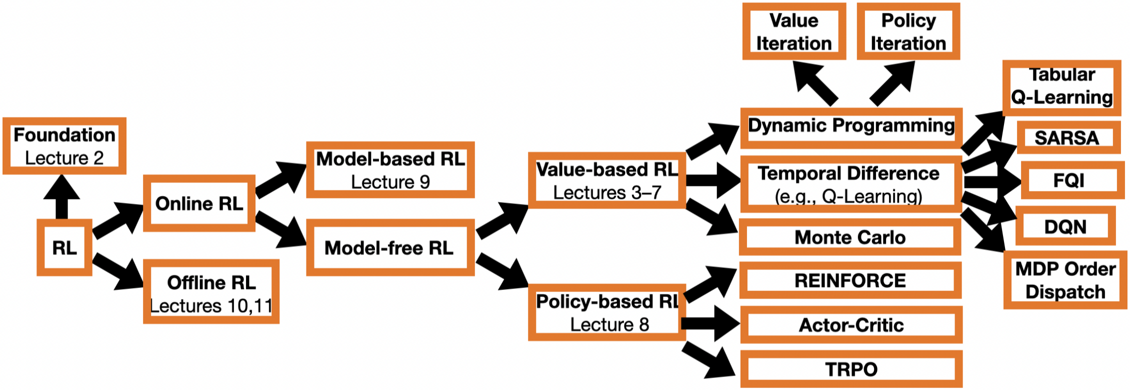
ST455: Reinforcement Learning

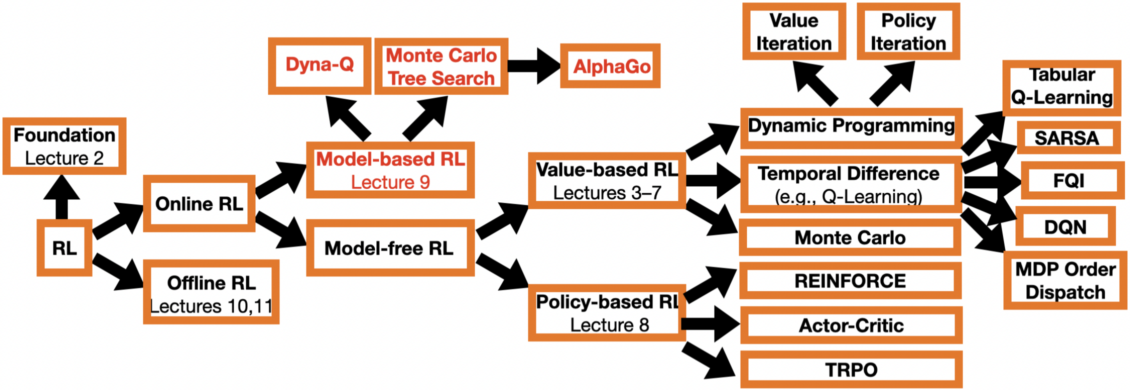
Lecture 9: Model-based Reinforcement Learning

Chengchun Shi

Roadmap


Foundation
Lecture 2
RL
Online RL
Offline RL
Lectures 10,11
Model-based RL
Lecture 9
Model-free RL
Value-based RL
Lectures 3–7
Policy-based RL
Lecture 8
Value Iteration
Policy Iteration
Dynamic Programming
Temporal Difference
(e.g., Q-Learning)
Monte Carlo
REINFORCE
Actor-Critic
TRPO
Tabular Q-Learning
SARSA
FQI
DQN
MDP Order Dispatch

Roadmap (Cont'd)



Lecture Outline

1. What is Model-based RL
2. How to Implement Model-based RL
3. Mastering the Game of Go

Lecture Outline

1. **What is Model-based RL**
2. How to Implement Model-based RL
3. Mastering the Game of Go

Recap: Planning vs Learning

Two fundamental problems in sequential decision making

- **Planning**
  - A model of the environment (e.g., state transition, reward function) is **known**
  - The agent performs computations with its model, **without** any external interaction
  - a.k.a. deliberation, reasoning, introspection, pondering, thought, search
- **Learning**
  - The environment is initially **unknown**
  - The agent **interacts** with the environment
  - The agent **learns** the optimal policy from experience

RL Algorithms We Have Covered So Far

- **Dynamic Programming** (Lecture 3): learn **value** from **model** (planning)
- **MC**, **TD** (Lectures 3 - 7): learn **value** from **experience** (learning)
- **Policy Gradient** (Lecture 8): learn **policy** from **experience** (learning)
- Today's lecture: **Model-based** RL
  - learn **model** from experience
  - use both learned model and experience to construct a **value** function or **policy**
  - combine learning with planning

What is a Model?

