Lecture 18-Stochastic approximation approaches

Introduction

- We will focus on the algorithms that have been used for learning how to choose the optimal actions when agents are playing matrix games repeatedly

- We going to discuss the learning algorithms that looks similar to gradient ascent algorithm

$$w' \leftarrow w + \eta \frac{\partial L(w)}{\partial w}$$

- Centralized learning
  - Infinitesimal gradient ascent (IGA)
    - Win or Learn Fast (WoLF) + IGA = WoLF-IGA

- Decentralized learning
  - Policy Hill Climbing (PHC)
    - Win or Learn Fast (WoLF) + PHC = WoLF-PHC
  - Linear reward-inaction (L_{RI}) algorithm
  - Lagging anchor algorithm
  - L_{RI} −lagging anchor algorithm

Infinitesimal gradient ascent (IGA)

- Used in in relatively simple two-action/two-player general-sum games

	β action 1	$1-\beta$ action 2
α action 1	(r_{11}, c_{11})	(r_{12}, c_{12})
$1-\alpha$ action 2	(r_{21}, c_{21})	(r_{22}, c_{22})

  - Payoff matrix for the row player

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

  - Payoff matrix for the column player

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

- $P(a_r = 1) = \alpha, \quad P(a_r = 2) = 1 - \alpha$
- $P(a_c = 1) = \beta, \quad P(a_c = 2) = 1 - \beta$

Infinitesimal gradient ascent (IGA)

- Define the expected payoff to each player
  - $V_r(\alpha,\beta) = \alpha\beta r_{11} + \alpha(1-\beta)r_{12} + (1-\alpha)\beta r_{21} + (1-\alpha)(1-\beta)r_{22}$
    $= u_r\alpha\beta + \alpha(r_{12} - r_{22}) + \beta(r_{21} - r_{22}) + r_{22}$
    with $u_r = r_{11} - r_{12} - r_{21} + r_{22}$
  - $V_c(\alpha,\beta) = \alpha\beta c_{11} + \alpha(1-\beta)c_{12} + (1-\alpha)\beta c_{21} + (1-\alpha)(1-\beta)c_{22}$
    $= u_c\alpha\beta + \alpha(c_{12} - c_{22}) + \beta(c_{21} - c_{22}) + c_{22}$
    with $u_c = c_{11} - c_{12} - c_{21} + c_{22}$

- We can compute the gradient of the payoff function with respect to the strategy as

$$\frac{\partial V_r(\alpha,\beta)}{\partial\alpha} = \beta u_r + (r_{12} - r_{12})$$

$$\frac{\partial V_c(\alpha,\beta)}{\partial\beta} = \alpha u_c + (c_{12} - c_{12})$$

Infinitesimal gradient ascent (IGA)

- The gradient ascent (GA) algorithm then becomes

$$\alpha_{k+1} = \alpha_k + \eta \frac{\partial V_r(\alpha_k, \beta_k)}{\partial \alpha_k}$$

$$\beta_{k+1} = \beta_k + \eta \frac{\partial V_c(\alpha_k, \beta_k)}{\partial \beta_k}$$

Infinitesimal gradient ascent (IGA)

Theorem

If both players follow infinitesimal gradient ascent (IGA), where $\eta \to 0$, then their strategies will converge to a Nash equilibrium, or the average payoffs over time will converge in the limit of the expected payoffs of a Nash equilibrium

- To implement IGA, one needs to know
  - ✓ the payoff matrix in advance (his own)
  - ✓ the current strategy of other agent
- Difficult to select parameters, i.e., step size decaying rate
  - With poorly selected learning rate, the strategy oscillates between 0 and 1
- Not a practical algorithm

WoLF-IGA

- The WoLF-IGA algorithm, introduced by Bowling and Veloso
- allows the player to update its strategy based on the current gradient and a variable learning rate.
- The value of the learning rate is
  - Smaller when the player is wining, and it is
  - larger when the player is losing

- Used in in relatively simple two-action/two-player general-sum games

	β action 1	$1-\beta$ action 2
α action 1	(r_{11}, c_{11})	(r_{12}, c_{12})
$1-\alpha$ action 2	(r_{21}, c_{21})	(r_{22}, c_{22})

