Lecture 3: Generalization, Structure, and Realism in Reinforcement Learning

Wouter van Heeswijk

University of Twente
w.j.a.vanheeswijk@utwente.nl

23 July 2025

Outline

1 Structured and Combinatorial Reinforcement Learning

2 Model-Based Reinforcement Learning

3 Multi-Agent Reinforcement Learning

Lecture 3: Structure, Models, and Realism in Reinforcement Learning

Overview:

- **Structured and Combinatorial RL:**
  - Encoding structure with **Graph Neural Networks (GNNs)**
  - **Neural Combinatorial Optimization (NCO)**: Pointer Networks, POMO
- **Model-Based RL and Planning:**
  - Learning system transitions
  - Planning via rollouts, MPC, MCTS
  - Model-based approaches for OR-style control and scheduling
- **Multi-Agent RL:**
  - Analyzing systems of intelligent agents and connections to game theory

Theme: Solving realistic, structured, and data-driven problems in RL for operations research.

Structured and Combinatorial Reinforcement Learning

Why Structured RL?

Motivation:

- OR problems often involve **structured states** (graphs, sets) and **combinatorial actions** (tours, matchings).
- Standard RL with flat state/action spaces struggles to scale or respect constraints.

Goal of this block:

- Use **GNNs** for structured representations.
- Apply RL to **combinatorial decision problems**.
- Explore **POMO**, **Pointer Nets**, and **GFlowNets** as generative methods.

Graph Neural Networks

Why Use GNNs in Reinforcement Learning?

Many RL problems are graph-structured:

- Routing, scheduling, and resource allocation naturally map to graphs.
- MLPs ignore relationships — GNNs model them explicitly.
- GNNs are **permutation-invariant** and generalize across instance sizes.

Example domains: Vehicle Routing, Job Scheduling, Traffic Networks

Key Properties of GNNs

Permutation Invariance:

- Node ordering doesn't matter — GNN outputs stay the same.
- Achieved via `sum`, `mean`, or `max` aggregation in message passing.

Generalization:

- Models adapt to graphs of varying size and structure.
- Supports transfer to unseen problem instances.

GNNs: Message Passing Overview

Input: Graph $G = (V, E)$ with node features x_v, edge features e_{uv}

- **Initialization:** $h_v^{(0)} = x_v$
- **K-step propagation:** $h_v^{(K)}$ captures local/global structure
- **(Optional)** Readout for graph-level outputs

Using GNNs in RL Pipelines

- Replace standard MLPs with GNNs to encode:
  - **State representations:** Graph-structured environments (e.g., maps, schedules)
  - **Action representations:** When actions are edges, node pairs, or graph selections
- Works in policy gradient and actor-critic frameworks
- Outputs can be node-wise decisions or graph-level actions

Benefits:

- Generalizes across graph sizes
- Learns relational policies that adapt to structure

GNNs in RL Pipelines

Where GNNs fit:

- **State encoder:** for graph-structured environments
- **Action encoder:** when actions are nodes, edges, or subgraphs

Use cases:

- Works with policy gradient, actor-critic, and Q-learning
- Outputs can be node-level or graph-level decisions

Advantages:

- Learns structure-aware policies
- Transfers across varying problem instances

Case Study: GNN for VRP

Graph:

- Nodes: depot + customers
- Node features: $(x_i, y_i), d_i, [a_i, b_i]$
- Edge features: t_{ij}, ℓ_{ij} (travel time, distance)

GNN Encoding:

- $h_i^{(0)} = \text{MLP}(x_i, y_i, d_i, a_i, b_i)$
- $e_{ij} = \text{MLP}(t_{ij}, \ell_{ij})$
- Message passing yields context-aware node embeddings for routing

Applications of GNNs in RL

- Traveling Salesman Problem (TSP)
- Vehicle Routing Problem (VRP)
- Network design and flow control
- Scheduling with precedence
- Resource allocation on graphs

Most SOTA methods combine GNNs with attention + RL (policy gradient or actor-critic).

Challenges and Open Problems

- Scalability to large or dynamic graphs
- Incorporating domain constraints and feasibility checks
- Improving sample efficiency and training stability
- Interpretability of graph-based policies

Neural Combinatorial Optimization

Neural Combinatorial Optimization

Goal: Learn to solve discrete optimization problems using deep neural networks.

Typical setup:

- Input: instance of a combinatorial problem (e.g., graph, coordinates)
- Output: structured solution (e.g., tour, schedule, matching)
- Model: encoder–decoder architecture (e.g., LSTM, GNN, transformer)
- Training: via **reinforcement learning** or imitation learning

Handling Structured Action Spaces

Three main strategies:

1. **Autoregressive policies**
   - Generate complex action (e.g., route) step by step
   - E.g., Pointer Networks, Transformers, POMO
   - Enables sampling without enumerating full action space
2. **Action masking or feasibility projection (not in this lecture)**
   - Enforce constraints at each step
   - Use attention masks, feasibility checks, or decoders
   - Keeps actions valid without manual filtering
3. **Neighborhood search**
   - Search discrete action space around continuous proxy action
   - Local neighborhood search
   - Keeps actions valid without manual filtering

Problem: Action = structured object (e.g., tour, matching)
Solution: Generate solution step-by-step:

$$\pi(a_1, a_2, \ldots, a_T) = \prod_{t=1}^{T} \pi(a_t \mid a_{<t}, s)$$

Used in:

- Pointer Networks
- Transformers (attention-based decoding)

Benefits:

- No need to enumerate full action space
- Flexibility for variable-length outputs

Popular methods:

- Pointer Networks, Graph Neural Networks, Transformers to encode and decode graph structure
- **RL objective**: maximize reward = negative cost (e.g., tour length)
- **REINFORCE, PPO, Actor-Critic** commonly used

Why it's useful:

- Avoid hand-crafted heuristics
- Learn fast inference from data
- Generalize to unseen instances of similar structure

Why Learn TSP Heuristics with RL?