- A model $\mathcal{M}$ is a **representation** of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
- The state space $\mathcal{S}$ and action space $\mathcal{A}$ are usually known to us
- The discounted factor γ is **user-specified**
- Only need to learn the state transition $\mathcal{P}$

$$\mathcal{P}_{ss'}^{a} = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$$

and reward function $\mathcal{R}$

$$\mathcal{R}_{s}^{a} = \mathbb{E}(R_t | S_t = s, A_t = a)$$

Model-free v.s. Model-based RL

- Model-based RL
  - Learn the model (e.g., reward $\mathcal{R}_s^a$ and transition $\mathcal{P}_{ss'}^a$) from experience
  - **Plan** value or policy from model or **integrate** planning with learning
- Model-free RL
  - **Learn** value or policy **without** learning the reward and transition function
  - Rely on Bellman optimality equation
  - Examples: MC, TD, Policy gradient

Model-free v.s. Model-based RL (Cont'd)

- **Pros** of model-based RL
- In some applications, we have a **perfect** model (e.g., Go, chess)
- Can handle **offline** data (more in the next lecture)

- **Pros** of model-free RL
- **Dimensional reduction**
- Easier to learn value than model
- # of parameters of $Q^{\pi^{\text{opt}}}$: $|\mathcal{S}||\mathcal{A}|$
- # of parameters of $\mathcal{R}_s^a$: $|\mathcal{S}||\mathcal{A}|$
- # of parameters of $\mathcal{P}_{ss'}^a$: $|\mathcal{S}|^2|\mathcal{A}|$

Lecture Outline

1. What is Model-based RL

2. How to Implement Model-based RL

3. Mastering the Game of Go

How to Implement Model-Based RL

- First, we learn a **model** (reward and state transition functions) based on data
- Next, we can implement **planning** based on the learned model
- Alternatively, we can **integrate planning with learning** (Dyna)
- Finally, we can implement **Monte Carlo tree search** for decision-time planning

How to Implement Model-Based RL

- First, we learn a **model** (reward and state transition functions) based on data
- Next, we can implement **planning** based on the learned model
- Alternatively, we can **integrate planning with learning** (Dyna)
- Finally, we can implement **Monte Carlo tree search** for decision-time planning

Model Learning

- **Goal**: estimate $\mathcal{R}_s^a$ and $\mathcal{P}_{ss'}^a$ from experience $\{S_0, A_0, R_0, \cdots, S_T\}$
- Using supervised learning

$$
\begin{aligned}
S_0, A_0 &\rightarrow R_0, S_1 \\
S_1, A_1 &\rightarrow R_1, S_2 \\
&\vdots \\
S_{T-1}, A_{T-1} &\rightarrow R_{T-1}, S_T
\end{aligned}
$$

- Learning $s, a \rightarrow r$ is a **regression** problem
- Learning $s, a \rightarrow s'$ is a **conditional density estimation** problem
- Loss function: least square/Huber loss, KL divergence
- Compute parameter that minimizes empirical loss

Models for Conditional Density Estimation

- Table lookup model
- Conditional kernel density estimation
- Gaussian process model [Williams and Rasmussen, 2006]
- Deep conditional generative learning[1]
    - mixture density network [Rothfuss et al., 2019]
    - normalising flows [Trippe and Turner, 2018]

[1] https://deepgenerativemodels.github.io/notes/index.html

Table Lookup Model

- Finite MDP model
- Count visits $N(s,a) = \sum_{t=0}^{T-1} \mathbb{I}(S_t = s, A_t = a)$ to each state-action pair

