Multi-Agent Reinforcement Learning

Multi-Agent Deep Reinforcement Learning – Part 2

Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer

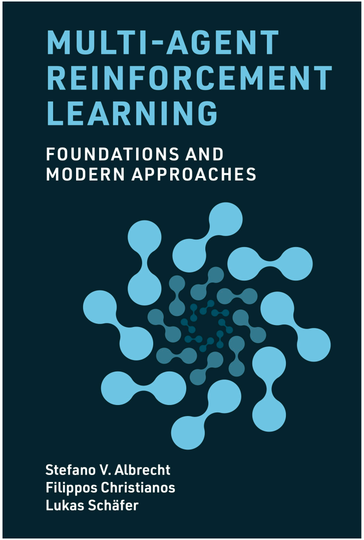
This lecture is based on

Multi-Agent Reinforcement Learning: Foundations and Modern Approaches

by Stefano V. Albrecht, Filippos Christianos and Lukas Schäfer

MIT Press, 2024

Download book, slides, and code at:
www.marl-book.com



Lecture Outline

- Agent modeling with deep learning
- Parameter and experience sharing
- Policy self-play in zero-sum games
- Population-based training

Agent Modeling with Deep Learning

In MARL, agents need to consider the policies of other agents to coordinate their actions.

Agents Modeling – Motivation

In MARL, agents need to consider the policies of other agents to coordinate their actions.

Approaches presented so far account for the action selection of other agents through:

- Distribution of training data is dependent on the policies of all agents
- Training centralized critics conditioned on the actions of other agents

Agents Modeling – Motivation

In MARL, agents need to consider the policies of other agents to coordinate their actions.

Approaches presented so far account for the action selection of other agents through:

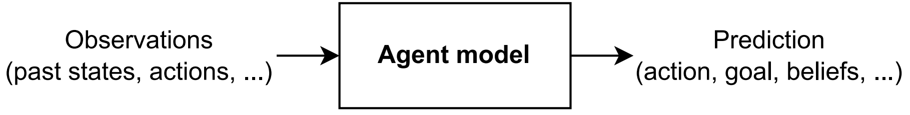
- Distribution of training data is dependent on the policies of all agents
- Training centralized critics conditioned on the actions of other agents

Can we provide agents with more **explicit** information about the policies of other agents so they can learn to coordinate better, e.g. by learning best-response policies?

Reminder

In Chapter 6, we have seen approaches that model other agents' policies:

- Learn models of other agents to predict their actions
- Compute optimal action (best-response) against agent models



S. Albrecht, P. Stone. **Autonomous Agents Modelling Other Agents: A Comprehensive Survey and Open Problems.** *Artificial Intelligence*, 2018

Recap: Tabular Agent Modeling

In Chapter 6, we modeled other agents' policies as stationary distributions by maintaining tables of action frequencies for each state

Recap: Tabular Agent Modeling

In Chapter 6, we modeled other agents' policies as stationary distributions by maintaining tables of action frequencies for each state

Problem

Similar to tabular value functions, **tabular agent models** are limited due to their inability to generalise across states.

Recap: Tabular Agent Modeling

In Chapter 6, we modeled other agents' policies as stationary distributions by maintaining tables of action frequencies for each state

Problem

Similar to tabular value functions, **tabular agent models** are limited due to their inability to generalise across states.

Solution

Use **deep learning** to learn generalisable agent models!

Recap: Joint-Action Learning with Agent Models

Reminder

We have already seen joint-action learning with agent models (JAL-AM)

Recap: Joint-Action Learning with Agent Models

Reminder

We have already seen joint-action learning with agent models (JAL-AM)

- Agents learn models of other agents conditioned on states s: $\hat{\pi}_{-i}(a_{-i} \mid s)$

Recap: Joint-Action Learning with Agent Models

Reminder

We have already seen joint-action learning with agent models (JAL-AM)

- Agents learn models of other agents conditioned on states s: $\hat{\pi}_{-i}(a_{-i} \mid s)$
- Agents learn value functions conditioned on the joint action: $Q_i(s, a)$

Recap: Joint-Action Learning with Agent Models

Reminder

We have already seen joint-action learning with agent models (JAL-AM)

- Agents learn models of other agents conditioned on states s: $\hat{\pi}_{-i}(a_{-i} \mid s)$
- Agents learn value functions conditioned on the joint action: $Q_i(s, a)$
- Using the value function and agent models, agent i can compute its expected action values under the current models of other agents:

$$AV_i(s, a_i) = \sum_{a_{-i} \in A_{-i}} Q_i(s, \langle a_i, a_{-i} \rangle) \prod_{j \in I \setminus \{i\}} \hat{\pi}_j(a_j \mid s)$$

Recap: Joint-Action Learning with Agent Models

Reminder

We have already seen joint-action learning with agent models (JAL-AM)

- Agents learn models of other agents conditioned on states s: $\hat{\pi}_{-i}(a_{-i} \mid s)$
- Agents learn value functions conditioned on the joint action: $Q_i(s, a)$
- Using the value function and agent models, agent i can compute its expected action values under the current models of other agents:

$$AV_i(s, a_i) = \sum_{a_{-i} \in A_{-i}} Q_i(s, \langle a_i, a_{-i} \rangle) \prod_{j \in I \setminus \{i\}} \hat{\pi}_j(a_j \mid s)$$

- Use AV_i to select optimal actions and as learning update targets

Joint-Action Learning with Deep Agent Models

How to do this with deep learning for partially observable environments (POSG)?

Joint-Action Learning with Deep Agent Models

How to do this with deep learning for partially observable environments (POSG)?

- Represent agent i's model of agent j as a neural network: $\hat{\pi}_j^i(a_j \mid h_i; \phi_j^i)$
- Given the observation history of agent i, its model $\hat{\pi}_j^i(a_j \mid h_i; \phi_j^i)$ for agent j outputs a probability distribution over actions

Joint-Action Learning with Deep Agent Models

How to do this with deep learning for partially observable environments (POSG)?

- Represent agent i's model of agent j as a neural network: $\hat{\pi}^i_j(a_j \mid h_i; \phi^i_j)$
- Given the observation history of agent i, its model $\hat{\pi}^i_j(a_j \mid h_i; \phi^i_j)$ for agent j outputs a probability distribution over actions
- Agent i can train its model for agent j by minimizing the cross-entropy loss between the predicted policy $\hat{\pi}^i_j$ and the observed actions of agent j:

$$\mathcal{L}(\phi^i_j) = \mathbb{E}_{a^t_j \sim \pi_j(h^t_j)}\left[-\log \hat{\pi}^i_j(a^t_j \mid h^t_i; \phi^i_j)\right]$$

Joint-Action Learning with Deep Agent Models

How to do this with deep learning for partially observable environments (POSG)?

- Represent agent i's model of agent j as a neural network: $\hat{\pi}^i_j(a_j \mid h_i; \phi^i_j)$
- Given the observation history of agent i, its model $\hat{\pi}^i_j(a_j \mid h_i; \phi^i_j)$ for agent j outputs a probability distribution over actions
- Agent i can train its model for agent j by minimizing the cross-entropy loss between the predicted policy $\hat{\pi}^i_j$ and the observed actions of agent j:

$$\mathcal{L}(\phi^i_j) = \mathbb{E}_{a^t_j \sim \pi_j(h^t_j)}\left[-\log \hat{\pi}^i_j(a^t_j \mid h^t_i; \phi^i_j)\right]$$

- Then, agent i can compute expected action values:

$$AV(h_i, a_i; \theta_i) = \sum_{a_{-i} \in A_{-i}} Q(h_i, \langle a_i, a_{-i} \rangle; \theta_i) \prod_{j \neq i} \hat{\pi}^i_j(a_j \mid h_i; \phi^i_j)$$

Problem

To compute AV, we need to sum over all possible joint actions of other agents which is infeasible for environments with large action spaces or many agents.

Joint-Action Learning with Deep Agent Models

Problem

To compute *AV*, we need to sum over all possible joint actions of other agents which is infeasible for environments with large action spaces or many agents.