Travelling Salesman Problem (TSP):

- Given n cities, find shortest tour visiting all exactly once
- NP-hard: optimal solvers scale poorly for large n

Motivation:

- Learn policies that generalize across TSP instances
- Replace hand-crafted heuristics with trainable solvers
- Allow amortized optimization: fast inference once trained

Why RL?

- Objective (tour length) is **non-differentiable**
- Output is a **discrete sequence**
- No ground truth solutions needed → train from scratch

Pointer Networks (Vinyals et al. 2015)

Key idea:

- Use attention to "point" to elements of an input sequence
- Output is a permutation (e.g., tour over cities)

Model:

- Encoder: Transformer encodes city coordinates
- Decoder: Autoregressively generates the tour
- Policy: $\pi_\theta(\text{tour} \mid \text{cities})$

Training:

- Use REINFORCE: reward $= -(\text{tour length})$
- No supervision needed (unsupervised)

Limitation:

- Sampling one tour per gradient step $\rightarrow$ high variance

Attention-based GNNs for Routing (Kool et al. 2022)

Key idea:

- Learn deep policies for routing problems (e.g., TSP, VRP)
- Output a feasible route via an autoregressive decoder with masking

Model:

- **Encoder:** Deep Graph Attention Network (GNN with multi-head attention)
- **Decoder:** Autoregressive attention-based decoder
- **Policy:** $\pi_\theta(\text{solution} \mid \text{graph})$

Training:

- Deep RL: train π_θ using REINFORCE with learned baseline
- Reward: negative cost (e.g., tour length)
- No supervision needed — purely reward-driven learning

GNN-Based Heatmap

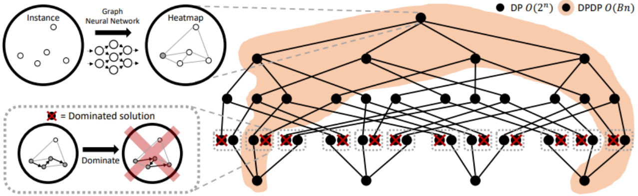


Figure 2: DPDP for the TSP. A GNN creates a (sparse) heatmap indicating promising edges, after which a tour is constructed using forward dynamic programming. In each step, at most B solutions are expanded according to the heatmap policy, restricting the size of the search space. Partial solutions are dominated by shorter (lower cost) solutions with the same DP state: the same nodes visited (marked grey) and current node (indicated by dashed rectangles).

Figure: Kool, W., van Hoof, H., Gromicho, J., & Welling, M. (2022). Deep policy dynamic programming for vehicle routing problems. CPAIOR.

POMO: Policy Optimization with Multiple Optima (Kwon et al. 2020) [1/2]

Problem: RL methods like REINFORCE sample 1 tour → high variance and slow learning

Key idea:

- Use multiple diverse starting points per TSP instance
- Generate multiple tours with the same policy
- Take best tour as reward → reduces variance

POMO: Policy Optimization with Multiple Optima (Kwon et al. 2020) [2/2]

Training:

$$\nabla_\theta J(\theta) = \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta \log \pi_\theta(a^i | s) \cdot R_{\text{best}}(s)$$

- B = number of rollouts (e.g., 20)
- R_{best} = reward of the best tour

Benefits:

- Stable training, faster convergence
- Improves performance without supervision

POMO Starting points

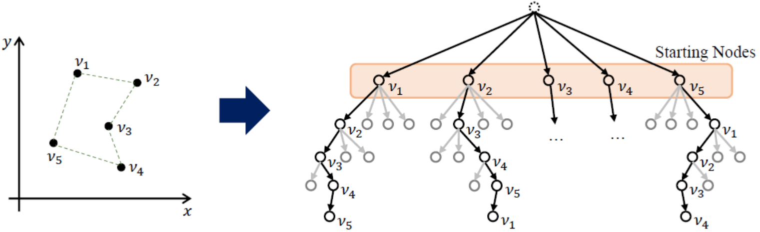


Figure: POMO utilizes multiple starting nodes to generate the same Hamiltonian cycle (Kwon, Y. D. et al. (2020))

Beyond POMO: Extensions and Variants

Model Enhancements:

- Use Transformer instead of LSTM $\rightarrow$ better scalability
- Masking and attention constraints to enforce feasibility

Training Variants:

- PPO instead of REINFORCE
- Imitation learning from solvers or heuristics (DAGGER)

Other problems:

- VRP (with capacity constraints)
- Orienteering Problem
- Job shop scheduling

Generative Flow Networks (GFlowNets)

What are GFlowNets?

- A framework for learning stochastic policies that **generate complex structured objects such as graphs, sequences or sets**.
- Instead of finding a single solution, GFlowNets **sample diverse high-reward solutions** proportionally to their reward.
- Useful for combinatorial generation and structured prediction.

Motivation for GFlowNets

- Standard RL aims to find **one** optimal policy or solution.
- In many applications, we want a **diverse set of good solutions** rather than just one.
- Sampling proportionally to reward helps explore multiple promising candidates.
- Bridges ideas from probabilistic modeling and RL.

