Multi-Agent Reinforcement Learning

吉建民

USTC
jianmin@ustc.edu.cn

2024 年 1 月 1 日

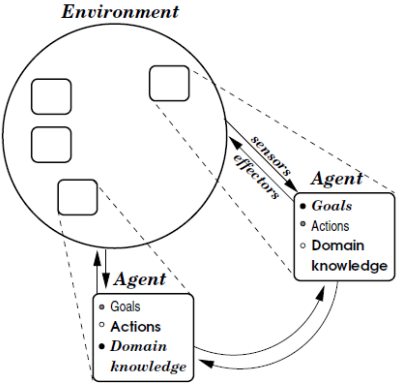
Used Materials

Disclaimer: 本课件大量采用了 Rich Sutton's RL class, David Silver's Deep RL tutorial 和其他网络课程课件，也采用了 GitHub 中开源代码，以及部分网络博客内容

Table of Contents

Multiagent Systems

- Multiple agents interact in common environment
- Each agent with own sensors, effectors, goals, ...
- Agents have to coordinate actions to achieve goals



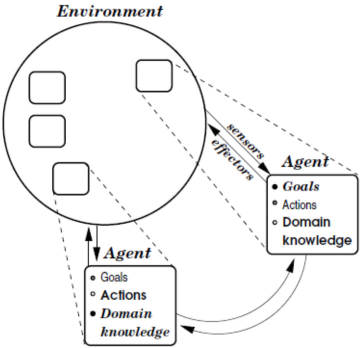
Multiagent Systems

Environment defined by:

- state space
- available actions
- effects of actions on states
- what agents can observe

Agents defined by:

- domain knowledge
- goal specification
- policies for selecting actions



Many problems can be modelled as multiagent systems!

Multiagent Systems: Applications

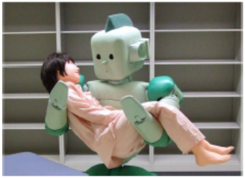
Chess

Poker

Starcraft

Robot soccer

Home assistance



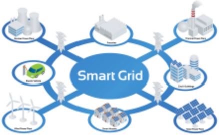
Autonomous cars

Multiagent Systems: Applications

Negotiation

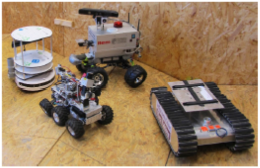
Wireless networks

Smart grid



User interfaces

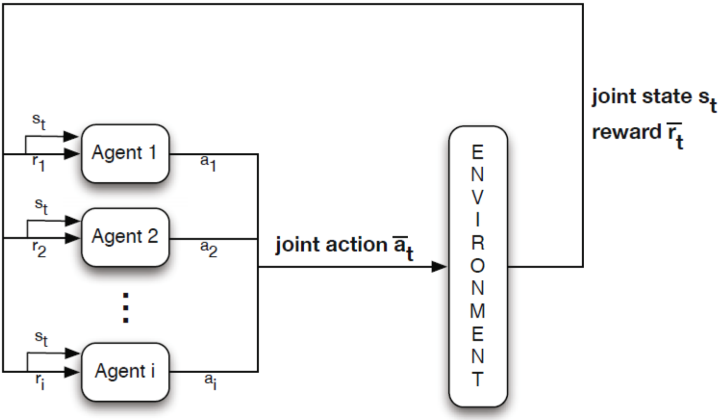
Multi-robot rescue



MARL: Motivation

- Optimal solutions for the decision problem, particularly in multi-agent systems, are sometimes not obvious to the programmer.
- **So...** Multi-agent reinforcement learning provides a way of programming agents without the complete knowledge of the task.
- **But...** Reinforcement Learning for the single-agent domain can't always be used in a multi-agent scenario.
- **So...** there is the need to study specific reinforcement learning techniques in the presence of other agents.