Solution

Approximate *AV* with only *K* joint actions sampled from the agent models:

$$AV(h_i, a_i; \theta_i) = \frac{1}{K} \sum_{k=1}^{K} Q(h_i, \langle a_i, a_{-i}^k \rangle; \theta_i) \Big|_{a_j^k \sim \hat{\pi}_j^i(\cdot | h_i)}$$

Joint-Action Learning with Deep Agent Models

Problem

To compute *AV*, we need to sum over all possible joint actions of other agents which is infeasible for environments with large action spaces or many agents.

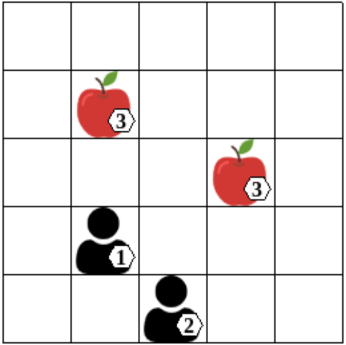
Solution

Approximate *AV* with only *K* joint actions sampled from the agent models:

$$AV(h_i, a_i; \theta_i) = \frac{1}{K} \sum_{k=1}^{K} Q(h_i, \langle a_i, a_{-i}^k \rangle; \theta_i) \Big|_{a_j^k \sim \hat{\pi}_j^i(\cdot | h_i)}$$

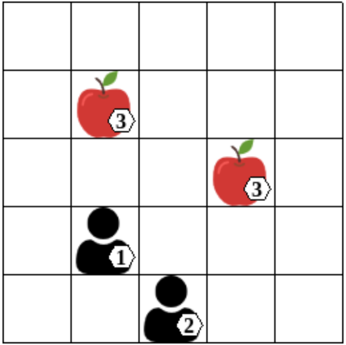
To optimise the centralized joint-action-value function of agent *i*, we then minimize the following loss over batches of experiences sampled from a replay buffer:

Joint-Action Learning with Deep Agent Models in LBF

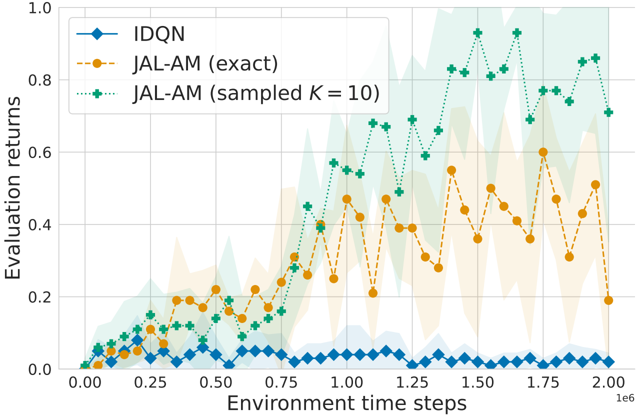


(a) Environment

Joint-Action Learning with Deep Agent Models in LBF



(a) Environment



(b) Learning curve

Learning Compact Representations of Agent Policies

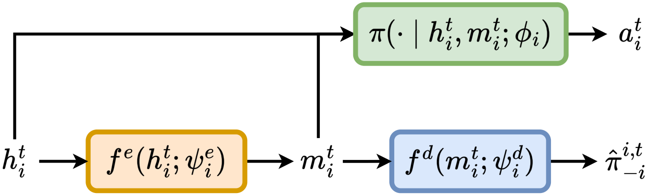
- JAL-AM combines agent models and centralized value functions to compute best-response policies.
- Can we integrate agent models into multi-agent policy gradient algorithms, e.g. by conditioning policies on agent models?

- JAL-AM combines agent models and centralized value functions to compute best-response policies.
- Can we integrate agent models into multi-agent policy gradient algorithms, e.g. by conditioning policies on agent models?

Problem

To condition policies (and value functions) of agents on the policies of other agents, we need **compact** representations of the policies of other agents. How can we learn such representations?

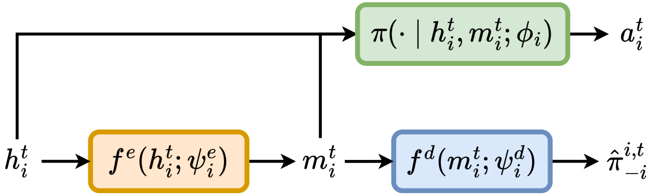
Learning Compact Representations of Agent Policies



Agent i trains encoder-decoder architecture with

- Encoder f^e with parameters ψ_i^e: given observation history h_i^t of agent i, output compact representation m_i^t of the policies of other agents
- Decoder f^d with parameters ψ_i^d: given compact representation m_i^t, predict the policies $\hat{\pi}_{-i}^{i,t}$ of other agents

Learning Compact Representations of Agent Policies



Agent i trains encoder-decoder architecture with

- Encoder f^e with parameters ψ_i^e: given observation history h_i^t of agent i, output compact representation m_i^t of the policies of other agents
- Decoder f^d with parameters ψ_i^d: given compact representation m_i^t, predict the policies $\hat{\pi}_{-i}^{i,t}$ of other agents

Then, agent i can condition its policy on the compact representations m_i^t.

Learning Compact Representations of Agent Policies

The encoder and decoder are jointly trained to minimize the cross-entropy loss for the predicted action probabilities and true actions of all other agents:

$$\mathcal{L}(\psi_i^e, \psi_i^d) = \sum_{j \neq i} - \log \hat{\pi}_j^{i,t}(a_j^t) \quad \text{with} \quad \hat{\pi}_j^{i,t} = f^d \left(f^e(h_i^t; \psi_i^e); \psi_i^d \right)_j$$

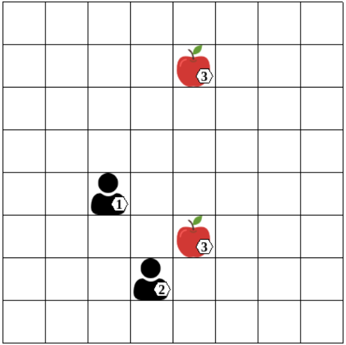
Learning Compact Representations of Agent Policies

The encoder and decoder are jointly trained to minimize the cross-entropy loss for the predicted action probabilities and true actions of all other agents:

$$\mathcal{L}(\psi_i^e, \psi_i^d) = \sum_{j \neq i} - \log \hat{\pi}_j^{i,t}(a_j^t) \quad \text{with} \quad \hat{\pi}_j^{i,t} = f^d\left(f^e(h_i^t; \psi_i^e); \psi_i^d\right)_j$$

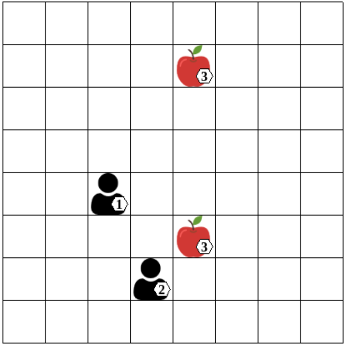
Encoder-decoder agent models can be integrated into MARL algorithm by conditioning value functions and policies on the obtained policy representations.

Compact Agent Policy Representations in LBF

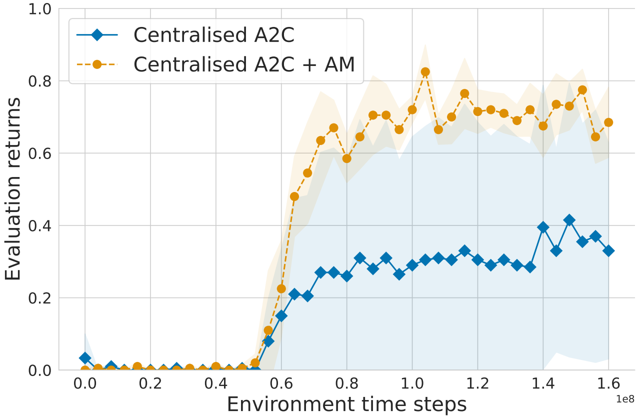


(a) Environment

Compact Agent Policy Representations in LBF

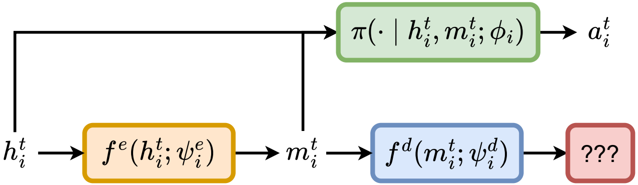


(a) Environment



(b) Learning curve

Reconstruction Targets to Learn Compact Representations



Note

We used the ground truth actions as information to encode by using them as targets for the decoder. We could also use

- Observations – try to capture information that other agents have access to
- Rewards – try to predict the objectives that other agents optimise for
- ...