Core Idea of GFlowNets

- Define a **generation process** as a sequence of actions building an object x.
- Assign a **flow** $F(s)$ to each intermediate state s.
- Flows satisfy the **flow matching condition**, which ensures that the total incoming and outgoing flow at each state is balanced, and terminal states receive flow proportional to their reward:

$$P(x) \propto R(x)$$

- Learn a policy that respects these flow constraints via trajectory balance or detailed balance objectives.

GFlowNets

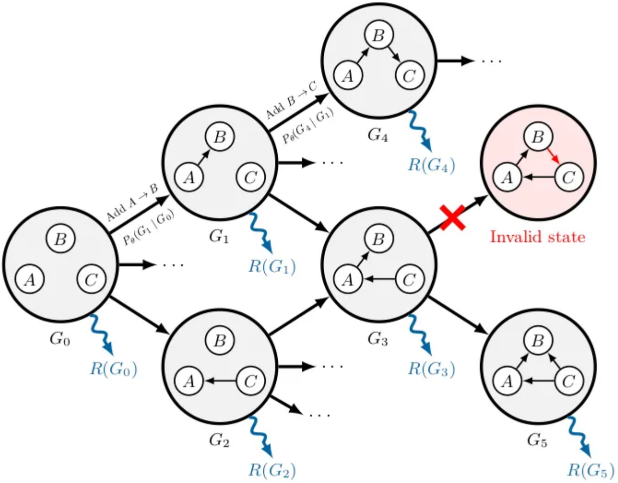


Figure: https://arxiv.org/abs/2202.13903

GFlowNets vs. Traditional RL

- **RL:** Optimizes expected return, focusing on one best solution.
- **GFlowNets:** Aim to sample diverse high-reward solutions proportionally.
- Provide a natural way to explore multimodal solution spaces.
- Can be trained with policy gradients, but with different objectives and constraints.

OR Example: Diverse Resource Allocation with GFlowNets

Problem: Allocate resources (e.g., staff, trucks, machines) to tasks under constraints.

Challenge: Many near-optimal solutions with trade-offs (cost, risk, availability).

Standard RL:

- Finds one optimal allocation (e.g., cheapest).
- Risks ignoring diverse trade-offs or alternatives.

GFlowNet Approach:

- Generate allocation plans step-by-step (sequential actions).
- Reward = objective value (e.g., efficiency score, feasibility).
- Learn to **sample diverse feasible plans** $\propto$ reward.

Applications of GFlowNets

- Combinatorial optimization: multiple near-optimal solutions
- Structured prediction and design problems
- Complement to existing RL approaches when diversity is critical

Research frontier: GFlowNets offer promising avenues for combining RL with probabilistic modeling.

OR Example: Diverse Resource Allocation with GFlowNets

Problem: Allocate resources (e.g., staff, trucks, machines) to tasks under constraints.

Challenge: Many near-optimal solutions with trade-offs (cost, risk, availability).

Standard RL:

- Finds one optimal allocation (e.g., cheapest).
- Risks ignoring diverse trade-offs or alternatives.

GFlowNet Approach:

- Generate allocation plans step-by-step (sequential actions).
- Reward = objective value (e.g., efficiency score, feasibility).
- Learn to **sample diverse feasible plans** $\propto$ reward.

Benefit: Decision-makers can explore a rich solution space and balance competing goals.

Summary: Generative Flow Networks

Key takeaways:

- GFlowNets are a novel framework to **learn stochastic policies** that generate complex structures.
- They aim to **sample from a reward-proportional distribution**, not just maximize it.
- Useful in tasks where **diverse high-quality solutions** are needed.
- Bridge the gap between **reinforcement learning and probabilistic inference**.

Outlook: Promising for applications in design, discovery, and combinatorial decision-making.

Large Combinatorial Action Spaces

Dealing with Large Combinatorial Action Spaces

Challenge: Discrete combinatorial action spaces often grow exponentially, making exhaustive search or enumeration infeasible.

Solution direction:

- Use actor network to generate continuous proxy action
- Search neighborhood around proxy action
- Evaluate quality of neighbors based on, e.g., Q-values

This section explores various techniques for efficient neighborhood search.

Neighborhood search

Large discrete action spaces in Deep RL

- Dynamically create promising neighborhoods around continuous proxy action (actor)
- Control search space
- Optional: Explore generated neighborhood for best Q-value (critic)
- No need for enumeration of full action space

MinMax

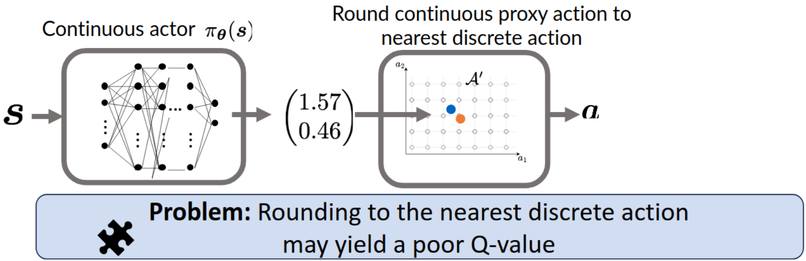


Figure: MinMax rounds the continuous proxy action to the nearest discrete action (Vanvuchelen et al., 2023)

k-Nearest Neighbor

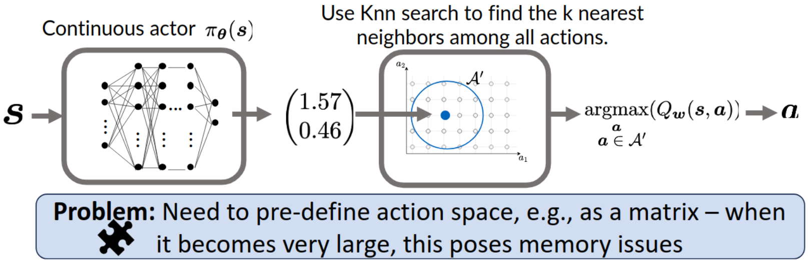


Figure: k-Nearest Neighbor stores k nearest neighbors based on Euclidean distance