MARL



joint state s_t
reward $\overline{r}_t$
s_t
r_1
Agent 1
a_1
s_t
r_2
Agent 2
a_2
joint action $\overline{a}_t$
s_t
r_i
Agent i
a_i
ENVIRONMENT

Table of Contents

Game Theory

- Models strategic interactions as games
- In normal-form games (matrix games), all players simultaneously select an action, and their joint action determines their individual payoff
  - One-shot interaction
  - Can be represented as an n-dimensional payoff matrix, for n players
- A player's strategy is defined as a probability distribution over his possible actions
- Stochastic games is an extension of normal-form games and MDPs in the sense that they deal with multiple agents in a multiple state situation.

Normal-Form Game

- A normal-form game can be defined as a tuple $(n, A_{1\ldots n}, R_{1\ldots n})$ where:
  - n is the number of agents
  - A_i is the action set for player i
    - $A = A_1 \times \cdots \times A_n$ is the joint action set
  - $R_i : A \to \mathbb{R}$ is the reward function of player i
- Each agent i selects policy $\pi_i : A_i \to [0, 1]$ $(\pi_i \in PD(A_i))$, takes action $a_i \in A_i$ with probability $\pi_i(a_i)$, and receives utility $R_i(a_1, \ldots, a_n)$
- Given policy profile $\langle \pi_1, \ldots, \pi_n \rangle$, expected utility to i is

$$R_i(\pi_1, \ldots, \pi_n) = \sum_{a \in A} R_i(a) \prod_{j=1}^{n} \pi_j(a_j)$$

- Agents want to maximise their expected utilities

Normal-Form Game: Prisoners' Dilemma

Example: Prisoner's Dilemma

- Two prisoners questioned in isolated cells
- Each prisoner can **Cooperate** or **Defect**
- Utilities (row = agent 1, column = agent 2):

	C	D
C	-1,-1	-5,0
D	0,-5	-3,-3

Normal-Form Game: Rock-Paper-Scissors

Example: Rock-Paper-Scissors

- Two players, three actions
- **Rock** beats **Scissors** beats **Paper** beats **Rock**
- Utilities:

	R	P	S
R	0,0	-1,1	1,-1
P	1,-1	0,0	-1,1
S	-1,1	1,-1	0,0

Optimality Concepts

Optimality Concepts in Normal-Form Games:

- Best-Response Function: set of optimal strategies given the other agents current strategies.

$$\begin{array}{ll} \pi_i^* \in BR_i(\pi_{-i}) & \text{iff} \\ \forall \pi_i \in PD(A_i) & R_i(\langle \pi_i^*, \pi_{-i} \rangle) \geq R_i(\langle \pi_i, \pi_{-i} \rangle) \end{array}$$

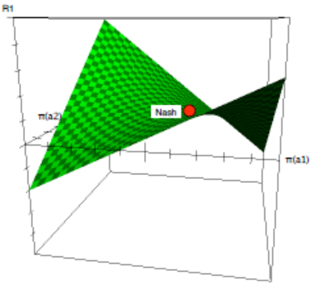
- Nash Equilibria: all agents are using best-response strategies.

$$\forall i = 1 \ldots n \quad \pi_i \in BR_i(\pi_{-i})$$

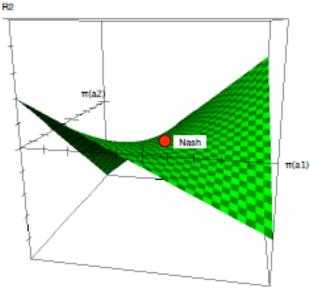
- All Normal-Form Games have at least one Nash Equilibrium

Game Classification: Zero-sum

- 2 players with opposing objectives.
- There is only one Nash equilibrium
  - Minimax to find it.