WoLF-IGA

- The updating rules of the WoLF-IGA algorithm as follows:

$$\alpha_{k+1} = \alpha_k + \eta l_{r,k} \frac{\partial V_r(\alpha_k, \beta_k)}{\partial \alpha_k}$$

$$\beta_{k+1} = \beta_k + \eta l_{c,k} \frac{\partial V_c(\alpha_k, \beta_k)}{\partial \beta_k}$$

- The varying learning rates, $l_{r,k}$ and $l_{c,k}$, can be computed as

$$l_{r,k} = \begin{cases} l_{min}, & \text{if } V_r(\alpha_k, \beta_k) > V_r(\alpha^*, \beta_k) \\ l_{max}, & \text{otherwise} \end{cases}$$

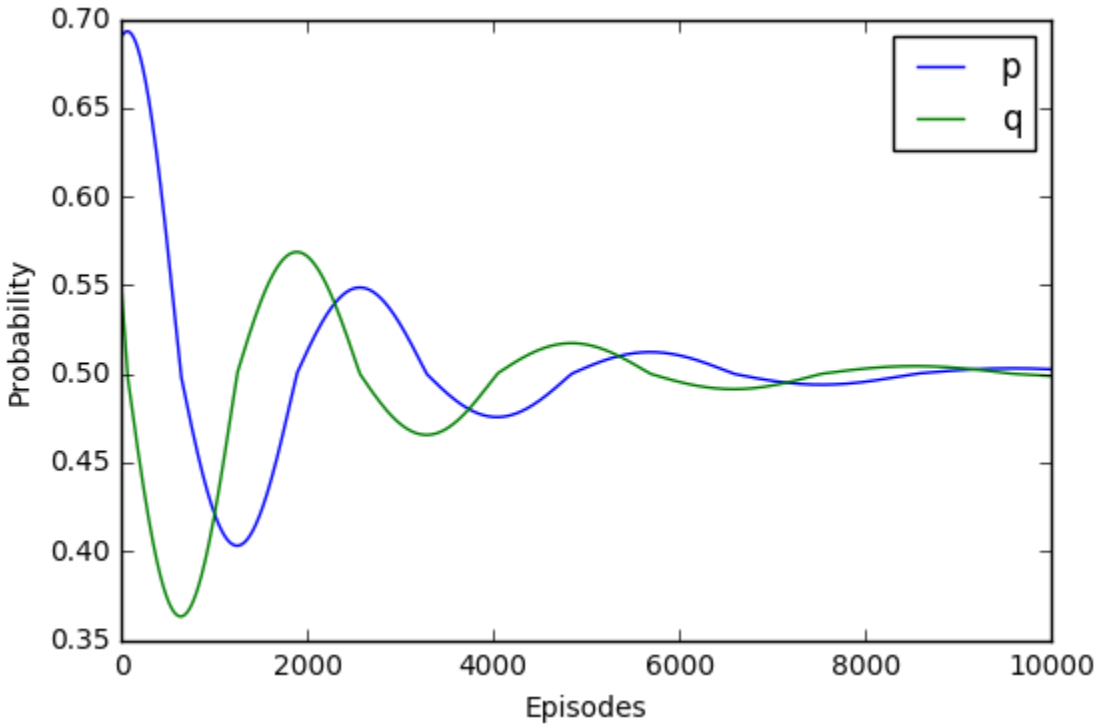
$$l_{c,k} = \begin{cases} l_{min}, & \text{if } V_c(\alpha_k, \beta_k) > V_r(\alpha_k, \beta^*) \\ l_{max}, & \text{otherwise} \end{cases}$$

where (α^*, β^*) is Nash equilibrium

- In a two-player two-action matrix game if each player uses the WoLF-IGA algorithm with $l_{max} > l_{min}$, the players' strategy converges to an NE as the step size $\eta \to 0$.
- It requires the knowledge of $V_r(\alpha^*, \beta_k)$ and $V_r(\alpha_k, \beta^*)$ to compute varying learning rate
  - ➢ To do this, each agent needs to know its own payoff matrix and other's current strategy
- WoLF-IGA algorithm is only available for the two actions