Learned action representations

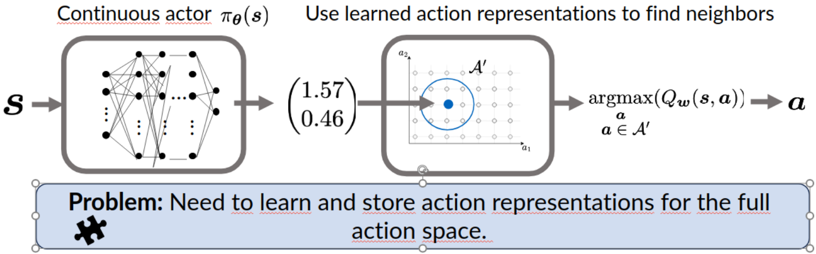


Figure: Learned action representations reflect KL divergence of actions through a preliminary supervised learning phase (Chandak et al., 2019)

MILP formulation[1/3]

$$
\begin{aligned}
&\max_{\boldsymbol{a}} \left(Q_{\boldsymbol{w}}(\boldsymbol{s}, \boldsymbol{a})\right) = \max_{a_i, y_{d_l}} \sum_{d_L} w_{d_L,o}\, y_{d_L} && \\
\text{s.t.} \quad & a_i^{l'}(\hat{\boldsymbol{a}}) \le a_i \le a_i^{u'}(\hat{\boldsymbol{a}}), && \forall i \in \{1, \dots, N\}, \\
& y_{d_l} \ge 0, && \forall l \in \mathcal{L}, d \in \mathcal{D}, \\
& y_{d_l} \ge \sum^{|\boldsymbol{a}|<D_l} w_{d_{l-1},d_l}\, a_{d_{l-1}} + \sum^{D_l} w_{d_{l-1},d_l}\, y^{\boldsymbol{s}}_{d_{l-1}}, && l = 2, \forall d_l.
\end{aligned}
$$

Figure: Neighborhood search (Akkerman et al., 2024))

Optimize local neighborhood with constrained decision variables

MILP formulation [2/3]

$$
\begin{aligned}
y_{d_l} &\geq \sum_{d_{l-1} \in \mathcal{D}_{l-1}} w_{d_{l-1},d_l}\, y_{d_{l-1}} \\
y_{d_l} &\leq z_{d_l}\, M \\
y_{d_l} &\leq (1 - z_{d_l})\, M + \sum_{d_{l-1} \in \mathcal{D}_{l-1}} w_{d_{l-1},d_l}\, y_{d_{l-1}} \\
z_{d_l} &\geq \frac{\sum_{d_{l-1} \in \mathcal{D}_{l-1}} w_{d_{l-1},d_l}\, y_{d_{l-1}}}{M} \\
z_{d_l} &\leq 1 + \frac{\sum_{d_{l-1} \in \mathcal{D}_{l-1}} w_{d_{l-1},d_l}\, y_{d_{l-1}}}{M} \\
z_{d_l} &\in \{0, 1\}\,.
\end{aligned}
$$

Figure: RELU constraints (Van Heeswijk & La Poutré, 2020)

Technical constraints required to describe and ensure consistency of the ReLU activation functions

MILP formulation [3/3]

$$k \geq \sum_{j:\bar{a}_j = a_j^{l'}} \mu_j \left(\bar{a}^* - a_j^{l'}\right) + \sum_{j:\bar{a}_j = a_j^{u'}} \mu_j \left(a_j^{u'} - \bar{a}^*\right) + \sum_{j:a_j^{k'} < \bar{a}_j < a_j^{u'}} \mu_j \left(a_j^+ + a_j^-\right),$$

where $\mu_j = \frac{1}{a_j^{u'} - a_j^{l'}}$ and

$$\bar{a}_j^* = \bar{a}_j + a_j^+ - a_j^-; \qquad a_j^+ \geq 0,\, a_j^- \geq 0; \qquad \forall j : a_j^{l'} < \bar{a}_j^* < a_j^{u'}.$$

Figure: Local branching constraints (Fischetti & Lodi, 2003)

Bounds the maximum Hamming distance k between the base action $\bar{\boldsymbol{a}}$ and the resulting optimal action $\bar{\boldsymbol{a}}^*$

MILP

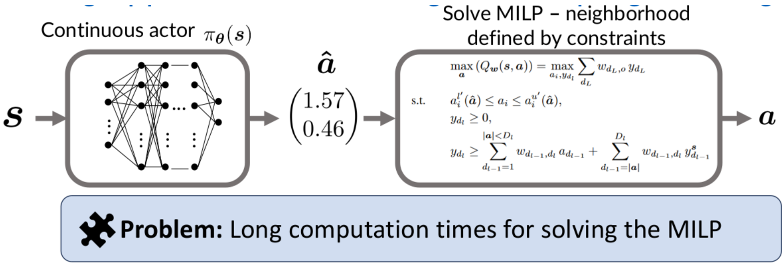


Figure: A restricted neighborhood is searched by solving a MILP (Akkerman et al., 2024)

Dynamic neighborhood construction: Main idea

Dynamic neighborhood construction

- Construct limited set of perturbed actions within the neighborhood, offsetting one element at a time
- Use simulated annealing to search around perturbed actions, creating new neighborhoods
- Balance improving and randomized actions