(a) Reward function for player 1



(b) Reward function for player 2

Two-Player Zero-Sum Games

- Characteristics:
  - Two opponents play against each other.
  - symmetrical rewards (always sum zero).
  - Usually only one equilibrium and if more exist they are interchangeable
    - Interchangeable: $\langle \pi_1, \pi_2 \rangle$ 和 $\langle \mu_1, \mu_2 \rangle$ 是两个 Nash equilibria，则 $\langle \pi_1, \mu_2 \rangle$，$\langle \mu_1, \pi_2 \rangle$ 也是 Nash equilibria；并且它们效用都相等
- Minimax to find an equilibrium $(2, A, O, R, -R)$:

$$\max_{\pi \in PD(A)} \min_{o \in O} \sum_{a \in A} \pi(a) R(a, o)$$

  - Formulated as a Linear Program.
  - Solution in the strategy space: simultaneous playing invalidates deterministic strategies.

Minimax

- A value function defines the expected total reward given joint policies $\pi = \langle \pi^1, \pi^2 \rangle$

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

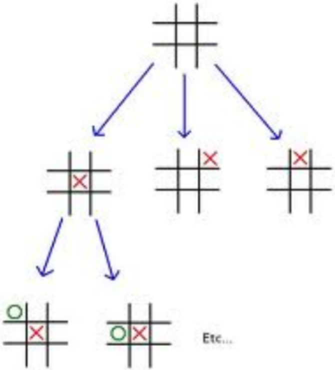
- A minimax value function maximizes white's expected return while minimizing black's expected return

$$v_*(s) = \max_{\pi^1} \min_{\pi^2} v_\pi(s)$$

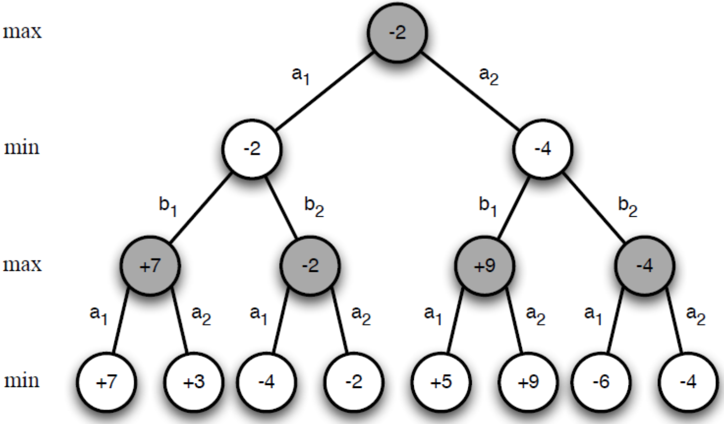
- A minimax policy is a joint policy $\pi = \langle \pi^1, \pi^2 \rangle$ that achieves the minimax values
- There is a unique minimax value function
- A minimax policy is a Nash equilibrium

Minimax Search

- Minimax values can be found by depth-first game-tree search
- Introduced by Claude Shannon: *Programming a Computer for Playing Chess*
- Ran on paper!



Minimax Search Example

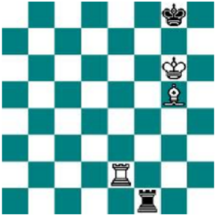


Value Function in Minimax Search

- Search tree grows exponentially
- Impractical to search to the end of the game
- Instead use value function approximator $v(s, \mathbf{w}) \approx v^*(s)$
- Use value function to estimate minimax value at leaf nodes
- Minimax search run to fixed depth with respect to leaf values

Binary-Linear Value Function

- Binary feature vector $\mathbf{x}(\mathbf{s})$: e.g. one feature per piece
- Weight vector $\mathbf{w}$: e.g. value of each piece
- Position is evaluated by summing weights of active features

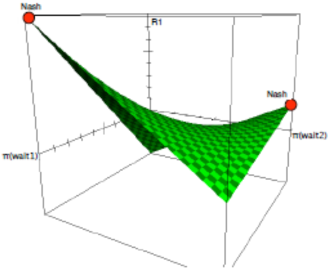


$$v(s, \mathrm{w}) = \mathrm{x}(s) \cdot \mathrm{w} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} +5 \\ +3 \\ +1 \\ -5 \\ -3 \\ -1 \\ \vdots \end{bmatrix} \begin{matrix} \text{♖} \\ \text{♗} \\ \text{♙} \\ \text{♜} \\ \text{♝} \\ \text{♟} \\ \end{matrix}$$