WoLF-IGA

	$Heads$ q	$Tails$ $1-q$
$Heads$ p	$1, -1$	$-1, 1$
$Tails$ $1-p$	$-1, 1$	$1, -1$



Decentralized Learning in Matrix Games

- Decentralized learning means that there is no central learning strategy for all of the agents. Instead, each agent learns its own strategy.
  - Used when an agent has “incomplete information”

  - The agent does not know
    - Its own reward function
    - The other players strategies
    - Other players’ reward function

  - The agent only knows
    - its own action and
    - the received reward at each time step

Policy Hill Climbing (PHC)

- Policy Hill Climbing (PHC) algorithm is a more practical version of the gradient descent algorithm

- PHC is based on Q-learning algorithm

- PHC is a rational algorithm
  - Converge to the optimal mixed strategies if the other players are not learning and are therefore playing stationary strategies
  - However, if the other players are learning (non-stationary), PHC may not converge to a stationary policy

- PHC algorithm learns mixed strategies

- PHC does not require much of information. It does not need to know
  - Agent's payoff matrix
  - Agent's recent actions executed
  - Other agents' current strategies (we don't need to know opponent's strategies!)

Policy Hill Climbing (PHC)

Algorithm Policy hill − climbing (PHC) algorithm for agent *i*

Initialize

learning rate $\alpha \in (0,1], \delta \in (0,1]$

discunt factor $\gamma \in (0,1)$

exploration rate ϵ

$Q_i(a_i) \leftarrow 0$ and $\pi_i(a_i) \leftarrow \frac{1}{|A_i|}\ \forall a_i \in A_i$

Repeat

(a) select an action a_i according to the straegy $\pi(a_i)$ with some exploration rate ϵ

(b) observe the immediate reward r_i

(c) update Q values:

$$Q_i(a_i) = (1-\alpha)Q_i(a_i) + \alpha\left(r_i + \gamma \max_{a_i'} Q_i(a_i')\right)$$

(d) Update the strategy $\pi_i(a_i)$ and constrain it to a legal probability distribution

$$\pi_i(a_i) = \pi_i(a_i) + \begin{cases} \delta & if\ a_i = \max_{a_i'} Q_i(a_i') \\ -\frac{\delta}{|A_i|-1} & otherwise \end{cases}$$

WoLF-PHC

- The WoLF-PHC algorithm is an extension of the PHC algorithm
- It uses the mechanism of win-or-learn-fast (WoLF) so that the PHC algorithm converges to an NE in self-play
- The algorithm has two different learning rates:
  - δ_w when the algorithm is wining
  - δ_l when the algorithm is losing
  - The difference between the average strategy and the current strategy is used as a criterion to decide when the algorithm wins or lose.
- The learning rate δ_l for losing is larger than the learning rate δ_w for winning
  - When a player is losing, it learns faster then when wining
  - Allow the player to
    - adapt quickly to the changes in the strategies of the other player when it is doing more poorly than expected
    - learns cautiously when it is doing better than expected, which also gives the other players the time to adopt to the player's strategy changes

WoLF-PHC

- The WoLF-PHC algorithm is an extension of the PHC algorithm
  - WoLF(win-or-learn-fast) allows variable learning rate→ faster convergence

WoLF

(c) update Q values:

$$Q_i(a_i) = (1-\alpha)Q_i(a_i) + \alpha\left(r_i + \gamma \max_{a_i'} Q_i(a_i')\right)$$

update estimate of average policy $\bar{\pi}(s,a')$:

$$C \leftarrow C + 1$$

$$\forall a' \in A_i, \qquad \bar{\pi}_i(a') \leftarrow \pi_i(a') + \frac{1}{C}\left(\pi_i(a') - \bar{\pi}_i(a')\right)$$

(d) Update the strategy $\pi_i(s,a_i)$ and constrain it to a legal probability distribution

$$\pi_i(a_i) = \pi_i(s,a_i) + \begin{cases} \delta & \text{if } a_i = \max_{a_i'} Q_i(a_i') \\ -\dfrac{\delta}{|A_i|-1} & \text{otherwise} \end{cases}$$

