R_1 = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

时，智能体 1 的奖励

$$R_2 = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

△ 零和博弈： $R_1 = -R_2$

-一般和博弈中： $v_{if} > v_{-if}$, $c_{if} > c_{-if}$ 时，为严格纳什均衡。（其中 v_{if} 表另一个策略）

II. 随机博弈 (Stochastic Gaming)

本质：矩阵博弈 + MDP (Markov Decision Process)

Multi-agent, one state + Multi-state, one agent \Rightarrow Multi-agent, Multi-state.

△ 这导致了随机博弈具有 Markov property $\Rightarrow S_t, R_t$ 只与 S_t, A_t 有关。

子博弈：static game：无状态转移的博弈。E.g.: 矩阵博弈

stage game：随机博弈的组成部分，为 s 固定时的随机博弈

repeat game: Agent 重复 求同 $\underbrace{\text{同一状态的随机博弈}}_{\text{同一阶段博弈}}$ 。

\Rightarrow 在重复博弈过程中该 agent 收集其它智能体的信息与奖励值。

数学表示: $(n, s, \underbrace{a_1, \dots, a_n}_{\# \text{agent}}, \underbrace{T_r, r, R_1, \dots, R_n}_{\text{agent's action space}})$

$\underbrace{\text{state space}}_{\text{# agent}}$ $\underbrace{\text{agent's action space}}_{\text{agent's action space}}$ $\underbrace{\text{状态转移}}_{\text{转移概率}} \quad \underbrace{\text{Discount factor}}_{\text{折扣率}}$ $\underbrace{\text{reward}}_{\text{奖励}}$

Classification (by reward function $Q(s, a)$):

随机+博弈 ————— 完全合作+博弈 \Rightarrow 智能体 reward function 相同
 (团队+博弈)

完全竞争+博弈 \Rightarrow 智能体 reward function
 (零和+博弈) 逆号。 (只有1个纳什均衡点)

一般和博弈 \Rightarrow 任何类型的矩阵博弈 (可能存在多个纳什均衡点)

阶段博弈中 Reward Function 的表达式:

According to RL's Bellman equation:

$$V_i^*(s) = \sum_{\substack{[a_1, a_2, \dots, a_n] \in A_1 \times \dots \times A_n \\ \text{状态奖励函数}}} Q_i^*(s, a_1, \dots, a_n) \pi_i^*(s, a_1, \dots, a_n)$$

$$Q_i^*(s, a_1, \dots, a_n) = \sum_{\substack{s' \in S \\ \text{状态} s \text{下的 state space,}}} T_r(s, a_1, \dots, a_n, s') [R_i(s, a_1, \dots, a_n, s') + r \cdot V_i^*(s')]$$

$\underbrace{\text{动作奖励函数}}_{\text{short-term}} \quad \underbrace{\text{状态转移概率}}_{\text{short-term}} \quad \underbrace{\text{reward function}}_{\text{(短期)}} \quad \underbrace{\text{reward function}}_{\text{(长期)}}$

s' 为 s 所能达到的下一状态

纳什均衡:

Definition:

$$V_i(\pi_1^*, \dots, \pi_{i-1}^*, \dots, \pi_n^*) \geq V_i(\pi_1^*, \dots, \pi_i, \dots, \pi_n^*),$$

agent i 的奖励

$$\forall \pi_i \in \Pi_i, i = 1, \dots, n$$

意义: 为所有 agent 的联结策略, 在纳什均衡外, 任何一个 agent 均不能在仅改变自身策略 (π_i) 的情况下来获得更大的奖励 (Value)

应用 \Rightarrow 多智能体的纳什均衡 (纳什均衡定义式 + 阶段博弈中奖励函数表达式)

$$V_i(s - \pi_i^*, \dots, \pi_n^*)$$

$$\leq Q_i(a_1, \dots, a_n) \underbrace{\pi_1^*(a_1) \dots \pi_i^*(a_i) \dots \pi_n^*(a_n)}_{\substack{[a_1, \dots, a_n] \in A_1 \times \dots \times A_n \\ \text{联结动作}}} \geq$$

联结动作下的 Agent i 的奖励

$$\leq Q_i(a_1, \dots, a_n) \pi_1^*(a_1) \dots \pi_i^*(a_i) \dots \pi_n^*(a_n),$$

$$\underbrace{[a_1, \dots, a_n] \in A_1 \times \dots \times A_n}_{\substack{\forall \pi_i \in \Pi_i, i = 1, \dots, n \\ V_i(s, \pi_1^* \dots \pi_i, \dots \pi_n^*)}}$$