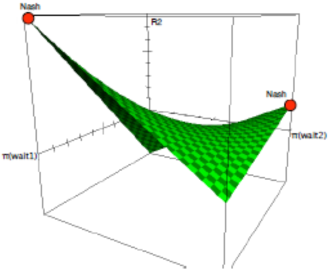
$$v(s, \mathrm{w}) = 5 + 3 - 5 = 3$$

Game Classification: Team

- N players with the same objective.
- Nash equilibria are deterministic.
  - Just look for higher payoffs.



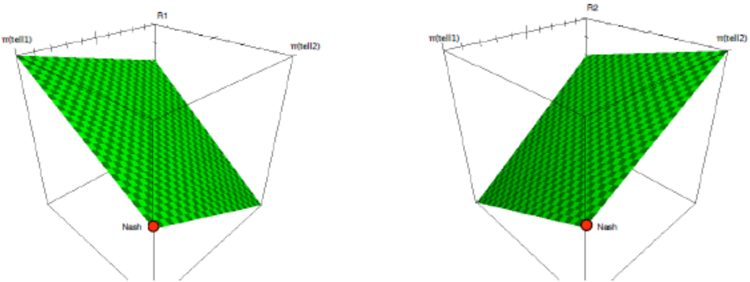
(a) Reward function for player 1



(b) Reward function for player 2

Game Classification: General-sum

- All kinds of games.
- Several Nash equilibria requiring complex solutions.
  - With 2 players it is possible to use quadratic programming.



(a) Reward function for player 1

(b) Reward function for player 1

Stochastic Game

- Multiple-state / Multiple-agent environment. Like an extension of MDPs and Normal-Form Games.
- Markovian but not from each player's point of view.
- A stochastic game is a tuple $(n, S, A_{1,\dots,n}, T, R_{1,\dots,n})$ where:
  - n represents the number of agents
  - S the state set
  - A_i the action set of agent i and $A = A_1 \times \cdots \times A_n$ the joint action set
  - $T : S \times A \times S \to [0, 1]$ is a transition function which depends on the actions of all players
  - $R : S \times A \times S \to \mathbb{R}$ is a reward function representing the expected value of the next reward, which also depends on the actions of all players.
- Each agent i selects policy $\pi_i : S \to PD(A_i)$ (probability $\pi_i(a_i \mid s)$)
- Joint policy $\pi = \langle \pi_i, \pi_{-i} \rangle$

Optimality Concepts in Stochastic Games

Optimality Concepts in Stochastic Games:

- The discounted reward over time is usually considered, as in MDPs:

$$V_i^{\pi}(s) = E\left[\sum_{k=0}^{\infty} \gamma^k r^i_{t+k+1} \mid s_t = s, \pi\right]$$
$$= \sum_a \pi(s,a) \sum_{s'} T(s,a,s')\left(R_i(s,a,s') + \gamma V_i^{\pi}(s')\right)$$
$$Q_i^{\pi}(s,a) = \sum_{s'} T(s,a,s')\left(R_i(s,a,s') + \gamma V_i^{\pi}(s')\right)$$

- Best-response function: defined for policies with the state values as reference.