Dynamic Neighborhood Construction

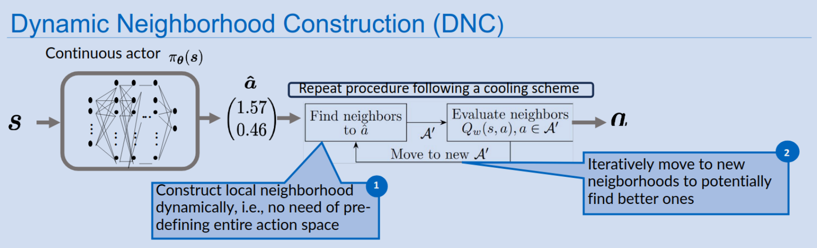
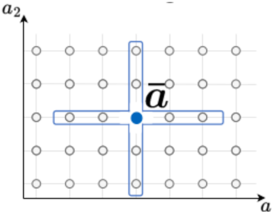


Figure: Dynamic Neighborhood Construction iteratively constructs and searches local neighborhoods

Step 1: Dynamically construct neighborhood of $\hat{a}$

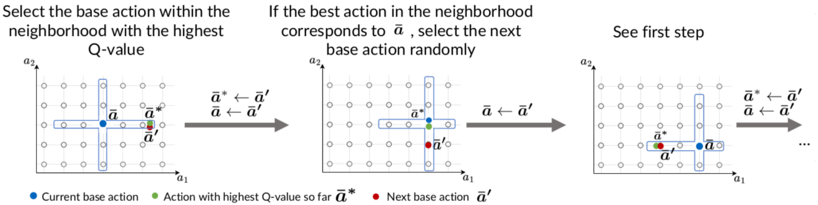


- Generate neighborhood to $\bar{a}$ by perturbing it using perturbation matrices P_{ij}

$$P_{ij} = \begin{cases} \epsilon\left(\lfloor (j-1)/N \rfloor + 1\right), & \text{if, } j \in \{i, i+N, i+2N, \ldots, i+(d-1)\,N\}, \\ -\epsilon\left(\lfloor (j-1)/N \rfloor + 1 - d\right), & \text{if, } j \in \{i+dN, i+(d+1)N, \ldots, i+(2d-1)\,N\}, \\ 0, & \text{otherwise.} \end{cases}$$

- Neighbors now all have Hamming distance of 1 and maximum L2 distance of $d \cdot \epsilon$

Step 2: Iteratively explore neighborhood



- Number of iterations and degree of randomness controlled by simulated annealing (SA) parameters, i.e., cooling factor and maximum number of iterations.
- SA process ensures exploration and avoidance of local action minima.

Model-Based Reinforcement Learning

Model-Based Reinforcement Learning (MBRL)

What is Model-Based RL?

- Learns or uses a **model** of environment dynamics $\hat{P}(s'|s,a)$
- Uses model to plan or generate synthetic experience
- Contrasts with model-free methods that learn value or policy directly

Advantages:

- Sample efficient: can learn from fewer real interactions
- Can leverage classical planning and optimization
- Useful in OR for simulating complex systems

Core Components of Model-Based RL

- **Model Learning:** Learn or specify $\hat{P}(s'|s, a)$, reward model $\hat{r}(s, a)$
- **Planning:** Use the model for policy improvement or action selection
  - Dynamic programming
  - Tree search (e.g., MCTS)
  - Trajectory optimization
- **Policy Learning:** Learn a policy from model-generated data or planning results

Trade-offs:

- Model bias vs. sample efficiency
- Computational complexity in planning

Monte Carlo Tree Search (MCTS)

Key idea: Build a search tree by simulating rollouts to evaluate actions.

Four steps:

1. **Selection:** Traverse tree to select promising node (using, e.g., UCT)
2. **Expansion:** Add a new child node (state-action)
3. **Simulation (Rollout):** Simulate to end or depth with a simple policy, e.g., heuristic
4. **Backpropagation:** Update value estimates up the tree

Applications: Game playing (Go, Chess), planning in robotics

Monte Carlo Tree Search

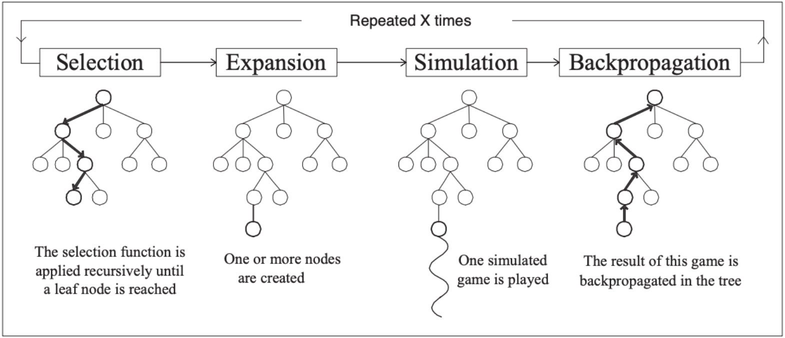


Figure 1: Outline of a Monte-Carlo Tree Search.