Parameter and Experience Sharing

Parameter and Experience Sharing – Motivation

Problem

Training agents with MARL is difficult for environments with many agents due to the increased number of parameters to train $\Rightarrow$ unstable or slow training!

Parameter and Experience Sharing – Motivation

Problem

Training agents with MARL is difficult for environments with many agents due to the increased number of parameters to train $\Rightarrow$ unstable or slow training!

Solution

Two approaches to improve the efficiency of training many agents:

- **Parameter sharing**: agents share their network parameters with each other
- **Experience sharing**: agents share experiences with each other

Environments with Homogeneous Agents

- Parameter and experience sharing assume that agents are trying to learn similar or identical policies $\rightarrow$ not applicable to all environments
- We call agents in such environments homogeneous

Environments with Homogeneous Agents

- Parameter and experience sharing assume that agents are trying to learn similar or identical policies $\rightarrow$ not applicable to all environments
- We call agents in such environments homogeneous
  - Strongly homogeneous agents: All agents have the same optimal policy, i.e.

$$\pi_1^* = \ldots = \pi_n^*$$

Environments with Homogeneous Agents

- Parameter and experience sharing assume that agents are trying to learn similar or identical policies $\rightarrow$ not applicable to all environments
- We call agents in such environments homogeneous
  - Strongly homogeneous agents: All agents have the same optimal policy, i.e.

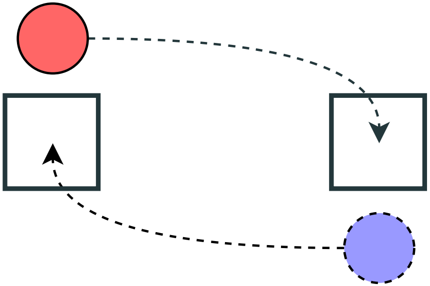
$$\pi_1^* = \ldots = \pi_n^*$$

  - Weakly homogeneous agents: Agents can be permuted and their expected returns remain the same under the permutation $\sigma : I \mapsto I$:

$$U_i(\pi) = U_{\sigma(i)}\left(\langle \pi_{\sigma(1)}, \pi_{\sigma(2)}, \ldots, \pi_{\sigma(n)} \rangle\right), \ \forall i \in I$$

Environments with Homogeneous Agents – Examples

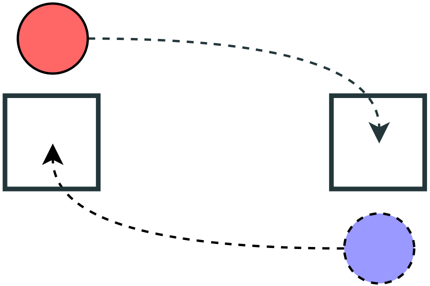
Weakly homogeneous agents:



Agents need to learn similar policies.

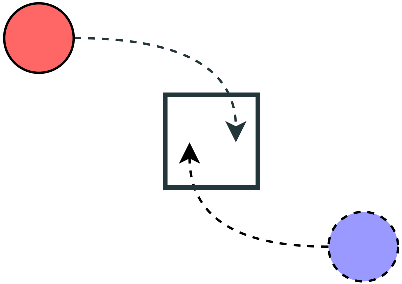
Environments with Homogeneous Agents – Examples

Weakly homogeneous agents:



Agents need to learn similar policies.

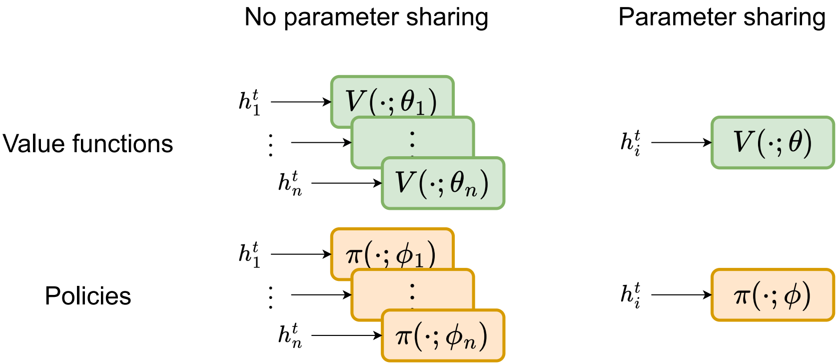
Strongly homogeneous agents:



Agents need to learn identical policies.

Parameter Sharing

Sharing network parameters across agents is common practice to make MARL training more efficient. Share parameters across value functions, policies, or both.



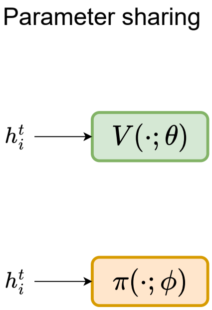
Parameter Sharing

Sharing network parameters across agents is common practice to make MARL training more efficient. Share parameters across value functions, policies, or both.

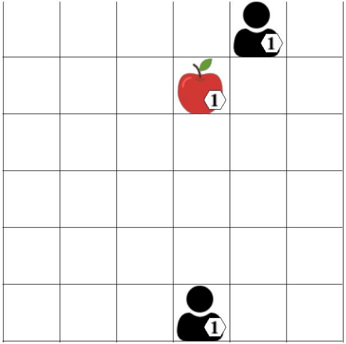
Parameter sharing has two primary benefits:

- **Scalability**: the number of parameters remains constant independent of the number of agents $\rightarrow$ less computational cost
- **Efficiency**: shared parameters are updated using the experiences of all agents $\rightarrow$ more training data for the shared parameters

The downside is that (naive) parameter sharing assumes strongly homogeneous agents.

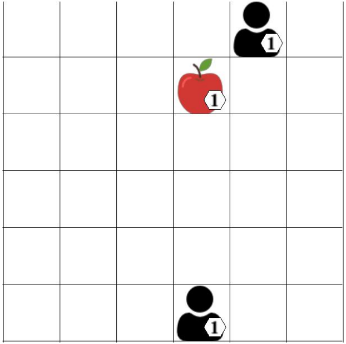


Parameter Sharing in LBF

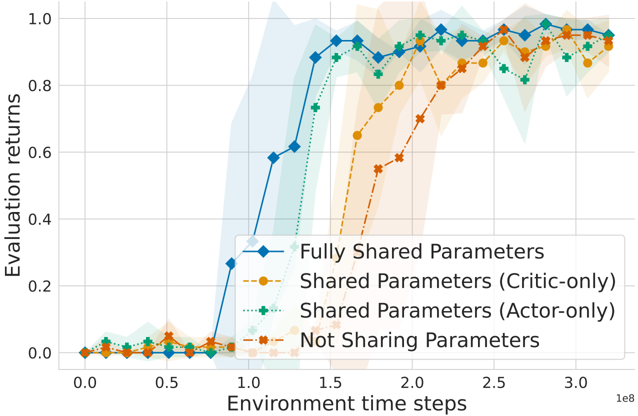


(a) Environment

Parameter Sharing in LBF



(a) Environment



(b) Learning curve

Experience Sharing for Weakly Homogeneous Agents

Under experience sharing, agents still learn separate policies (and value functions) but share their trajectories to allow training on the collective data.

Experience Sharing for Weakly Homogeneous Agents

Under experience sharing, agents still learn separate policies (and value functions) but share their trajectories to allow training on the collective data.

Assumption: Agents are (at least) weakly homogeneous, i.e. they need to learn similar policies.

Experience Sharing for Weakly Homogeneous Agents

Under experience sharing, agents still learn separate policies (and value functions) but share their trajectories to allow training on the collective data.

Assumption: Agents are (at least) weakly homogeneous, i.e. they need to learn similar policies.

Note

The experiences of agent j is **off-policy** data for agent i $\rightarrow$ experience sharing needs to use off-policy MARL algorithms or correct for the differences in data distributions.