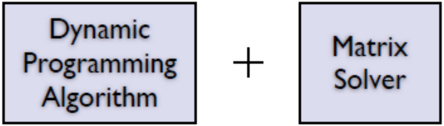
$$\begin{array}{ll} \pi_i^* \in BR_i(\pi_{-i}) & \text{iff} \\ \forall \pi_i \in S \times PD(A_i), \forall s \in S & V_i^{\langle \pi_i^*, \pi_{-i}\rangle}(s) \geq V_i^{\langle \pi_i, \pi_{-i}\rangle}(s) \end{array}$$

- Nash equilibria: All players are using best-response policy.

$$\forall i = 1 \ldots n \quad \pi_i \in BR_i(\pi_{-i})$$

Solving Stochastic Games

- Usually, each algorithm solves one type of game.
- A common approach:



Minimax Value Iteration

- Suitable for (two-person) zero-sum stochastic games.

Value Iteration Algorithm + Minimax Solver in each State

- Algorithm expression (based on the Bellman optimality equation for the zero-sum SG)

$$V^{k+1}(s) \leftarrow \max_{\pi \in PD(A)} \min_{o \in O} \sum_{a \in A} \pi(a) Q^{k+1}(s, a, o)$$

$$Q^{k+1}(s, a, o) \leftarrow \sum_{s'} R(s, a, o, s') + \gamma T(s, a, o, s') V^{k}(s')$$

Stationary Opponents（固定对手）

- The game reduces to an MDP with:

$$S^{MDP} = S^{SG}$$

$$A^{MDP} = A_i^{SG}$$

$$T^{MDP}(s, a_i, s') = \sum_{a_{-i} \in A_{-i}^{SG}} \pi_{-i}(s, a_{-i})\, T^{SG}(s, \langle a_i, a_{-i} \rangle, s')$$

$$R^{MDP}(s, a_i, s') = \sum_{a_{-i} \in A_{-i}^{SG}} \pi_{-i}(s, a_{-i})\, T^{SG}(s, \langle a_i, a_{-i} \rangle, s')\, R^{SG}(s, \langle a_i, a_{-i} \rangle, s')$$

Table of Contents

Best-Response Learners

- Not specifically concerned with Nash equilibria.
- Try to learn a policy that is optimal with respect to the policies of the other players.
- This methods adapt to the other players trying taking advantage of their weaknesses.
- Three popular approaches:
  - MDP methods
  - Joint-action learners (JALs) and Opponent Modelling
  - WoLF (Win or Learn Fast) Policy Hill Climber

MDP Methods

- Use reinforcement learning methods for Markov Decision Processes to learn in Stochastic Games: *Q-learning, Sarsa, Actor-critic, ...*
- Some success with this approach (Tan, 93; Sen *et al*, 94).
- **Pros:**
  - Simple implementation.
- **Cons:**
  - Cannot learn stochastic policies (MDP optimal is deterministic).
  - Environment is not stationary from the agent's point of view (MDP methods assume stationarity).

JALs

It assumes: full observability of the state and of the other agents' actions

- Learn Q-values based on joint actions.
- Maintain statistics of the opponents actions to compute joint policies.
  - In JALs when deciding, Q-values are replaced by:

$$EV(a_i) = \sum_{a_{-i} \in A_{-i}} Q(\langle a_i, a_{-i} \rangle) \prod_{j \neq i} \widehat{\pi}_j(a_{-i}[j])$$

- **Pros:**
  - Use information of the other players.
- **Cons:**
  - Also learn deterministic policies (max operator).

Opponent Modeling

Opponent Modeling is similar to to JALs and the estimator is:

$$\widehat{\pi_{-i}}(a_{-i}) = \frac{n(s, a_{-i})}{n(s)}$$

Algorithm 3.1 Opponent modeling

Initialize $Q(s, a)$ arbitrarily
$\forall_{s \in S} \forall_{a_{-i} \in A_{-i}} \quad n(s) \leftarrow 0$ and $n(s, a_{-i}) \leftarrow 0$

Initialize s
loop
$a_i \leftarrow$ probabilistic outcome of policy (e.g. ϵ-greedy) based on $O(s, a_i)$
with $O(s, a_i) = \sum\limits_{a_{-i}} \frac{n(s,a_{-i})}{n(s)} Q(s, \langle a_i, a_{-i} \rangle)$