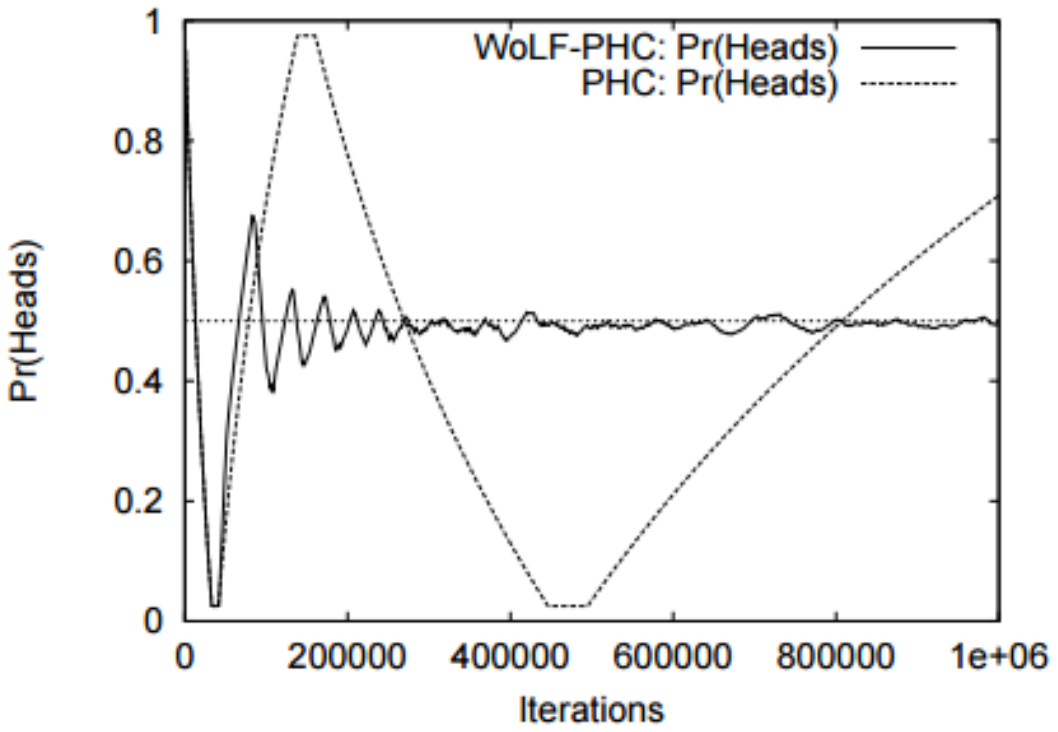
where

$$\delta = \begin{cases} \delta_w & \text{if } \sum_{a_i} \pi_i(a_i)Q_i(a_i) > \sum_{a_i} \bar{\pi}_i(a_i)Q_i(a_i) \\ \delta_l & \text{otherwise} \end{cases}$$

- Average strategy can be computed in running average sense
- The difference between the average strategy and the current strategy is used as a criterion to decide when the algorithm wins or lose.