Deep Q-Networks with Shared Experience Replay

We can extend IDQN with experience sharing by following the steps below:

- Collect the experience of all agents in a shared replay buffer $\mathcal{D}_{\text{shared}}$
- Each agent samples from $\mathcal{D}_{\text{shared}}$ to update its value function
- Each agent optimises its value function using the typical IDQN loss

Deep Q-Networks with Shared Experience Replay

We can extend IDQN with experience sharing by following the steps below:

- Collect the experience of all agents in a shared replay buffer $\mathcal{D}_{\text{shared}}$
- Each agent samples from $\mathcal{D}_{\text{shared}}$ to update its value function
- Each agent optimises its value function using the typical IDQN loss

Note

DQN is an off-policy algorithm so it is theoretically sound to use the experience of other agents that have different policies.

Shared Experience Actor-Critic (SEAC)

We can also extend independent actor-critic algorithms (e.g. IA2C and IPPO) with experience sharing by optimising the loss over the data of all agents.

Shared Experience Actor-Critic (SEAC)

We can also extend independent actor-critic algorithms (e.g. IA2C and IPPO) with experience sharing by optimising the loss over the data of all agents.

SEAC policy loss (based on IA2C) with importance sampling (IS) weight correction:

$$\mathcal{L}(\phi_i) = \overbrace{-\left(r_i^t + \gamma V(h_i^{t+1};\theta_i) - V(h_i^t;\theta_i)\right)\log\pi(a_i^t \mid h_i^t;\phi_i)}^{\text{policy loss on own data}}$$

Shared Experience Actor-Critic (SEAC)

We can also extend independent actor-critic algorithms (e.g. IA2C and IPPO) with experience sharing by optimising the loss over the data of all agents.

SEAC policy loss (based on IA2C) with importance sampling (IS) weight correction:

$$\mathcal{L}(\phi_i) = \overbrace{-\left(r_i^t + \gamma V(h_i^{t+1};\theta_i) - V(h_i^t;\theta_i)\right)\log \pi(a_i^t \mid h_i^t;\phi_i)}^{\text{policy loss on own data}}$$
$$- \lambda \sum_{k \neq i} \underbrace{\frac{\pi(a_k^t \mid h_k^t;\phi_i)}{\pi(a_k^t \mid h_k^t;\phi_k)}}_{\text{IS correction}} \underbrace{\left(r_k^t + \gamma V(h_k^{t+1};\theta_i) - V(h_k^t;\theta_i)\right)\log \pi(a_k^t \mid h_k^t;\phi_i)}_{\text{policy loss on data of agent } k}$$

Shared Experience Actor-Critic (SEAC)

We can also extend independent actor-critic algorithms (e.g. IA2C and IPPO) with experience sharing by optimising the loss over the data of all agents.

SEAC policy loss (based on IA2C) with importance sampling (IS) weight correction:

$$\mathcal{L}(\phi_i) = \overbrace{-\left(r_i^t + \gamma V(h_i^{t+1};\theta_i) - V(h_i^t;\theta_i)\right) \log \pi(a_i^t \mid h_i^t;\phi_i)}^{\text{policy loss on own data}}$$

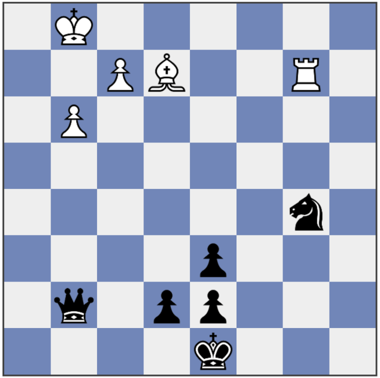
$$- \lambda \sum_{k \neq i} \underbrace{\frac{\pi(a_k^t \mid h_k^t;\phi_i)}{\pi(a_k^t \mid h_k^t;\phi_k)}}_{\text{IS correction}} \underbrace{\left(r_k^t + \gamma V(h_k^{t+1};\theta_i) - V(h_k^t;\theta_i)\right) \log \pi(a_k^t \mid h_k^t;\phi_i)}_{\text{policy loss on data of agent } k}$$

Hyperparameter λ determines weighting for the loss over the experience of other agents. The same IS weight correction can be applied to the critic loss.

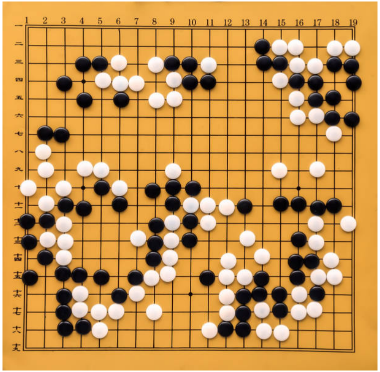
Policy Self-Play in Zero-Sum Games

The Challenge of Zero-Sum Board Games

Next we will take a closer look at (turn-based) zero-sum board games such as chess, shogi, or Go.



(a) Chess



(b) Go

The Challenge of Zero-Sum Board Games

Next we will take a closer look at (turn-based) zero-sum board games such as chess, shogi, or Go.

These games are challenging decision-making problems due to

- **Sparse rewards:** agents only receive a reward once the game terminates (+1 for winning, -1 for losing, 0 for a draw)

The Challenge of Zero-Sum Board Games

Next we will take a closer look at (turn-based) zero-sum board games such as chess, shogi, or Go.

These games are challenging decision-making problems due to

- **Sparse rewards:** agents only receive a reward once the game terminates (+1 for winning, -1 for losing, 0 for a draw)
- **Large action space:** in many board games, agents can choose from a large action space (e.g. moving a particular piece to a particular position)

The Challenge of Zero-Sum Board Games

Next we will take a closer look at (turn-based) zero-sum board games such as chess, shogi, or Go.

These games are challenging decision-making problems due to

- **Sparse rewards:** agents only receive a reward once the game terminates (+1 for winning, -1 for losing, 0 for a draw)
- **Large action space:** in many board games, agents can choose from a large action space (e.g. moving a particular piece to a particular position)
- **Long horizon:** episodes can last for many time steps before reaching a terminal state

The Challenge of Zero-Sum Board Games

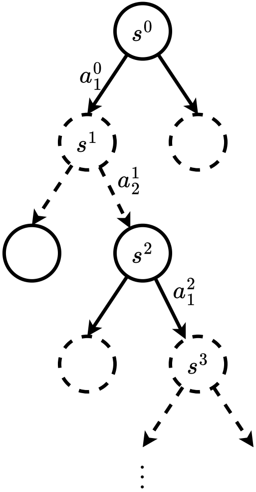
Next we will take a closer look at (turn-based) zero-sum board games such as chess, shogi, or Go.

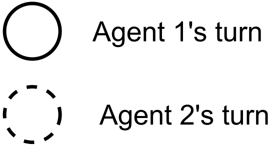
These games are challenging decision-making problems due to

- **Sparse rewards:** agents only receive a reward once the game terminates (+1 for winning, -1 for losing, 0 for a draw)
- **Large action space:** in many board games, agents can choose from a large action space (e.g. moving a particular piece to a particular position)
- **Long horizon:** episodes can last for many time steps before reaching a terminal state

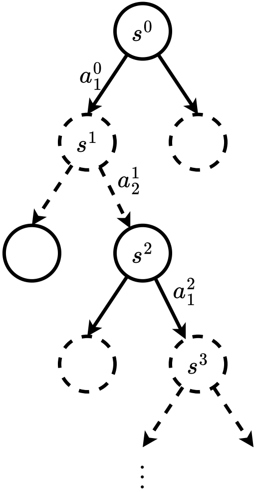
Fortunately, we can exploit the structure of these games to develop effective algorithms.