严格纳什均衡: 将 \geq 改为 $>$ 则为严格纳什均衡。

Something about policy:

完全混合策略 $\Rightarrow \pi(a_i) > 0, i = 1, 2, \dots, n$

一般策略 \Rightarrow 介于上下之间。

纯策略 $\Rightarrow \pi(a_k) = 1, \text{ 除此之外动作概率分布均为 } 0.$

△ Learning process of MARL 就是求解每一个状态的最优纳什均衡策略，再将这些策略联合起来形成 MARL 的最优策略(π_1^*, \dots, π_n^*)

Basic Algorithm for MARL

Evaluation Matrices:

Rationality (合理性) \Rightarrow 当对手策略恒定时，agent
能学习并收敛到一个比对手策略更优的策略

Convergence (收敛性) \Rightarrow 当其它 agents 也使用学习算法时，
Agent 能学习并收敛到一个稳定的策略。

I. Minimax-Q Algorithm

Steps of Minimax-Q Algorithm:

① $\pi_i(s) \rightarrow a_i$

② $R(s, a_i) \rightarrow r_i; S \times A \rightarrow S';$ ~~agent(-i)~~ $\rightarrow a_{-i}$

③ Update $Q(s, a_i, a_{-i})$ by TD method.

④ Use LP to solve $V_i^*(s)$, then update $\tilde{V}_i(s), \tilde{\pi}_i(s)$

Minimax-Q 算法

Used in: ③, ④

1. 初始化 $Q_i(s, a_i, a_{-i}), V_i(s), \pi_i$

For iteration do:

2. 第 i 个智能体根据当前状态 s 采用探索-利用策略得到动作 a_i 并执行。

3. 得到下一个状态 s' , 以及智能体 i 获得的奖励 r_i , 并且观测智能体 -i 在状态 s 执行的策略 a_{-i}

4. 更新 $Q_i(s, a_i, a_{-i})$: \Rightarrow 使用 Q-learning 相同的方法: 1-step TD 更新 Q
 $Q_i(s, a_i, a_{-i}) \leftarrow Q_i(s, a_i, a_{-i}) + \alpha[r_i + \gamma V_i(s') - Q_i(s, a_i, a_{-i})]$ Reason: MARL 中无法直接用 LP (Linear Programming)
来解出 state=s 时的均衡策略 $\pi_i(s)$, 只能用逼近方法获得 $Q(s, a_i, a_{-i})$ 值

5. 利用线性规划求解 $V_i^*(s) = \max_{\pi_i(s, \cdot)}$ $\min_{a_{-i} \in A_{-i}} \sum_{a_i \in A_i} Q_i^*(s, a_i, a_{-i}) \pi_i(s, a_i)$, $i=1, 2$ 并更新 $V_i(s)$ 与 $\pi_i(s, \cdot)$.
此时联结动作状态值函数与: (i) 当前状态
(ii) 动作 a_i
(iii) 对手动作 a_{-i}

End for.

知乎 @ECKai

Application:

Two-Agent Zero-Sum Stochastic Game.
Constraints: (i) (ii) (iii)

Disadvantages:

1. In step ④ it uses LP to solve, it needs a lot of time. ($O(n^3)$)

2. In step ④ it need to know all the action space A_i and A_{-i} to calculate $\max\{\min(\cdot)\}$. Thus it can't be used in distributed systems.

3. It only satisfies convergence, not rationality.

\Rightarrow Minimax-Q Algorithm is a opponent-independent algorithm. Even if the opponent use a weak policy, the agent still converge to Nash equilibrium strategy of the game. It can't learn more.

II. Nash Q-Learning Algorithm

Nash Q-Learning 算法

1. 初始化 $Q_i(s, a_1, \dots, a_n) = 0 \forall a_i \in A_i$

For iteration do:

2. 第 i 个智能体根据当前状态 s 采用探索-利用策略得到动作 a_i 并执行。

3. 得到下一个状态 s' , 以及智能体 i 观测所有智能体的奖励 r_1, \dots, r_n , 并且观测所有智能体在状态 s 执行的策略 a_1, \dots, a_n . Difference between Minimax-Q and Nash Q-learning:

4. 更新 $Q_i(s, a_1, \dots, a_n)$: (i) $V_i(s)$ 替换为 $\text{Nash}Q_i(s')$

$$Q_i(s, a_1, \dots, a_n) \leftarrow Q_i(s, a_1, \dots, a_n) + \alpha[r_i + \gamma \text{Nash}Q_i(s') - Q_i(s, a_1, \dots, a_n)]$$

5. 利用二次规划求解状态 s 处的纳什均衡策略并更新 $\text{Nash}Q_i(s)$ 与 $\pi_i(s, \cdot)$.