WoLF-PHC

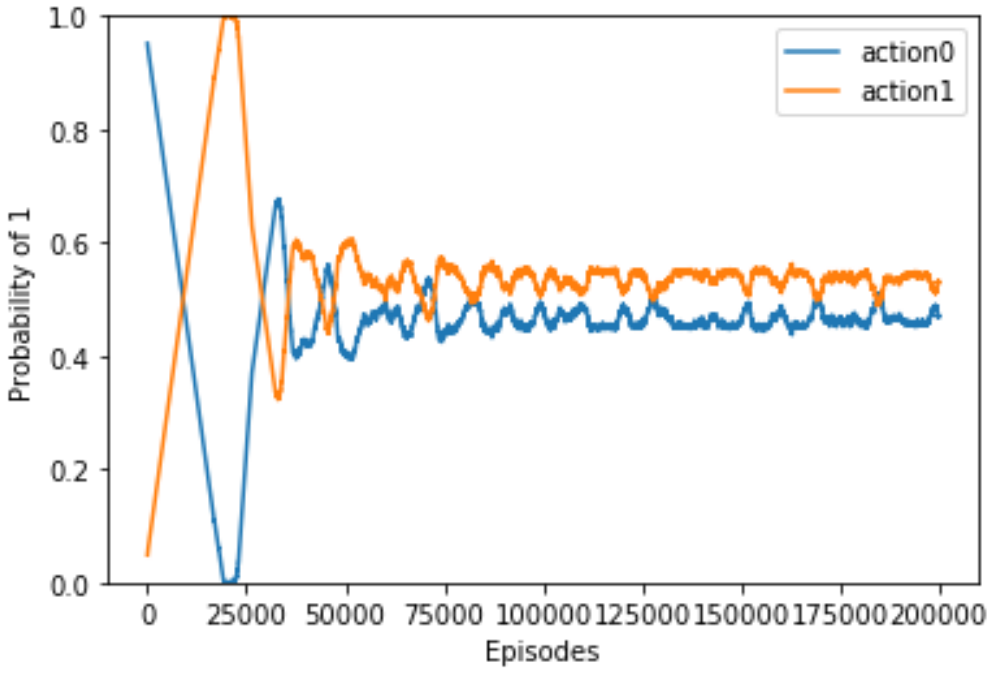
	$Heads$	$Tails$
$Heads$	$1, -1$	$-1, 1$
$Tails$	$-1, 1$	$1, -1$



- The algorithm oscillate about the NE as expected by the theory because both players are learning
- It takes may iterations to converge about the 50% equilibrium point
- Choosing all the parameters is difficult

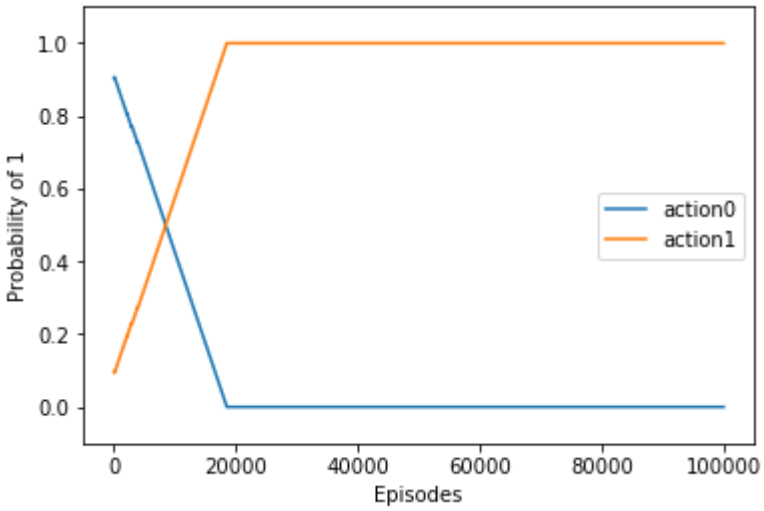
WoLF-PHC

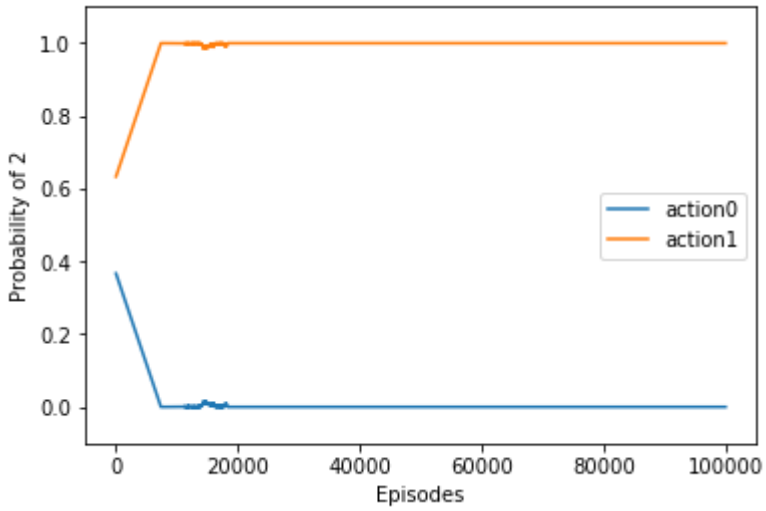
	$Heads$	$Tails$
$Heads$	$1, -1$	$-1, 1$
$Tails$	$-1, 1$	$1, -1$