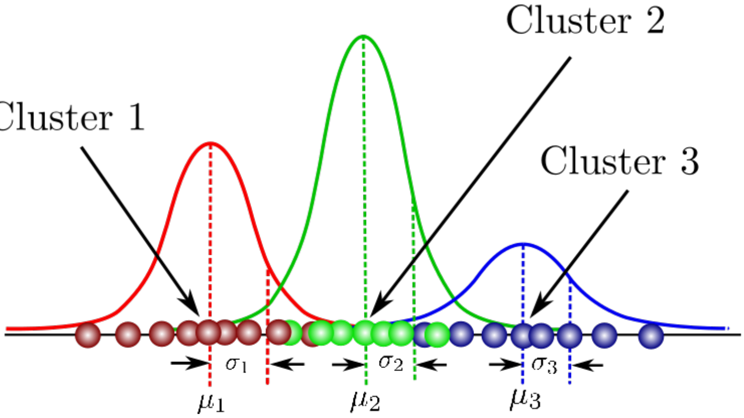
$$
\begin{aligned}
\widehat{\mathcal{P}}_{ss'}^{a} &= \frac{1}{N(s,a)} \sum_{t=0}^{T-1} \mathbb{I}(S_t = s, A_t = a, S_{t+1} = s') \\
\widehat{\mathcal{R}}_{s}^{a} &= \frac{1}{N(s,a)} \sum_{t=0}^{T-1} \mathbb{I}(S_t = s, A_t = a) R_t
\end{aligned}
$$

- Alternatively
  - At each time step t, record experience tuple $\langle S_t, A_t, R_t, S_{t+1} \rangle$
  - To sample model, based on a state-action pair (s,a), randomly pick tuple matching $\langle s, a, \bullet, \bullet \rangle$

Mixture Density Network

- Learn a generic conditional probability mass/density function of $\boldsymbol{Y}$ given $\boldsymbol{X} = \boldsymbol{x}$, $\boldsymbol{f}(\boldsymbol{y}|\boldsymbol{x})$ ($\boldsymbol{Y} = \boldsymbol{S}_{t+1}$ and $\boldsymbol{X} = (\boldsymbol{S}_t, \boldsymbol{A}_t)$ in our RL setting)
- Combine **Gaussian mixture model** with **deep neural networks**
- Gaussian mixture model has universal approximation property to approximate any **density** function
- Deep neural networks have universal approximation property to approximate any **mean and variance** functions in Gaussian distribution

What is a Gaussian Mixture Model



Taken from `https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95`

Gaussian Mixture Model (Cont'd)

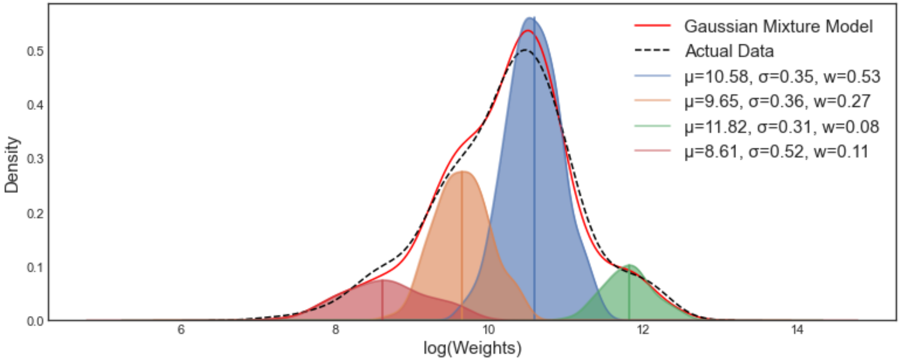
Model a probability density function $\boldsymbol{f}(\boldsymbol{y})$ by

$$\boldsymbol{f}(\boldsymbol{y}) = \sum_{\boldsymbol{k}=1}^{\boldsymbol{K}} \omega_{\boldsymbol{k}} \phi(\boldsymbol{y}; \boldsymbol{\mu}_{\boldsymbol{k}}, \boldsymbol{\sigma}_{\boldsymbol{k}}^2),$$

where $\phi(\bullet; \boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ denotes the probability density function of a Gaussian variable with mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma}^2$, and $\boldsymbol{\omega}_{\boldsymbol{k}}$ denotes the probability the variable belongs to the $\boldsymbol{k}$th cluster

Universal Approximation Property

Gaussian mixture model approximates any probability density function as the number of clusters $K \to \infty$