Limitations of Classic MCTS

- Requires known, accurate environment model
- Rollouts can be costly or low-quality if policy is weak
- Scalability issues in large or continuous state spaces
- Not directly applicable to unknown or partially observable environments

AlphaGo: Deep Learning + Tree Search

Innovation:

- Combines supervised learning, RL, and **MCTS**
- Learns both **policy networks** (to guide tree search) and **value networks** (to prune branches)
- Achieves superhuman performance in Go using expert games and self-play

Architecture:

- **Policy network:** proposes promising next moves
- **Value network:** estimates win probability of a position
- **MCTS:** guided by policy priors and refined by value estimates

Result: First system to defeat a world champion in Go (Lee Sedol, 2016)

MCTS in AlphaGo

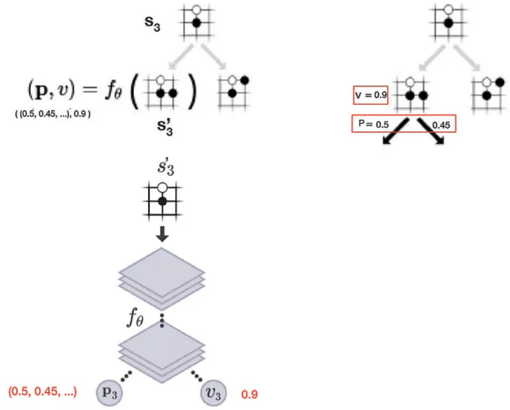


Figure: Comparison of AlphaZero and MuZero architectures
(`https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a`)

AlphaZero: Extension to Other Environments

Innovation:

- Learns to play Go, Chess, and Shogi **from scratch**
- Uses only **self-play**, without expert data
- Unified deep RL + MCTS framework across games

Architecture:

- **Shared ResNet:** outputs both policy logits and value estimate
- **MCTS:** uses policy as prior, value for leaf evaluation
- **Training:** improves policy and value via self-play rollouts

Result: Surpasses all previous versions and top engines (e.g., Stockfish) using general principles

MuZero: Learning Models and Planning

Innovation:

- Learns a **latent dynamics model** without observing true states
- Integrates **MCTS** with deep networks for policy and value estimation
- Combines **model-free** learning and planning

Components:

- **Representation network:** encodes history into latent state
- **Dynamics network:** predicts next latent state and reward
- **Prediction network:** outputs policy and value from latent state

Results: State-of-the-art in Go, Chess, Atari without explicit environment model

AlphaZero and MuZero Architectures

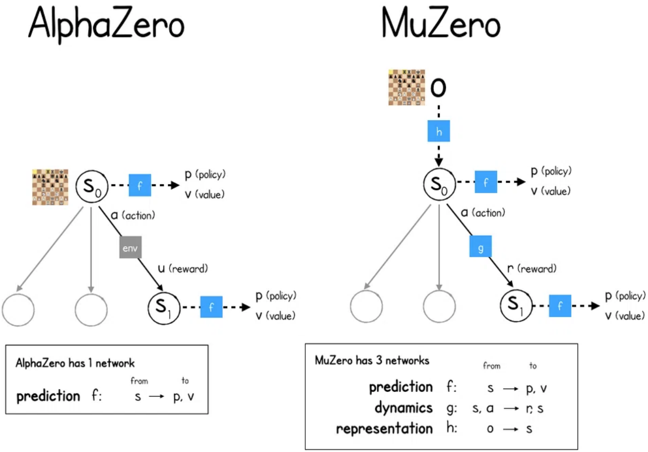


Figure: Comparison of AlphaZero and MuZero architectures (`https://note.com/npaka/n/n2085bfddd86c`)

AlphaGo vs AlphaZero vs MuZero

Aspect	AlphaGo	AlphaZero	MuZero
Learning	Human + RL	Self-play RL	Self-play RL
Games	Go	Go, Chess, Shogi	+ Atari
Model	Rules-based	Rules-based	**Latent, learned**
State Pred.	Yes	Yes	**No (latent only)**
Planning	MCTS + rollouts	MCTS + value	MCTS in latent space
Network Out	Policy, Value	Policy, Value	Policy, Value, Reward
Architecture	Separate nets	Unified net	**3 nets (repr., dyn., pred.)**

Note: MuZero is model-based but learns a latent model optimized for planning—not for predicting observations.

Other Model-Based RL Methods

- **Dyna:** Combines model learning and model-free updates (Sutton, 1991)
- **MBPO:** Short model-generated rollouts to boost training
- **PlaNet / Dreamer:** Latent dynamics with VAEs for continuous control
- **Guided Policy Search:** Uses trajectory optimization with learned dynamics

MBRL in Operations Research

- Simulate complex systems (supply chains, logistics) with learned models
- Combine with classical optimization and stochastic programming
- Improve sample efficiency in costly or slow-to-simulate environments
- Plan under uncertainty using tree search or trajectory optimization

Open research: Integrating MBRL with domain constraints, scalability, and interpretability

Multi-Agent Reinforcement Learning

What is Multi-Agent Reinforcement Learning (MARL)?

- **Multiple agents** interact in a shared environment
- Each agent learns a policy to **maximize its own expected return**
- Agents may be:
  - **Cooperative**: shared reward (team setting)
  - **Competitive**: one agent's gain is another's loss
  - **Mixed**: partially aligned or opposing incentives
- Applications: games, traffic control, auctions, energy markets, logistics

MARL Visualized

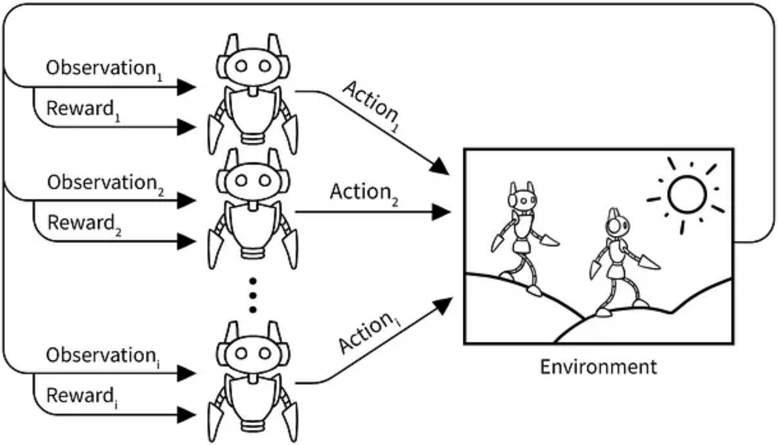


Figure: https://medium.com/data-science/multi-agent-deep-reinforcement-learning-in-15-lines-of-code-using-pettingzoo-e0b963c0820b

Why is Multi-Agent RL Hard?