- The algorithm oscillate about the NE as expected by the theory because both players are learning
- It takes may iterations to converge about the 50% equilibrium point
- Choosing all the parameters is difficult

WoLF-PHC


Probability of 1
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
Episodes
action0
action1


Probability of 2
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
Episodes
action0
action1

WoLF-PHC

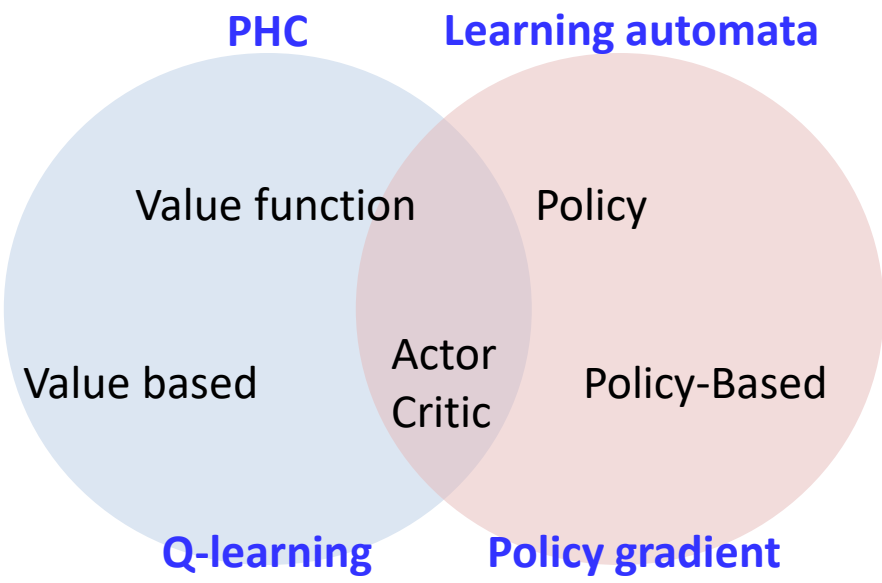
- The WoLF-PHC algorithm exhibits the property of convergences as it makes the player converge to one of its NEs.

- The algorithm is also a rational learning algorithm because
  - It makes the player converge to its optimal strategy when its opponent plays a stationary strategy

- WoLF-PHC algorithm is widely applied to a variety of stochastic games.

Learning Automata

- Learning automation is a learning unit for adaptive decision making in an unknown environment.
- The objective of learning automation is to learning the optimal action or strategy by updating its action probability distribution based on the environment response.
- The learning automata approach is a completely decentralized learning algorithm because each learner only considers its action and the received reward from the environment and ignores any information from other agents such as the actions taken by other agents.
- Two typical learning algorithms based on learning automata
  - Linear reward-inaction (L_{RI})
  - Linear reward-penalty $(L_{R\boldsymbol{P}})$
- It directly learn the policy



Linear reward-inaction (L_{RI}) algorithm

- The linear reward-inaction (L_{RI}) algorithm for player $i (i = 1, \dots, n)$ is defined as follows:

$$\begin{cases} P_c^i(k+1) = p_c^i(k) + \eta r^i(k)\left(1 - p_c^i(k)\right) & \text{if } a_c \text{ is the current action at } k \\ P_j^i(k+1) = p_j^i(k) - \eta r^i(k) p_j^i(k) & \text{for all } a_j^i \neq a_c^i \text{ is the current action at } k \end{cases}$$

  - $p_c^i(k)$ is the probability for player i to select action $a_c^i (c = 1, \dots, m)$ at time k
  - $0 < \eta < 1$ is the learning parameter
  - $r^i(k)$ response of the environment given player i's action a_c^i at k
  - $\sum_j P_j^i(k+1) = \sum_j P_j^i(k)$
  - Increase the probability of selecting the chosen action, while reducing probability of selecting other actions.