Mixture Density Network

- Model a probability density function $f(y)$ by

$$f(y) = \sum_{k=1}^{K} \omega_k \phi(y; \mu_k, \sigma_k^2)$$

- We want to model a **conditional** probability density function $f(y|x)$
- Can be modelled via a **conditional** Gaussian mixture model

$$f(y|x) = \sum_{k=1}^{K} \omega_k(x) \phi(y; \mu_k(x), \sigma_k^2(x))$$

- Use **deep neural networks** to parametrize $\omega_k(\bullet)$, $\mu_k(\bullet)$ and $\sigma_k^2(\bullet)$

How to Implement Model-based RL

- First, we learn a **model** (reward and state transition functions) based on data
- Next, we can implement **planning** based on the learned model
- Alternatively, we can **integrate planning with learning** (Dyna)
- Finally, we can implement **Monte Carlo tree search** for decision-time planning

Planning with Dynamic Programming

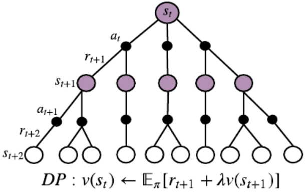
- Give a model $\langle \widehat{\mathcal{R}}, \widehat{\mathcal{P}} \rangle$
- Use dynamic programming algorithm
  - Policy iteration

$$\pi_0 \longrightarrow V^{\pi_0} \longrightarrow \pi_1 \longrightarrow V^{\pi_1} \longrightarrow \cdots \longrightarrow \pi^{opt} \longrightarrow V^{\pi^{opt}}$$

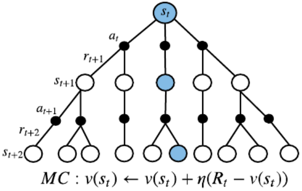
  - Value iteration

$$V^{\pi_0} \longrightarrow V^{\pi_1} \longrightarrow V^{\pi_2} \longrightarrow \cdots \longrightarrow V^{\pi^{opt}} \longrightarrow \pi^{opt}$$

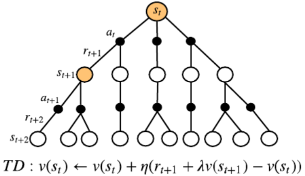
Difference From Model-Free Methods



Dynamic Programming (DP)



Monte Carlo (MC)



Temporal Difference (TD)

Planning with Model-free RL

- A simple but powerful approach to planning
- Use the model only to **generate samples**
- **Sample** experience from model:

$$S' \sim \widehat{\mathcal{P}}^{A}_{S,\bullet} \quad \text{and} \quad R = \widehat{\mathcal{R}}^{A}_{S}$$

- Apply **model-free** RL to samples
  - MC control
  - SARSA
  - Q-learning
- This is often more **efficient** than dynamic programming-based method

Planning with an Inaccurate Model

- Model-based RL computes $\boldsymbol{\pi}^{\textbf{opt}}$ with respect to the model $\langle \mathcal{S}, \mathcal{A}, \widehat{\mathcal{R}}, \widehat{\mathcal{P}}, \gamma \rangle$
- Quality of the estimated policy depends heavily on the accuracy of the model
- When model is inaccurate, planning yields a **suboptimal** policy
- **Solution** 1: when model is wrong, using model-free RL
- **Solution** 2: integrate planning with learning

How to Implement Model-based RL

- First, we learn a **model** (reward and state transition functions) based on data
- Next, we can implement **planning** based on the learned model
- Alternatively, we can **integrate planning with learning** (Dyna)
- Finally, we can implement **Monte Carlo tree search** for decision-time planning

Real and Simulated Experience

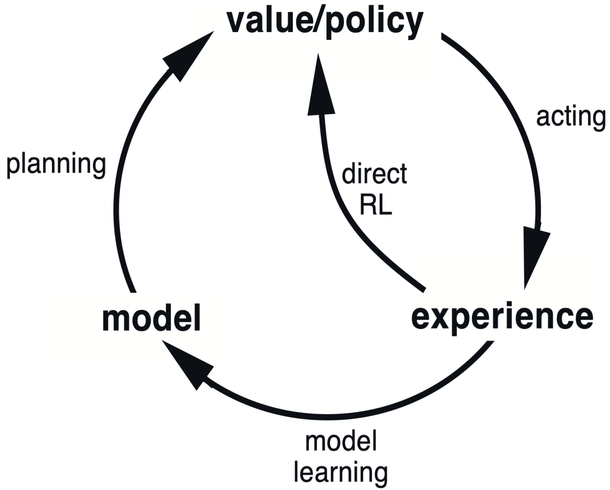
- We consider two sources of experience
- **Real experience**: Sampled from environment (true MDP)

$$\{S_0, A_0, R_0, \cdots, S_T\}$$

- **Simulated experience**: Sampled from model (estimated MDP)

$$S' \sim \widehat{\mathcal{P}}_{S,\bullet}^{A} \quad \text{and} \quad R = \widehat{\mathcal{R}}_{S}^{A}$$