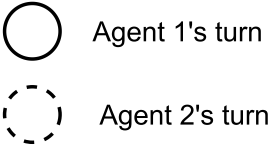
Tree Search for Zero-Sum Games





Tree Search for Zero-Sum Games



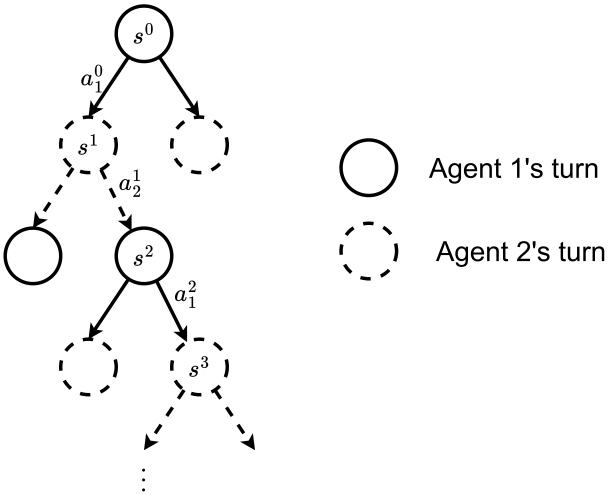


We can view turn-based zero-sum games as trees where

- *nodes* represent game states
- *edges* represent actions
- *leaves* represent terminal states

in each node either agent 1 or agent 2 makes a move.

Tree Search for Zero-Sum Games



Problem

The tree can grow very large depending on its

- **Depth**: number of time steps until terminal states
- **Breadth**: number of actions available in each state

$\rightarrow$ makes search computationally expensive

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that simulates (samples) possible outcomes of the game, and then updating value estimates of actions.

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that simulates (samples) possible outcomes of the game, and then updating value estimates of actions.

MCTS consists of three key steps that are repeated:

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that simulates (samples) possible outcomes of the game, and then updating value estimates of actions.

MCTS consists of three key steps that are repeated:

- **Simulation:** simulate a game from the current state until a leaf node is reached

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that simulates (samples) possible outcomes of the game, and then updating value estimates of actions.

MCTS consists of three key steps that are repeated:

- **Simulation:** simulate a game from the current state until a leaf node is reached
- **Expansion:** expand the tree by adding a new node for the reached state if it does not exist yet

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a tree-search algorithm that simulates (samples) possible outcomes of the game, and then updating value estimates of actions.

MCTS consists of three key steps that are repeated:

- **Simulation:** simulate a game from the current state until a leaf node is reached
- **Expansion:** expand the tree by adding a new node for the reached state if it does not exist yet
- **Backpropagation:** update the estimated values of the nodes visited during the selection step

Monte Carlo Tree Search – Simulation

MCTS maintains two statistics for each visited state-action pair:

- Value estimates $Q(s, a)$
- Visitation counts $N(s, a)$

Monte Carlo Tree Search – Simulation

MCTS maintains two statistics for each visited state-action pair:

- Value estimates $Q(s, a)$
- Visitation counts $N(s, a)$

MCTS generates k simulations from the current state by sampling actions according to a policy until a leaf node is reached. Leaf nodes are previously unvisited or terminal states.

Monte Carlo Tree Search – Simulation

MCTS maintains two statistics for each visited state-action pair:

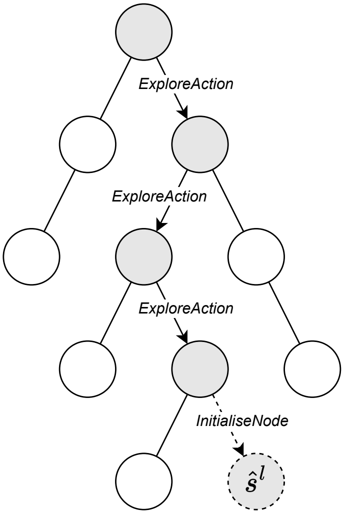
- Value estimates $Q(s, a)$
- Visitation counts $N(s, a)$

MCTS generates k simulations from the current state by sampling actions according to a policy until a leaf node is reached. Leaf nodes are previously unvisited or terminal states.

To sample actions, MCTS commonly uses ϵ-greedy policies with respect to action-value function Q, or the upper confidence bound (UCB) policy:

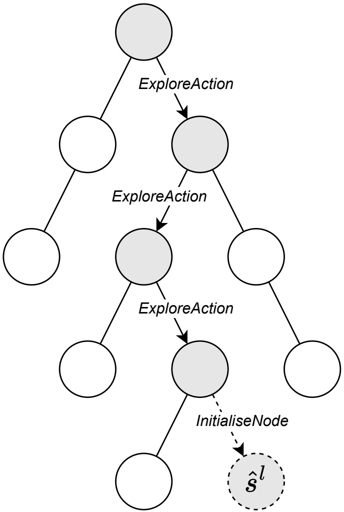
$$\hat{a}^\tau = \begin{cases} \hat{a} & \text{if } N(\hat{s}^\tau, \hat{a}) = 0 \\ \arg\max_{\hat{a} \in A} \left(Q(\hat{s}^\tau, \hat{a}) + \sqrt{\frac{2 \ln N(\hat{s}^\tau)}{N(\hat{s}^\tau, \hat{a})}} \right) & \text{otherwise} \end{cases}$$

Monte Carlo Tree Search – Expansion



If leaf node is reached $\rightarrow$ expand the search tree by adding a new node for the reached state $\hat{s}^l$

Monte Carlo Tree Search – Expansion

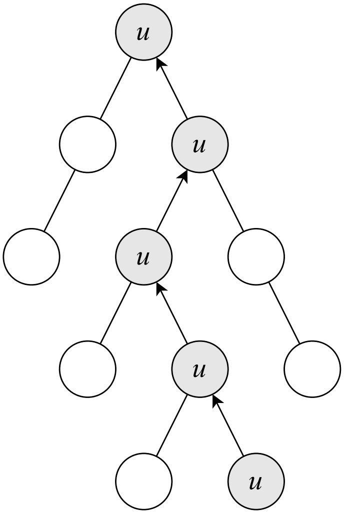


If leaf node is reached $\rightarrow$ expand the search tree by adding a new node for the reached state $\hat{s}^l$

The new node is initialized with

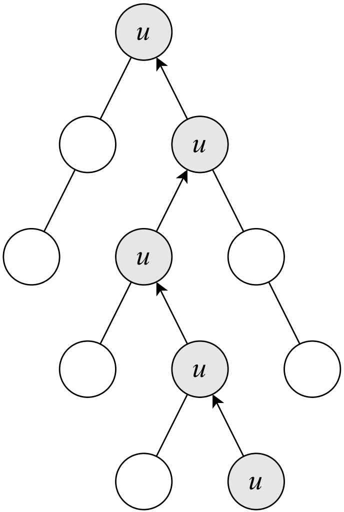
- $N(\hat{s}^l, \hat{a}) = 0$ for all actions $\hat{a}$
- An initial value estimate $Q(\hat{s}^l, \hat{a})$ for all actions $\hat{a}$, e.g. from a learned value function, heuristic, or random samples of outcomes.

Monte Carlo Tree Search – Backpropagation



Once a value estimate u for the leaf node is obtained $\rightarrow$ backpropagate rewards and value estimates starting from the leaf node up to the root node.

Monte Carlo Tree Search – Backpropagation

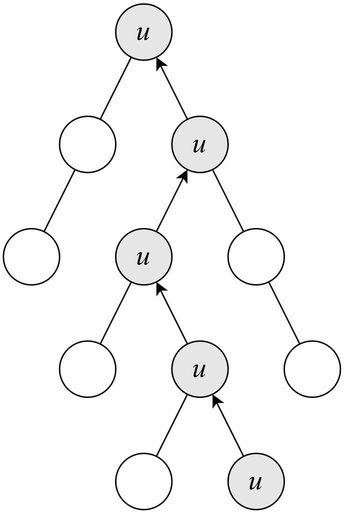


Once a value estimate u for the leaf node is obtained $\rightarrow$ backpropagate rewards and value estimates starting from the leaf node up to the root node.

For each visited state-action pair $(\hat{s}^\tau, \hat{a}^\tau)$, we increment the visitation count and update the value:

$$Q(\hat{s}^\tau, \hat{a}^\tau) \leftarrow Q(\hat{s}^\tau, \hat{a}^\tau) + \frac{1}{N(\hat{s}^\tau, \hat{a}^\tau)} \left[u - Q(\hat{s}^\tau, \hat{a}^\tau)\right]$$

Monte Carlo Tree Search – Backpropagation



Once a value estimate u for the leaf node is obtained $\rightarrow$ backpropagate rewards and value estimates starting from the leaf node up to the root node.