- For example, assume a_1^i is selected among $\left(a_1^i, a_2^i, a_3^i, a_4^i\right)$ is the current action at k

$$\begin{cases} P_1^i(k+1) = p_1^i(k) + \eta r^i(k)\left(1 - p_1^i(k)\right) \Rightarrow p_1^i(k) + \eta r^i(k)\left(p_2^i(k) + p_3^i(k) + p_4^i(k)\right) \\ P_2^i(k+1) = p_2^i(k) - \eta r^i(k) p_2^i(k) \\ P_3^i(k+1) = p_3^i(k) - \eta r^i(k) p_3^i(k) \\ P_4^i(k+1) = p_4^i(k) - \eta r^i(k) p_4^i(k) \end{cases}$$

$$\sum_j P_j^i(k+1) = \sum_j P_j^i(k) + \eta r^i(k)\left(1 - \underbrace{\left(p_1^i(k) + p_1^i(k) + p_1^i(k) + p_1^i(k)\right)}_{1}\right) = \sum_j P_j^i(k)$$

Linear reward-inaction (L_{RI}) algorithm

- The linear reward-inaction (L_{RI}) algorithm for player $i (i = 1, \dots, n)$ is defined as follows:

$$\begin{cases} P_c^i(k+1) = p_c^i(k) + \eta r^i(k)\left(1 - p_c^i(k)\right) & \text{if } a_c \text{ is the current action at } k \\ P_j^i(k+1) = p_j^i(k) - \eta r^i(k) p_j^i(k) & \text{for all } a_j^i \neq a_c^i \text{ is the current action at } k \end{cases}$$

  - $p_c^i(k)$ is the probability for player i to select action $a_c^i (c = 1, \dots, m)$ at time k
  - $0 < \eta < 1$ is the learning parameter
  - $r^i(k)$ response of the environment given player i's action a_c^i at k
  - $\sum_j P_j^i(k+1) = \sum_j P_j^i(k)$
  - Increase the probability of selecting the chosen action, while reducing probability of selecting other actions.

- In a matrix game with n players, if each player uses the L_{RI} algorithm,
  - Convergence to NE under the assumption that the game only has strict NEs in pure strategies (S. Lakshmivarahan, 1981)

Linear reward-penalty (L_{RP}) algorithm

- The linear reward-penalty (L_{RP}) algorithm for player $i (i = 1, \dots, n)$ is defined as follows:

$$\begin{cases} P_c^i(k+1) = p_c^i(k) + \eta_1 r^i(k)\left(1 - p_c^i(k)\right) - \eta_2\left[1 - r^i(k)\right]p_c^i(k) & \text{if } a_c \text{ is the current action} \\ P_j^i(k+1) = p_j^i(k) - \eta_1 r^i(k) p_j^i(k) + \eta_2\left[1 - r^i(k)\right]\left[\frac{1}{m-1} - p_j^i(k)\right] & \text{for all } a_j^i \neq a_c^i \end{cases}$$

  - $p_c^i(k)$ is the probability for player i to select action $a_c^i (c = 1, \dots, m)$ at time k
  - $0 < \eta_1, \eta_2 < 1$ is the learning parameters
  - $r^i(k)$ response of the environment given player i's action a_c^i at k
  - $\sum_j P_j^i(k+1) = \sum_j P_j^i(k)$

- For example, assume a_1^i is chosen among $\left(a_1^i, a_2^i, a_3^i, a_4^i\right)$ is the current action at k

$$\begin{cases} P_1^i(k+1) = p_1^i(k) + \eta_1 r^i(k)\left(1 - p_1^i(k)\right) & - \eta_2\left[1 - r^i(k)\right]p_1^i(k) \\ P_2^i(k+1) = p_2^i(k) - \eta_1 r^i(k) p_2^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_2^i(k)\right] \\ P_3^i(k+1) = p_3^i(k) - \eta_1 r^i(k) p_3^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_3^i(k)\right] \\ P_4^i(k+1) = p_4^i(k) - \eta_1 r^i(k) p_4^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_4^i(k)\right] \end{cases}$$