- **Non-stationarity:** The environment changes as other agents learn
- **Interdependent learning:** Each agent's strategy affects the others
- **Exploration difficulty:** Poor coordination leads to suboptimal behavior
- **Scalability:** Joint state/action spaces grow exponentially with number of agents
- **Credit assignment:** Difficult to attribute outcomes to individual agents

Key insight: Each agent learns in a dynamic multi-agent ecosystem — not a fixed world.

From Single-Agent RL to Multi-Agent Games

- In single-agent RL, the environment is stationary.
- In MARL, the environment includes other agents $\rightarrow$ **non-stationary**.
- This naturally leads to **game theory**: reasoning about other decision-makers.

RL Meets Game Theory

Game-theoretic View:

- Each agent = a player
- Environment = repeated or stochastic game
- Optimality concept = **Nash Equilibrium** or correlated equilibrium

Types of Games:

- **Fully cooperative:** shared reward (team game)
- **Zero-sum:** one agent's gain is another's loss
- **General-sum:** mix of incentives

Key insight: In MARL, learning becomes a form of equilibrium-seeking.

Markov Games (Stochastic Games)

Extension of MDP to multiple agents:

$$(\mathcal{S}, \{\mathcal{A}_i\}_{i=1}^n, P, \{r_i\}_{i=1}^n, \gamma)$$

- $\mathcal{S}$: state space
- $\mathcal{A}_i$: action space of agent i
- $P(s'|s, a_1, \ldots, a_n)$: transition dynamics
- $r_i(s, \vec{a})$: reward for agent i
- Each agent has its own policy: $\pi_i(a_i|s)$

Goal: Each agent maximizes its own expected return.

Centralized vs. Decentralized Learning

Centralized Training:

- Joint policy or value function
- Access to all agents' states/actions
- Often used during training (e.g., actor-critic)

Decentralized Execution:

- Each agent acts independently based on local observation
- Needed in real-time multi-agent systems

Popular setup: CTDE = Centralized Training with Decentralized Execution

Challenges in MARL

- **Non-stationarity:** agents' policies keep changing
- **Credit assignment:** who caused the global reward?
- **Scalability:** joint action/state space grows exponentially
- **Exploration:** need to coordinate exploration across agents

Research directions:

- Learn equilibrium concepts directly
- Implicit coordination via architecture or training scheme
- Scalability via factorization, graph structure

Key MARL Algorithms

Independent Learning:

- Each agent uses its own RL algorithm (e.g., DQN, PPO)
- Simple but suffers from non-stationarity

Joint Action Learners:

- Learn Q-values over joint actions
- Not scalable beyond small agent numbers

Actor-Critic Extensions:

- MADDPG (multi-agent DDPG with shared critic)
- QMIX (centralized Q-function factorized across agents)
- COMA (counterfactual advantage for credit assignment)

Nash Q-Learning [1/2]

Setting: Multi-agent general-sum stochastic games (Markov Games)
Goal: Learn Q-values such that each agent follows a **Nash equilibrium** policy.

Key idea:

- Each agent learns a Q-function $Q_i(s, \vec{a})$ over joint actions.
- At each step, agents compute the **stage-game Nash equilibrium** for current state s:

$$\pi^*(s) \in \mathrm{NashEq}(\{Q_i(s, \cdot)\}_{i=1}^n)$$

- Use equilibrium strategies to update Q-values.

Nash Q-Learning [2/2]

Update rule (simplified):

$$Q_i(s, \vec{a}) \leftarrow Q_i(s, \vec{a}) + \alpha \left[r_i + \gamma V_i(s') - Q_i(s, \vec{a}) \right]$$

where $V_i(s') = \mathbb{E}_{\vec{a}' \sim \pi^*(s')}[Q_i(s', \vec{a}')]$

Challenges:

- Computing Nash equilibria is expensive in large games.
- Multiple equilibria possible — which one to use?
- Does not scale to many agents or large action spaces.

COMA: Counterfactual Multi-Agent Policy Gradient

Problem: In cooperative MARL, it's hard to assign credit to individual agents.

COMA idea:

- Centralized critic estimates global Q-function: $Q(s, \vec{a})$
- Compute a **counterfactual baseline** to isolate agent i's contribution:

$$A_i = Q(s, \vec{a}) - \sum_{a_i'} \pi_i(a_i'|o_i) Q(s, (\vec{a}_{-i}, a_i'))$$

- Use A_i as the advantage in a standard actor-critic update:

$$\nabla J_i = \mathbb{E}[\nabla \log \pi_i(a_i|o_i) A_i]$$

Benefits:

- Addresses the **multi-agent credit assignment** problem
- Stable learning of decentralized policies with shared goals

MARL as Equilibrium Learning

When do agents converge to equilibrium?

- **Nash Equilibrium:** no agent can improve by deviating
- In general-sum games, convergence is not guaranteed

Approaches:

- Fictitious Play
- Policy Space Response Oracles (PSRO)
- Opponent modeling (e.g., LOLA: Learning with Opponent Learning Awareness)

Application: Bidding, traffic routing, pricing competitions

Opponent Modeling in Multi-Agent RL [1/2]

How to deal with non-stationary opponents?