For each visited state-action pair $(\hat{s}^\tau, \hat{a}^\tau)$, we increment the visitation count and update the value:

$$Q(\hat{s}^\tau, \hat{a}^\tau) \leftarrow Q(\hat{s}^\tau, \hat{a}^\tau) + \frac{1}{N(\hat{s}^\tau, \hat{a}^\tau)} [u - Q(\hat{s}^\tau, \hat{a}^\tau)]$$

(Any RL TD update rule can be used to update the value estimates.)

Monte Carlo Tree Search – Action Selection

Simulation, expansion, and backpropagation steps are repeated at every time step to iteratively grow the search tree and obtain better value estimates.

Monte Carlo Tree Search – Action Selection

Simulation, expansion, and backpropagation steps are repeated at every time step to iteratively grow the search tree and obtain better value estimates.

Following k simulations form the current state s^t, the best action is selected. This process can be done by choosing the action with:

- highest value estimate: $BestAction(s^t) = \arg\max_{\hat{a} \in A} Q(s^t, \hat{a})$
- highest visitation count: $BestAction(s^t) = \arg\max_{\hat{a} \in A} N(s^t, \hat{a})$

Monte Carlo Tree Search – Pseudocode

Algorithm Monte Carlo tree search (MCTS) for MDPs

Repeat for every episode:
for $t = 0, 1, 2, 3, \ldots$ **do**
  Observe current state s^t
  for k simulations **do**
    $\tau \leftarrow t$
    $\hat{s}^\tau \leftarrow s^t$ ▷ Perform simulation
    while $\hat{s}^\tau$ is non-terminal and $\hat{s}^\tau$-node exists in tree **do**
      $\hat{a}^\tau \leftarrow \mathit{ExploreAction}(\hat{s}^\tau)$
      $\hat{s}^{\tau+1} \sim \mathcal{T}(\cdot \mid \hat{s}^\tau, \hat{a}^\tau)$
      $\hat{r}^\tau \leftarrow \mathcal{R}(\hat{s}^\tau, \hat{a}^\tau, \hat{s}^{\tau+1})$
      $\tau \leftarrow \tau + 1$
    if $\hat{s}^\tau$-node does not exist in tree **then**
      $\mathit{InitializeNode}(\hat{s}^\tau)$ ▷ Expand tree
    while $\tau > t$ **do** ▷ Backpropagate
      $\tau \leftarrow \tau - 1$
      $\mathit{Update}(Q, \hat{s}^\tau, \hat{a}^\tau)$
  Select action a^t for state s^t:
    $\pi^t \leftarrow \mathit{BestAction}(s^t)$
    $a^t \sim \pi^t$

Monte Carlo Tree Search – Pseudocode

Algorithm Monte Carlo tree search (MCTS) for MDPs

Repeat for every episode:
for $t = 0, 1, 2, 3, \ldots$ **do**
    Observe current state s^t
    for k simulations **do**
        $\tau \leftarrow t$
        $\hat{s}^\tau \leftarrow s^t$ ▷ Perform simulation
        while $\hat{s}^\tau$ is non-terminal and $\hat{s}^\tau$-node exists in tree **do**
            $\hat{a}^\tau \leftarrow \mathit{ExploreAction}(\hat{s}^\tau)$
            $\hat{s}^{\tau+1} \sim \mathcal{T}(\cdot \mid \hat{s}^\tau, \hat{a}^\tau)$
            $\hat{r}^\tau \leftarrow \mathcal{R}(\hat{s}^\tau, \hat{a}^\tau, \hat{s}^{\tau+1})$
            $\tau \leftarrow \tau + 1$
        if $\hat{s}^\tau$-node does not exist in tree **then**
            $\mathit{InitializeNode}(\hat{s}^\tau)$ ▷ Expand tree
        while $\tau > t$ **do** ▷ Backpropagate
            $\tau \leftarrow \tau - 1$
            $\mathit{Update}(Q, \hat{s}^\tau, \hat{a}^\tau)$
    Select action a^t for state s^t:
        $\pi^t \leftarrow \mathit{BestAction}(s^t)$
        $a^t \sim \pi^t$

Note

MCTS assumes known transition function $\mathcal{T}$ and reward function $\mathcal{R}$.

Monte Carlo Tree Search – Pseudocode

Algorithm Monte Carlo tree search (MCTS) for MDPs

Repeat for every episode:
for $t = 0, 1, 2, 3, \ldots$ **do**
    Observe current state s^t
    for k simulations **do**
        $\tau \leftarrow t$
        $\hat{s}^\tau \leftarrow s^t$ ▷ Perform simulation
        while $\hat{s}^\tau$ is non-terminal and $\hat{s}^\tau$-node exists in tree **do**
            $\hat{a}^\tau \leftarrow \mathit{ExploreAction}(\hat{s}^\tau)$
            $\hat{s}^{\tau+1} \sim \mathcal{T}(\cdot \mid \hat{s}^\tau, \hat{a}^\tau)$
            $\hat{r}^\tau \leftarrow \mathcal{R}(\hat{s}^\tau, \hat{a}^\tau, \hat{s}^{\tau+1})$
            $\tau \leftarrow \tau + 1$
        if $\hat{s}^\tau$-node does not exist in tree **then**
            $\mathit{InitializeNode}(\hat{s}^\tau)$ ▷ Expand tree
        while $\tau > t$ **do** ▷ Backpropagate
            $\tau \leftarrow \tau - 1$
            $\mathit{Update}(Q, \hat{s}^\tau, \hat{a}^\tau)$
    Select action a^t for state s^t:
      $\pi^t \leftarrow \mathit{BestAction}(s^t)$
      $a^t \sim \pi^t$

Note

MCTS assumes known transition function $\mathcal{T}$ and reward function $\mathcal{R}$.

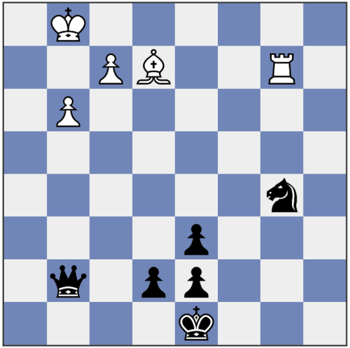
Can learn estimates of these functions from data to simulate possible outcomes of the game.

Self-Play Monte Carlo Tree Search

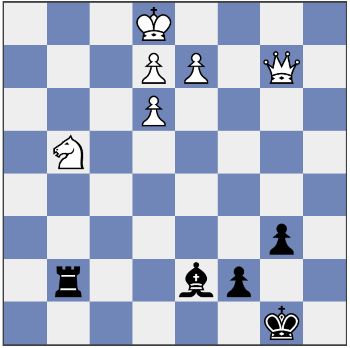
In zero-sum games with symmetrical roles and egocentric observations, agents can use the same policy to control both players

Self-Play Monte Carlo Tree Search

In zero-sum games with symmetrical roles and egocentric observations, agents can use the same policy to control both players → learn a policy in self-play



(a) Agent 1 perspective



(b) Agent 2 perspective

AlphaZero – Self-Play MCTS with Deep Learning

AlphaZero: combine self-play MCTS with deep learning to learn value estimates and policies $\rightarrow$ reached superhuman performance in Go, chess, and shogi!

AlphaZero – Self-Play MCTS with Deep Learning

AlphaZero: combine self-play MCTS with deep learning to learn value estimates and policies $\rightarrow$ reached superhuman performance in Go, chess, and shogi!

Learn functions with parameters θ conditioned on state s:

- Value estimate $V(s;\theta)$
- Policy $\pi(\cdot \mid s;\theta)$

AlphaZero – Self-Play MCTS with Deep Learning

AlphaZero: combine self-play MCTS with deep learning to learn value estimates and policies $\rightarrow$ reached superhuman performance in Go, chess, and shogi!