Take action a_i, observe reward r, next state s' and other players joint action a_{-i}

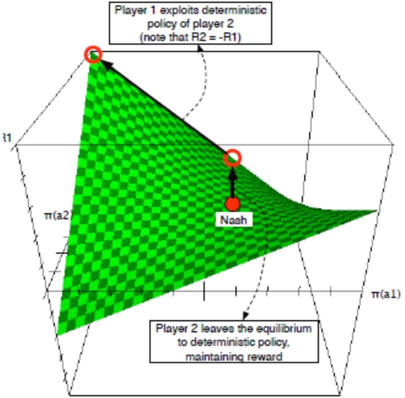
$Q(s, \langle a_i, a_{-i} \rangle) \leftarrow Q(s, \langle a_i, a_{-i} \rangle) + \alpha\big(r + \gamma V(s') - Q(s, \langle a_i, a_{-i} \rangle)\big)$
with $V(\sigma) = \max_{a_i} \sum\limits_{a_{-i}} \frac{n(s,a_{-i})}{n(s)} Q(s, \langle a_i, a_{-i} \rangle)$

$n(s, a_{-i}) \rightarrow n(s, a_{-i}) + 1$
$n(s) \rightarrow n(s) + 1$

$s \leftarrow s'$
end loop

Deterministic vs Stochastic

- Deterministic policies can be exploited.
  - Most Nash equilibria are stochastic...
- An example:



WoLF Policy Hill Climber

- Modifies the policy directly (Hill Climbing procedure)
- WoLF stands for *Win or Learn Fast*, meaning that the learning rate changes when the agent is winning/ loosing.

$$\sum_{a'} \pi(s,a')Q(s,a') > \sum_{a'} \tilde{\pi}(s,a')Q(s,a')$$

- **Pros:**
  - Can learn stochastic policies.
  - Variable learning rate controls exploration.
  - Converges to Nash when all are playing best-response.
- **Cons:**
  - Assumes convergence to stationary policies of the other agents.

Hill Climbing（爬山法）

一种局部最优搜索算法:

1. Pick initial state s
2. Pick t in neighbors(s) with the largest $f(t)$
3. If $f(t) \leq f(s)$ then stop, return s
4. $s = t$, GOTO 2.

WoLF Policy Hill Climber Algorithm

Algorithm 3.2 WoLF Policy Hill Climbing

Initialize $Q(s,a)$ and π arbitrarily (e.g. $\pi(s,a) \rightarrow \frac{1}{|A_i|}$)
$\forall_{s \in S}\ n(s) \leftarrow 0.$

Initialize s
loop
$a \leftarrow$ probabilistic outcome of policy $\pi(s)$ {Mixed with exploration policy}

Take action a, observe reward r and next state s'
$Q(s,a) \leftarrow Q(s,a) + \alpha\,(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

Update average policy $\tilde{\pi}$:

$$n(s) \leftarrow n(s) + 1$$

$$\tilde{\pi}(s) \leftarrow \tilde{\pi}(s) + \frac{1}{n(s)}\,(\pi(s) - \tilde{\pi}(s))$$

if $\sum_{a'} \pi(s,a')Q(s,a') > \sum_{a'} \tilde{\pi}(s,a')Q(s,a')$ **then**
$\delta \leftarrow \delta_w$ (winning)
else
$\delta \leftarrow \delta_l$ (loosing)
end if

WoLF Policy Hill Climber Algorithm

Update policy $\pi(s)$:

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

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

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

Normalize $\pi(s)$.