(ii) 替换线性规划

End for.

知乎 @ECKai

Application: Multi-agent (i) General-Sum (ii) Stochastic Game (iii)

Disadvantages: (Same as Minimax-Q Algorithm)

1. In step ④ it uses LP to solve, it needs a lot of time.

2. In step ④ it need to know all the action space A_i and A_{-i} to calculate $\max\{\min(\cdot)\}$. Thus it can't be used in distributed systems.

3. It only satisfies convergence, not rationality.

⇒ Nash Q-Learning Algorithm is a opponent-independent algorithm. Even if the opponent use a weak policy, the agent still converge to Nash equilibrium strategy of the game. It can't learn more.



Nash Q-learning 在合作性均衡 / 对抗性均衡 环境中均能收敛到纳什均衡点。



收敛性条件: 在每个阶段中, 能找到全局最优点 / 单点。

III FFQ (Friend-or-Foe Q-Learning) Algorithm:

目的：处理一般和博弃

Friend-or-Foe Q-Learning 算法

- 初始化 $V_i(s) = 0, Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2}) = 0$, (a_1, \dots, a_{n_1}) 表示 i 所有 friend 的动作, (o_1, \dots, o_{n_2}) 表示 i 所有 foe 的动作,

For iteration do:

- 第 i 个智能体根据当前状态 s 采用探索-利用策略得到动作 a_i 并执行。
- 得到下一个状态 s' , 以及智能体 i 观测自身的奖励 r_i , 并且观测所有 friend 的动作 (a_1, \dots, a_{n_1}) 与所有 foe 的动作 (o_1, \dots, o_{n_2}) 。

- 更新 $Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2}) = 0$: $\dots [i] \dots$

$$Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2}) \leftarrow Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2}) + \alpha[r_i + \gamma V_i(s') -$$

$Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2})]$ \Rightarrow 使用 Q-learning 相同的方法: 1-step TD 更新 Q

- 利用线性规划求解状态 s 处的纳什均衡策略并更新 $V_i(s)$ 与 $\pi_i(s, \cdot)$, 更新公式如下

$$V_i(s) = \max_{\pi_1(s, \cdot), \dots, \pi_{n_1}(s, \cdot)} \min_{\substack{o_1, \dots, o_{n_2} \in O_1 \times \dots \times O_{n_2} \\ a_1, \dots, a_{n_1} \in A_1 \times \dots \times A_{n_1}}} \sum_{\substack{[ii] \\ Q_i(s, a_1, \dots, a_{n_1}, o_1, \dots, o_{n_2}) \pi_1(s, a_1), \dots, \pi_{n_1}(s, a_{n_1})}}$$

与 Minmax-Q 一致

End for.

知乎 @ECKai

Main idea: 对每个 agent (i), 将其余 agents 分为 friend . foe
 帮助 i 最大化 rewards 对抗 i 降低其 rewards

将一个 n -agent General-Sum Game 转变为 Two-agent Zero-Sum Game.

Disadvantages: (Same as Minmax-Q Algorithm)

1. In step ④ it uses LP to solve, it needs a lot of time.

2. In step ④ it need to know all the action space A_i and A_{-i} to calculate $\max\{\min(\cdot)\}$ Thus it can't be used in distributed systems.

3. It only satisfies convergence, not rationality.

IV. Win-or-Lose-Fast PHC Algorithm

PHC 算法

1. 初始化 $Q(s, a) = 0, \pi(s, a) = \frac{1}{|A|}$ \Rightarrow random policy

For iteration do:

2. 智能体根据当前状态 s 采用探索-利用策略得到动作 a 并执行。

3. 观测下一个奖励值 r 以及下一个状态 s'

4. 更新 $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r_i + \gamma \max_{a'} Q(s', a') - Q(s, a)] \Rightarrow Q\text{-learning}$$