If $r^i(k)$ is large, give a larger reward

If $r^i(k)$ is small, give a larger penalty

Linear reward-penalty (L_{RP}) algorithm

- The linear reward-penalty (L_{RP}) algorithm for player $i (i = 1, \dots, n)$ is defined as follows:

$$\begin{cases} P_c^i(k+1) = p_c^i(k) + \eta_1 r^i(k)\left(1 - p_c^i(k)\right) - \eta_2\left[1 - r^i(k)\right]p_c^i(k) & \text{if } a_c \text{ is the current action} \\ P_j^i(k+1) = p_j^i(k) - \eta_1 r^i(k) p_j^i(k) + \eta_2\left[1 - r^i(k)\right]\left[\frac{1}{m-1} - p_j^i(k)\right] & \text{for all } a_j^i \neq a_c^i \end{cases}$$

  - $p_c^i(k)$ is the probability for player i to select action $a_c^i (c = 1, \dots, m)$ at time k
  - $0 < \eta_1, \eta_2 < 1$ is the learning parameters
  - $r^i(k)$ response of the environment given player i's action a_c^i at k
  - $\sum_j P_j^i(k+1) = \sum_j P_j^i(k)$

- For example, assume a_1^i is chosen among $\left(a_1^i, a_2^i, a_3^i, a_4^i\right)$ is the current action at k

$$\begin{cases} P_1^i(k+1) = p_1^i(k) + \eta_1 r^i(k)\left(1 - p_1^i(k)\right) & - \eta_2\left[1 - r^i(k)\right]p_1^i(k) \\ P_2^i(k+1) = p_2^i(k) - \eta_1 r^i(k) p_2^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_2^i(k)\right] \\ P_3^i(k+1) = p_3^i(k) - \eta_1 r^i(k) p_3^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_3^i(k)\right] \\ P_4^i(k+1) = p_4^i(k) - \eta_1 r^i(k) p_4^i(k) & + \eta_2\left[1 - r^i(k)\right]\left[1/3 - p_4^i(k)\right] \end{cases}$$

$$\begin{aligned} \sum_j P_j^i(k+1) &= \sum_j P_j^i(k) + \eta_1 r^i(k)\left(1 - \left(p_1^i(k) + p_1^i(k) + p_1^i(k) + p_1^i(k)\right)\right) \\ &\quad + \eta_2\left[1 - r^i(k)\right]\left(1/3 + 1/3 + 1/3 - \left(p_1^i(k) + p_1^i(k) + p_1^i(k) + p_1^i(k)\right)\right) \\ &= \sum_j P_j^i(k) \end{aligned}$$

Linear reward-penalty (L_{RP}) algorithm

- The linear reward-penalty (L_{RP}) algorithm for player $i (i = 1, \dots, n)$ is defined as follows:

$$\begin{cases} P_c^i(k+1) = p_c^i(k) + \eta_1 r^i(k)\left(1 - p_c^i(k)\right) - \eta_2\left[1 - r^i(k)\right]p_c^i(k) & \text{if } a_c \text{ is the current action} \\ P_j^i(k+1) = p_j^i(k) - \eta_1 r^i(k) p_j^i(k) + \eta_2\left[1 - r^i(k)\right]\left[\frac{1}{m-1} - p_j^i(k)\right] & \text{for all } a_j^i \neq a_c^i \end{cases}$$

  - $p_c^i(k)$ is the probability for player i to select action $a_c^i (c = 1, \dots, m)$ at time k
  - $0 < \eta_1, \eta_2 < 1$ is the learning parameters
  - $r^i(k)$ response of the environment given player i's action a_c^i at k
  - $\sum_j P_j^i(k+1) = \sum_j P_j^i(k)$

- fIn a two-player zero-sum matrix game, if each player uses the L_{RP} and chooses $\eta_2 < \eta_1$, then the expected value of the fully mixed strategies for both players can be made arbitrarily close to an NE (S. Lakshmivarahan, 1982)

- This means that the L_{RP} algorithm can guarantee the convergence to an NE in the sense of expected value but not the players' strategy itself.

Comparison

	Algorithms			
	Centralized	**Decentralized**		
Applicability	**WoLF-IGA**	**WoLF-PHC**	L_{RI}	L_{RP}
Allowable actions	Two actions	No limit	No limit	No limit
Convergence	Pure NE & Mixed NE	Pure NE & Mixed NE	Pure NE	Fully Mixed NE