$s \leftarrow s'$

end loop

Table of Contents

Equilibrium learners

- Specifically try to learn Nash equilibrium polices.
- Basic idea: a Nash equilibrium policy is a collection of Nash equilibrium strategies for Normal-form Games (one for each state). So an equilibrium policy π^* and a state s, the normal-form game whose equilibrium is the strategy $\pi^*(s)$ can be defined by the following rewards:
$$R_i(a) = Q_i^{\pi^*}(s, a)$$
- The solution for an equilibrium learner would be a fixed point in π^* of the following system of equations:
$$\forall i = 1 \dots n \quad Q_i^*(s, a) = \sum_{s'} R_i(s, a, s') + \gamma T(s, a, s') V_i^{\pi^*}(s')$$
where $V_i^{\pi^*}(s')$ represents the equilibrium value for agent i when the joint-policy being played is the Nash equilibrium π^*
- The Q-values can be computed like Q-learning:
$$\forall i = 1 \dots n \quad Q_i(s, a) \leftarrow Q_i(s, a) + \alpha \left(r_i + \gamma V_i(s') - Q_i(s, a) \right)$$
where the state value V is computes as the Nash equilibrium value for agent i
- Problem: Several Nash equilibria!

General equilibrium learner algorithm

Algorithm 3.3 General equilibrium learner algorithm

Initialize $Q(s, a)$ arbitrarily

Initialize s

loop

$a_i \leftarrow$ probabilistic outcome of Nash policy derived from $Q(s, a)$, for player i {Mixed with exploration policy}

Take action a_i, observe reward r, next state s' and the joint action of other players a_{-i}

for $i = 1 \ldots n$ **do**

$Q_i(s, \langle a_i, a_{-i} \rangle) \leftarrow Q_i(s, \langle a_i, a_{-i} \rangle) + \alpha \big(r_i + \gamma V_i(s') - Q_i(s, \langle a_i, a_{-i} \rangle) \big)$

end for

where $V(s) = Nash\,([Q(s, a)])$

$s \leftarrow s'$

end loop

Minimax-Q

- Find Nash equilibria in zero-sum games.
- Nash state values can be found with minimax:

$$V(s) = \max_{\pi \in PD(A)} \min_{o \in O} \sum_{a \in A} \pi(s,a) Q(s, \langle a, o \rangle)$$

  - Can be formulated as a linear program.
- **Pros:**
  - Lower bound for agent performance.
  - Convergence has been prooved – very solid for its domain.
- **Cons:**
  - Large actions spaces lead to big linear programs.

Minimax-Q Algorithm

Algorithm 3.4 Minimax-Q learner

Initialize $Q(s, \langle a, o \rangle)$ and $\pi(s)$ arbitrarily

Initialize s

loop

$a \leftarrow$ probabilistic outcome of $\pi(s)$ {Mixed with exploration policy}

Take action a, observe reward r, next state s' and opponent action o

$$Q(s, \langle a, o \rangle) \leftarrow Q(s, \langle a, o \rangle) + \alpha\big(r + \gamma V(s') - Q(s, \langle a, o \rangle)\big)$$

with $V(s) = \max\limits_{\pi' \in PD(A)} \min\limits_{o' \in O} \sum\limits_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$

$$\pi(s) \rightarrow \arg \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$$

$s \leftarrow s'$

end loop

Nash-Q

- Addresses the problem of learning in 2-player general-sum games.
- Quadratic programming to find Nash state values.
- Several equilibria (which one to choose??). Solved by strict conditions.
- **Pros:**
  - Applicable to a wider range of problems.
- **Cons:**
  - Convergence conditions are too strict and unrealistic
    - All intermediate games must have one equilibrium AND
    - It must be either a saddle point (like zero-sum games) or a global maximum (like team games).

Friend-or-Foe-Q

- Motivated by the assumptions of Nash-Q, it is restricted to a class of problems:
  - The agent is either playing against a Foe or with a Friend, and is informed by an external oracle.
- Two different solutions:
  - *Friend*: the game is cooperative and has a Nash at a global maximum – found using **max** operator (like MDPs).
  - *Foe*: the game is adversarial and has a Nash at a saddle point – use **minimax** operator (like Minimax-Q).
- **Pros:**
  - Solid in its domain (no strange convergence conditions).
- **Cons:**
  - When playing *Friend*, might need an oracle too coordinate equilibrium choice (all with the same payoff) – does learning make sense in this situation?