Learn functions with parameters θ conditioned on state s:

- Value estimate $V(s; \theta)$
- Policy $\pi(\cdot \mid s; \theta)$

For each episode, a triplet (s, π, z) of data is stored where

- s are the states
- π are policy distributions computed by *BestAction*
- z is the game outcome (+1 for win, -1 for loss, 0 for draw)

AlphaZero – Self-Play MCTS with Deep Learning

The network is randomly initialized and trained using sampled batches of data to minimise the following combined loss:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{value}} + \mathcal{L}_{\text{policy}} + c\,||\theta||^2$$

$$\mathcal{L}_{\text{value}} = \mathbb{E}_{(s,\pi,z)\sim\mathcal{D}}\left[(V(s;\theta) - u)^2\right]$$

$$\mathcal{L}_{\text{policy}} = \mathbb{E}_{(s,\pi,z)\sim\mathcal{D}}\left[\pi^\top \log \pi(\cdot \mid s;\theta)\right]$$

AlphaZero – Self-Play MCTS with Deep Learning

The network is randomly initialized and trained using sampled batches of data to minimise the following combined loss:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{value}} + \mathcal{L}_{\text{policy}} + c\,||\theta||^2$$

$$\mathcal{L}_{\text{value}} = \mathbb{E}_{(s,\pi,z)\sim\mathcal{D}}\left[\left(V(s;\theta) - u\right)^2\right]$$

$$\mathcal{L}_{\text{policy}} = \mathbb{E}_{(s,\pi,z)\sim\mathcal{D}}\left[\pi^\top \log \pi(\cdot \mid s;\theta)\right]$$

For exploration, AlphaZero combines a UCB policy with the learned policy:

$$\hat{a}^\tau = \begin{cases} \hat{a} & \text{if } N(\hat{s}^\tau, \hat{a}) = 0 \\ \arg\max_{\hat{a}\in A}\left(Q(\hat{s}^\tau, \hat{a}) + C(\hat{s}^\tau)P(\hat{s}^\tau, \hat{a})\frac{\sqrt{N(\hat{s}^\tau)}}{1+N(\hat{s}^\tau,\hat{a})}\right) & \text{otherwise} \end{cases}$$

with the additional exploration rate $C(\hat{s}^\tau)$.

Population-Based Training

Population-Based Training – Self-Play for General-Sum Games

Problem

With MCTS, we focused on policy self-play in **two-agent zero-sum** games. Can we extend the idea of self-play to **general-sum** games with **more than two agents**?

Population-Based Training – Self-Play for General-Sum Games

Problem

With MCTS, we focused on policy self-play in **two-agent zero-sum** games. Can we extend the idea of self-play to **general-sum** games with **more than two agents**?

Population-based training is a generalisation of self-play to general-sum games:

- Maintain a **population of policies** representing possible strategies of the agent
- Evolve populations so they become more effective against the populations of other agents
- We denote the population of policies for agent i at generation k as Π_i^k.

Population-Based Training – Overview

First, initialize a population of policies for each agent (e.g. random policies).

For each generation, population-based training then follows two steps:

First, initialize a population of policies for each agent (e.g. random policies).

For each generation, population-based training then follows two steps:

- **Evaluation:** evaluate the performance of the policies in the population by playing games against policies of the current populations of all other agents

Population-Based Training – Overview

First, initialize a population of policies for each agent (e.g. random policies).

For each generation, population-based training then follows two steps:

- **Evaluation:** evaluate the performance of the policies in the population by playing games against policies of the current populations of all other agents
- **Evolution:** based on evaluation results, evolve the populations of all agents. This can be done by selecting a subset of high-performing policies, mutating existing policies, or adding new policies to the population.

Population-Based Training – Overview

First, initialize a population of policies for each agent (e.g. random policies).

For each generation, population-based training then follows two steps:

- **Evaluation:** evaluate the performance of the policies in the population by playing games against policies of the current populations of all other agents
- **Evolution:** based on evaluation results, evolve the populations of all agents. This can be done by selecting a subset of high-performing policies, mutating existing policies, or adding new policies to the population.

This process is repeated until convergence or for a fixed number of generations.

Policy Space Response Oracles (PSRO)

Policy space response oracles (PSRO) is a population-based MARL algorithm with the following steps:

Policy Space Response Oracles (PSRO)

Policy space response oracles (PSRO) is a population-based MARL algorithm with the following steps:

1. Initialize populations of random policies for each agent

Policy Space Response Oracles (PSRO)

Policy space response oracles (PSRO) is a population-based MARL algorithm with the following steps:

1. Initialize populations of random policies for each agent
2. Construct a **meta-game** M^k at generation k from the populations of all agents as a (non-repeated) normal-form game with
   - Actions: policies of agent populations, i.e. $A_i = \Pi_i^k$
   - Rewards: returns of joint policies $\langle \pi_1, \ldots, \pi_n \rangle$, i.e. $\mathcal{R}_i(\pi_1, \ldots, \pi_n) = U_i(\pi_1, \ldots, \pi_n)$.

M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$

M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$

Problem

Need to compute the expected returns of each agent for any joint policy $\langle \pi_1, \ldots, \pi_n \rangle$ in the meta-game. How can we do this?

Policy Space Response Oracles – Construct Meta-Game

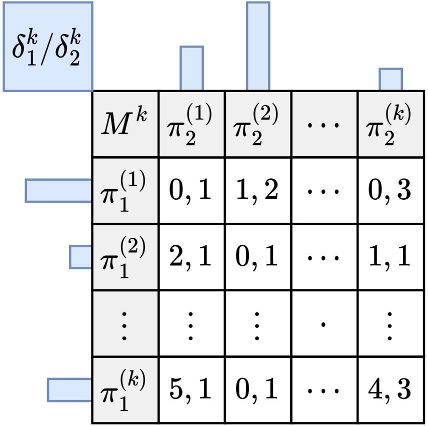
M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$

Problem

Need to compute the expected returns of each agent for any joint policy $\langle \pi_1, \ldots, \pi_n \rangle$ in the meta-game. How can we do this?

Solution

Compute average returns of each agent over multiple episodes of the underlying game with respective joint policy $\rightarrow$ converges to expected returns in the limit.



M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$

Compute a **solution to the meta-game** M^k following some solution concept, e.g. a Nash equilibrium.

Policy Space Response Oracles – Solve Meta-Game

δ_1^k / δ_2^k

M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$

Compute a **solution to the meta-game** M^k following some solution concept, e.g. a Nash equilibrium.

Solution to the meta-game: distributions δ_i^k over policies in the population of agent i (for each agent $i \in I$).

Policy Space Response Oracles – Add Best-Response Policies

M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$	π_2'
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$	?
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$	?
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$	?
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$	?
π_1'	?	?	?	?	?

Each agent i determines an effective **oracle policy** π_i' against the solution distribution of the other agents and adds this policy to its population:

$$\Pi_i^{k+1} = \Pi_i^k \cup \{\pi_i'\}$$

Policy Space Response Oracles – Add Best-Response Policies

M^k	$\pi_2^{(1)}$	$\pi_2^{(2)}$	$\cdots$	$\pi_2^{(k)}$	π'_2
$\pi_1^{(1)}$	$0, 1$	$1, 2$	$\cdots$	$0, 3$	?
$\pi_1^{(2)}$	$2, 1$	$0, 1$	$\cdots$	$1, 1$	?
$\vdots$	$\vdots$	$\vdots$	$\cdot$	$\vdots$	?
$\pi_1^{(k)}$	$5, 1$	$0, 1$	$\cdots$	$4, 3$	?
π'_1	?	?	?	?	?

Each agent i determines an effective **oracle policy** π'_i against the solution distribution of the other agents and adds this policy to its population:

$$\Pi_i^{k+1} = \Pi_i^k \cup \{\pi'_i\}$$

For example, agent i might determine its best-response policy

$$\pi'_i \in \arg\max_{\pi_i} \mathbb{E}_{\pi_{-i} \sim \delta_{-i}^k} [U_i(\langle \pi_i, \pi_{-i} \rangle)]$$

by training a policy π'_i using RL against sampled policies of the other agents.