- **Ignore:**
  - Assume opponent is stationary (e.g., fixed mixed strategy).
  - Example: Fictitious play.
  - Fails if opponent behavior changes later.
- **Forget:**
  - Adapt learning rate to changing behavior.
  - Example: WoLF-PHC adapts faster when losing.
  - Works well in self-play; less so against unknown strategies.
- **Respond to Target Opponents:**
  - Assume opponent switches among known strategies.
  - Example: HM-MDPs track mode-switching behavior.
  - Limited adaptability if opponent acts outside known set.

Opponent Modeling in Multi-Agent RL [2/2]

- **Learn Opponent Models:**
  - Learn models from data without assuming known strategy classes.
  - Respond to detected shifts or reused strategies.
  - Doesn't handle strategic reasoning (opponent reacting to you).
- **Theory of Mind:**
  - Recursive reasoning: you model them modeling you.
  - Levels of reasoning (L0, L1, L2, . . .), compute best response at each level.
  - Powerful, but expensive; requires known base strategies.

Decentralizing Fleet Optimization with Cooperative MARL

Problem: Centralized fleet control (e.g., taxis, trucks, drones) is intractable for large-scale systems.

MARL Perspective:

- Model each vehicle as an **agent** in a shared environment.
- Agents observe local information (location, demand, traffic).
- Learn decentralized policies to **maximize shared reward** (e.g., service level, efficiency).

Advantages:

- Scalable to many agents
- Robust to partial observability and local delays
- Enables online adaptation to changing environments

Example: Autonomous taxi fleet learning to position vehicles in real-time based on demand forecasts

Post-Optimization after MARL Planning

Problem: MARL may produce high-quality solutions that are **not fully feasible** under operational constraints.

Solution:

- Use MARL output (e.g., routes, assignments) as a **warm start**.
- Apply post-optimization (e.g., ILP, heuristics, metaheuristics) to:
  - Enforce hard constraints (capacity, working hours, regulations)
  - Improve cost efficiency and feasibility

Hybrid Optimization Pipeline:

MARL Policies $\rightarrow$ Candidate Solution $\rightarrow$ OR Post-Processing

Outcome: Combines **learning-based flexibility** with **OR precision**.

Bidding as Autonomous Decision-Making

Scenario: Freight logistics involves interaction between **carriers** and **shippers**, often in decentralized settings.

Mechanism:

- **Carrier:** Announces availability and asks for a minimum acceptable price to execute a shipment.
- **Shipper:** Observes multiple carriers and **bids a price** for its shipment to be picked up.
- If bid $\geq$ ask: shipment is accepted and executed.

Learning Opportunities:

- Shippers learn bidding strategies based on historical success and urgency.
- Carriers adjust asking prices based on capacity, time windows, and expected competition.

Bidding Example

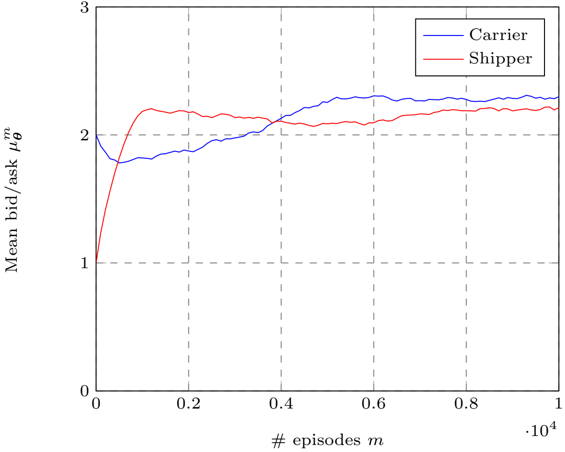


Figure: https://link.springer.com/article/10.1007/s10479-022-04572-z

OR Applications of MARL

Where does MARL arise in OR?

- **Traffic assignment:** vehicles act independently, with coupled constraints
- **Fleet routing:** decentralized truck decisions, central coordination
- **Bidding platforms:** freight marketplaces, online auctions
- **Energy markets:** agent-based simulations for price dynamics

Goal: Use MARL to simulate, optimize, or learn equilibria or credit assignment in competitive or collaborative OR environments

Summary: Multi-Agent Reinforcement Learning

What we've seen:

- **Definition:** Multiple agents learning simultaneously in shared environments
- **Challenges:** Non-stationarity, scalability, credit assignment, strategic behavior
- **Modeling:** Markov games, game-theoretic perspectives, centralized training
- **Algorithms:** Independent learners, Nash Q-learning
- **Opponent modeling:** From reactive to recursive (theory of mind)
- **OR relevance:** Fleet optimization, bidding, traffic, energy, auctions

Takeaway: MARL extends RL to competitive and cooperative multi-agent systems — crucial for many OR problems.

Wrapping up

Lecture 3 Summary: Generalization, Structure and Realism in RL

What we covered:

- Graph Neural Networks to encode problem structure
- Neural combinatorial optimization to autoregressively construct solutions
- Neighborhood sampling methods to handle large discrete action spaces
- Model-based Reinforcement Learning: from MCTS to MuZero
- Multi-Agent Reinforcement Learning (MARL), linking learning to game theory

Takeaway: Optimization problems typically have a structure that can be leveraged in tailored RL algorithms.

Contact details

Wouter van Heeswijk
Assistant Professor
Operations Research & Financial Engineering
University of Twente
w.j.a.vanheeswijk@utwente.nl
+31 53 489 8460