Friend-or-Foe-Q Algorithm

Algorithm 3.5 Friend-or-Foe-Q learner

Initialize $Q(s, \langle a, o\rangle)$ and $\pi(s)$ arbitrarily

Initialize s

loop

$a \leftarrow$ probabilistic outcome $\pi(s)$ {Mixed with exploration policy}

Take action a, observe reward r, next state s' and opponent action o

$Q(s, \langle a, o\rangle) \leftarrow Q(s, \langle a, o\rangle) + \alpha\big(r + \gamma V(s') - Q(s, \langle a, o\rangle)\big)$

where

if Playing against foe **then**

$$V(s) = \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o'\rangle)$$

$$\pi(s) \rightarrow \arg \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o'\rangle)$$

else

$$V(s) = \max_{a' \in A, o' \in O} Q(s, \langle a', o'\rangle)$$

$$\pi(s, a) = \begin{cases} 1 & a = \arg\max_{a' \in A} \{ \max_{o' \in O} Q(s, \langle a', o'\rangle)\} \\ 0 & \text{otherwise} \end{cases}$$

end if

$s \leftarrow s'$

end loop

Table of Contents

Single-Agent and Self-Play Reinforcement Learning

- Best response is solution to single-agent RL problem
  - Other players become part of the environment
  - Game is reduced to an MDP
  - Best response is optimal policy for this MDP
- Nash equilibrium is fixed-point of self-play RL
  - Experience is generated by playing games between agents

$$a_1 \sim \pi^1, a_2 \sim \pi^2, \ldots$$

  - Each agent learns best response to other players
  - One player's policy determines another player's environment
  - All players are adapting to each other

Self-Play Temporal-Difference Learning

- Apply value-based RL algorithms to games of self-play
- MC: update value function towards the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(0): update value function towards successor value $v(S_{t+1})$

$$\Delta \mathbf{w} = \alpha(v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(λ): update value function towards the λ-return G_t^λ

$$\Delta \mathbf{w} = \alpha(G_t^\lambda - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

Policy Improvement with Afterstates

- For deterministic games it is sufficient to estimate $v_*(s)$
- This is because we can efficiently evaluate the afterstate

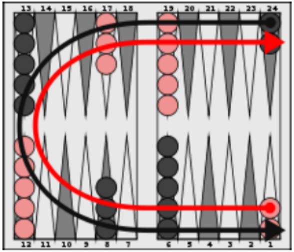
$$q_*(s, a) = v_*(succ(s, a))$$

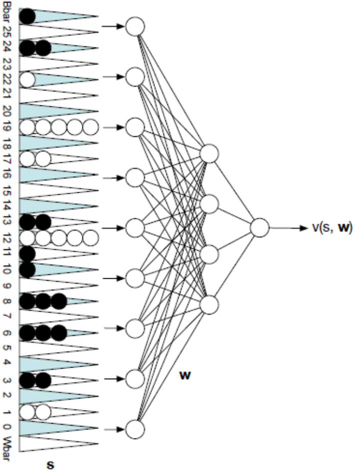
- Rules of the game define the successor state $succ(s, a)$
- Actions are selected e.g. by min/maximising afterstate value

$$A_t = \underset{a}{\operatorname{argmax}}\ v_*(succ(S_t, a)) \qquad \text{for white}$$
$$A_t = \underset{a}{\operatorname{argmin}}\ v_*(succ(S_t, a)) \qquad \text{for black}$$

- This improves joint policy for both players

TD Gammon: Non-Linear Value Function Approximation





Self-Play TD in Backgammon: TD-Gammon

- Initialised with random weights
- Trained by games of self-play
- Using non-linear temporal-difference learning

$$\delta_t = v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta_t \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- Greedy policy improvement (no exploration)
- Algorithm always converged in practice
- Not true for other games