Policy Space Response Oracles – Pseudocode

Algorithm Policy space response oracles (PSRO)

```
Initialize populations Π_i^1 for all i ∈ I (e.g., random policies)
for each generation k = 1, 2, 3, ... do
    Construct meta-game M^k from current populations {Π_i^k}_{i∈I}
    Use meta-solver on M^k to obtain distributions {δ_i^k}_{i∈I}
    for each agent i ∈ I do                    ▷ Train best-response policies
        for each episode e = 1, 2, 3, ... do
            Sample policies for other agents π_{−i} ∼ δ_{−i}^k
            Use single-agent RL to train π'_i wrt. π_{−i} in underlying game G
        Grow population Π_i^{k+1} ← Π_i^k ∪ {π'_i}
```

Policy Space Response Oracles – Pseudocode

Algorithm Policy space response oracles (PSRO)

Initialize populations Π_i^1 for all $i \in I$ (e.g., random policies)
for each generation $k = 1, 2, 3, \ldots$ **do**
  Construct meta-game M^k from current populations $\{\Pi_i^k\}_{i \in I}$
  Use meta-solver on M^k to obtain distributions $\{\delta_i^k\}_{i \in I}$
  for each agent $i \in I$ **do** ▷ Train best-response policies
    for each episode $e = 1, 2, 3, \ldots$ **do**
      Sample policies for other agents $\pi_{-i} \sim \delta_{-i}^k$
      Use single-agent RL to train π_i' wrt. π_{-i} in underlying game G
    Grow population $\Pi_i^{k+1} \leftarrow \Pi_i^k \cup \{\pi_i'\}$

Repeat this process until policy populations converge (new best-response policies are already in the respective populations), or for a fixed number of generations.

Policy Space Response Oracles in Rock-Paper-Scissors

If PSRO computes exact Nash equilibria solutions to the meta-game, and computes exact best-response policies, then the population distributions of PSRO converge to the Nash equilibrium of the underlying game.

Policy Space Response Oracles in Rock-Paper-Scissors

If PSRO computes exact Nash equilibria solutions to the meta-game, and computes exact best-response policies, then the population distributions of PSRO converge to the Nash equilibrium of the underlying game.

For example for rock-paper-scissors for two agents, with initial populations deterministically choosing rock and paper, respectively:

k	Π_1^k	Π_2^k	δ_1^k	δ_2^k	π_1'	π_2'
1	$\underline{\text{R}}$	$\underline{\text{P}}$	1	1	S	P
2	R,$\underline{\text{S}}$	P	$(0, 1)$	1	S	R
3	R,S	$\underline{\text{R}}$,P	$(\frac{2}{3}, \frac{1}{3})$	$(\frac{2}{3}, \frac{1}{3})$	P	R/P
4	R,$\underline{\text{P}}$,S	R,P	$(0, \frac{2}{3}, \frac{1}{3})$	$(\frac{1}{3}, \frac{2}{3})$	R	S
5	R,P,S	R,P,$\underline{\text{S}}$	$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	$(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$	R/P/S	R/P/S

StarCraft II is a real-time strategy game for two or more players in which players have to collect resources, build infrastructure and armies to defeat their opponents.

Figure: StarCraft II game, image source: `https://deepmind.google/discover/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii/`.

AlphaStar – GrandMaster in StarCraft II

StarCraft II is a real-time strategy game for two or more players in which players have to collect resources, build infrastructure and armies to defeat their opponents.

StarCraft II is challenging due to

- **Sparse rewards:** players only receive a terminal reward at the end of the game
- **Large action space:** players choose between many actions constituting of a type (e.g., build, move, attack), which unit should execute the action, and the target of the action
- **Long horizon:** games can last for thousands of time steps

AlphaStar – GrandMaster in StarCraft II

StarCraft II is a real-time strategy game for two or more players in which players have to collect resources, build infrastructure and armies to defeat their opponents.

StarCraft II is challenging due to

- **Sparse rewards:** players only receive a terminal reward at the end of the game
- **Large action space:** players choose between many actions constituting of a type (e.g., build, move, attack), which unit should execute the action, and the target of the action
- **Long horizon:** games can last for thousands of time steps
- **Partial observability:** players only observe a limited view of the game state
- **Diversity of strategies:** players choose between three available races offering many units and possible strategies

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

After pre-training, AlphaStar trains its policies with deep RL using a population-based training approach called **league training**: for each race, maintain a league with three types of populations, each with a different objective:

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

After pre-training, AlphaStar trains its policies with deep RL using a population-based training approach called **league training**: for each race, maintain a league with three types of populations, each with a different objective:

- **Main agents**: largely trained in self-play and against any prior policy

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

After pre-training, AlphaStar trains its policies with deep RL using a population-based training approach called **league training**: for each race, maintain a league with three types of populations, each with a different objective:

- **Main agents**: largely trained in self-play and against any prior policy
- **Exploiter agents**: largely trained against the current main agents to learn to exploit their weaknesses

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

After pre-training, AlphaStar trains its policies with deep RL using a population-based training approach called **league training**: for each race, maintain a league with three types of populations, each with a different objective:

- **Main agents**: largely trained in self-play and against any prior policy
- **Exploiter agents**: largely trained against the current main agents to learn to exploit their weaknesses
- **League exploiter agents**: trained against all agents in the league to identify effective strategies missing in the league

AlphaStar – GrandMaster in StarCraft II

AlphaStar pre-trains policies for each race using supervised imitation learning on human demonstrations from observation histories and game statistics.

After pre-training, AlphaStar trains its policies with deep RL using a population-based training approach called **league training**: for each race, maintain a league with three types of populations, each with a different objective:

- **Main agents**: largely trained in self-play and against any prior policy
- **Exploiter agents**: largely trained against the current main agents to learn to exploit their weaknesses
- **League exploiter agents**: trained against all agents in the league to identify effective strategies missing in the league

Agents of each type are added to the league whenever they become effective (measured by win rates) against their respective opponents.

During population-based training, AlphaStar computes distributions over the policies in the league to train any policy π_i' against using prioritized fictitious self-play (PFSP):

$$\delta_i^k(\pi_i) \propto f\left(\Pr[\pi_i' \text{ wins against } \pi_i]\right)$$

which is proportional to the probability of the agent winning against the policy π_i.

AlphaStar – GrandMaster in StarCraft II

During population-based training, AlphaStar computes distributions over the policies in the league to train any policy π_i' against using prioritized fictitious self-play (PFSP):

$$\delta_i^k(\pi_i) \;\propto\; f\left(\Pr[\pi_i' \text{ wins against } \pi_i]\right)$$

which is proportional to the probability of the agent winning against the policy π_i.

and $f : [0, 1] \to [0, \infty)$ is a weighting function with two components:

- f_{hard}: focus on the most difficult opponent policies for π_i'
- f_{var} focus on opponent policies that are at a similar level of performance as π_i'

AlphaStar – GrandMaster in StarCraft II

During population-based training, AlphaStar computes distributions over the policies in the league to train any policy π_i' against using prioritized fictitious self-play (PFSP):

$$\delta_i^k(\pi_i) \propto f\left(\Pr[\pi_i' \text{ wins against } \pi_i]\right)$$

which is proportional to the probability of the agent winning against the policy π_i.

and $f : [0, 1] \to [0, \infty)$ is a weighting function with two components:

- f_{hard}: focus on the most difficult opponent policies for π_i'
- f_{var} focus on opponent policies that are at a similar level of performance as π_i'

Win probabilities are estimated by running multiple matches of π_i' versus π_i.

AlphaStar – GrandMaster in StarCraft II

During population-based training, AlphaStar computes distributions over the policies in the league to train any policy π_i' against using prioritized fictitious self-play (PFSP):

$$\delta_i^k(\pi_i) \propto f\left(\Pr[\pi_i' \text{ wins against } \pi_i]\right)$$

which is proportional to the probability of the agent winning against the policy π_i.

and $f : [0,1] \to [0,\infty)$ is a weighting function with two components:

- f_{hard}: focus on the most difficult opponent policies for π_i'
- f_{var} focus on opponent policies that are at a similar level of performance as π_i'

Win probabilities are estimated by running multiple matches of π_i' versus π_i.

$\to$ reached "GrandMaster" level in StarCraft II (top 0.2% of ranked human players).

Summary

We covered:

- Agent modelling with deep learning
- Parameter and experience sharing
- Policy self-play in zero-sum games
- Population-based training