5. 根据 $Q(s, a)$, 对每一个 $a \in A$ 更新 $\pi(s, a)$ 比前均为 $\pi(s)$

$$\pi(s, a) \leftarrow \pi(s, a) + \Delta_{sa}$$

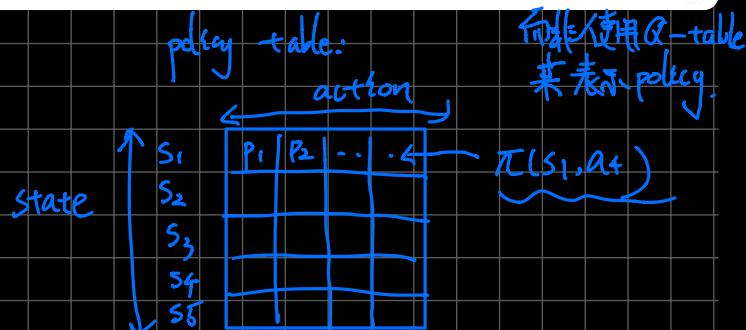
其中,

$$\Delta_{sa} = \begin{cases} -\delta_{sa} & \text{若 } a \neq \arg\max_{a'} Q(s, a') \\ \sum_{a' \neq a} \delta_{sa'} & \text{否则} \end{cases}$$

$$\delta_{sa} = \min(\pi(s, a), \frac{\delta}{|A| - 1})$$

与 Q -learning policy update 的区别: Q -learning 中为 ϵ -greedy policy improvement, 更新 $\pi(s)$, $\pi(s)$ 为 s 下取动作 a 的概率分布, 实现中即一张 Q -table。因此 PHC 的 $\pi(s, a)$ 即直接显式的表达了

End for.



WoLF-PHC 算法

1. 初始化 $Q_i(s, a_i) = 0, \pi_i(s, a_i) = \frac{1}{|A_i|}, \bar{\pi}_i(s, a_i) = \frac{1}{|A_i|}, \delta_l > \delta_w, C(s) = 0$ 为状态 s 出现的次数。

For iteration do:

2. 智能体 i 根据当前状态 s 采用探索-利用策略得到动作 a_c 并执行。

3. 观测下一个奖励值 r_i 以及下一个状态 s' 。

4. 更新 $Q_i(s, a_c)$:

$$Q_i(s, a_c) \leftarrow Q_i(s, a_c) + \alpha [r_i + \gamma \max_{a'} Q_i(s', a') - Q_i(s, a_c)]$$

5. 对每一个 $a_i \in A_i$, 更新平均估计策略 $\bar{\pi}_i(s, a_i)$:

对每个动作更新

$$C(s) = C(s) + 1$$

$$\bar{\pi}_i(s, a_i) = \bar{\pi}_i(s, a_i) + \frac{1}{C(s)} [\pi_i(s, a_i) - \bar{\pi}_i(s, a_i)]$$

} 增加计算期望策略的步骤

6. 根据 $Q_i(s, a_i)$, 对每一个 $a_i \in A_i$ 更新 $\pi_i(s, a_i)$:

$$\pi_i(s, a_i) \leftarrow \pi_i(s, a_i) + \Delta_{sa}$$

PHC

其中,

$$\Delta_{sa_i} = \begin{cases} -\delta_{sa_i} & \text{若 } a_i \neq \operatorname{argmax}_{a'} Q(s, a') \\ \sum_{a' \neq a_i} \delta_{sa'} & \text{否则} \end{cases}$$

$$\delta_{sa} = \min(\pi_i(s, a_i), \frac{\delta}{|A_i| - 1})$$

$$\delta = \begin{cases} \delta_w & \text{若 } \sum_{a_i \in A_i} \pi_i(s, a_i) Q_i(s, a_i) > \sum_{a_i \in A_i} \bar{\pi}_i(s, a_i) Q_i(s, a_i) \\ \delta_l & \text{否则} \end{cases}$$

} 增加 learning rate 调整步骤

End for.

Advantages : (i) 每个 agent 只需保存自己动作来计算 Q 函数, 将 agent 存储空间降到 $|s| \cdot |A|$

(ii) 具有合理性与收敛性 \Rightarrow 得益于 PHC

(iii) 学习速度快 \Rightarrow 得益于 PHC, 避免了使用 LP.

(iv) 使用可变 Learning rate δ , 在策略效果差时 $\delta = \delta_l$, 如时 $\delta = \delta_w$ ($\delta_l > \delta_w$) \Rightarrow Agent 在获得 rewards 低于预期时, 能快速调整以适应其它 agents 变化; rewards 低于预期时, 谨慎学习给其它 agents 适应策略变化的时间。

△ WOLF PHC Algorithm 收敛性 一直未得到证明