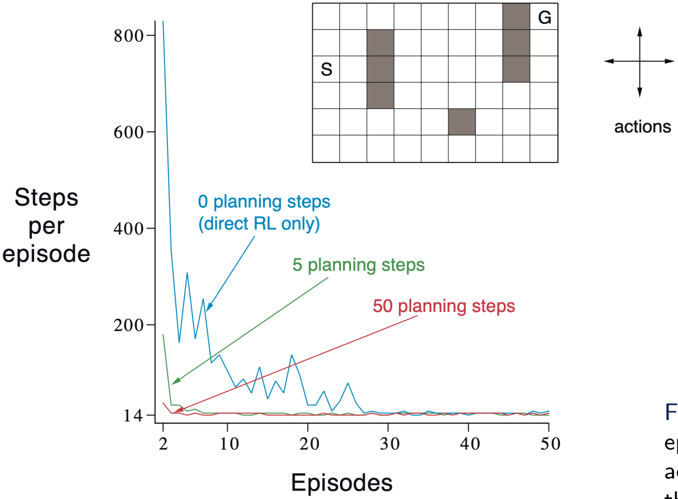
Dyna


value/policy
acting
planning
direct
RL
model
experience
model
learning

Dyna-Q Algorithm

- **Initialize** $Q(s, a)$ and **model**(s, a) for all s and a
- **do** forever:
  (a) $s \leftarrow$ current (non-terminal) state
  (b) $a \leftarrow \varepsilon$-greedy(s, Q)
  (c) Execute action a; observe reward r and next state s'
  (d) $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
  (e) **model**$(s, a) \leftarrow (r, s')$
  (f) Repeat n times:
      $s \leftarrow$ random previously observed state
      $a \leftarrow$ random action previously taking in s
      $(r, s') \sim$ **model**(s, a)
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

Dyna-Q on a Simple Maze



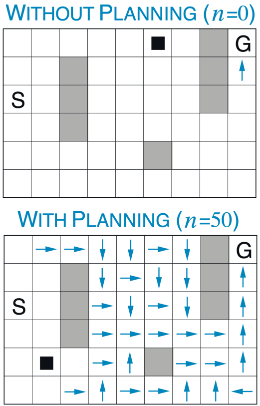


Figure: Policies found through 2nd episode. The arrows indicate greedy action; if no arrow is shown for a state, then all of its action values were equal.

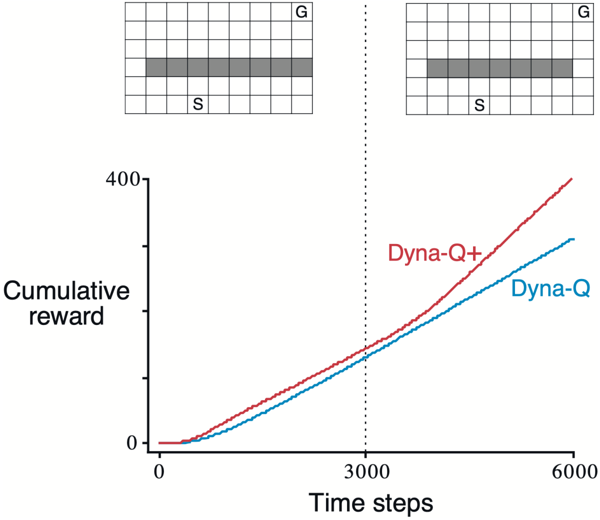
Dyna-Q$^+$

- **Motivation**: models maybe **incorrect**; leads to sub-optimal policies
  - **Limited** sample size for a given state-action pair
  - the environment **changes** and new behavior has not been observed
- **Idea**: encourage **long-untried** actions
  - For each state–action pair, check how many times have **elapsed** since it was last tried
  - Use **bonus reward** in action selection:

$$a \leftarrow \varepsilon - \mathrm{greedy}(s, Q + \kappa\sqrt{\tau}),$$

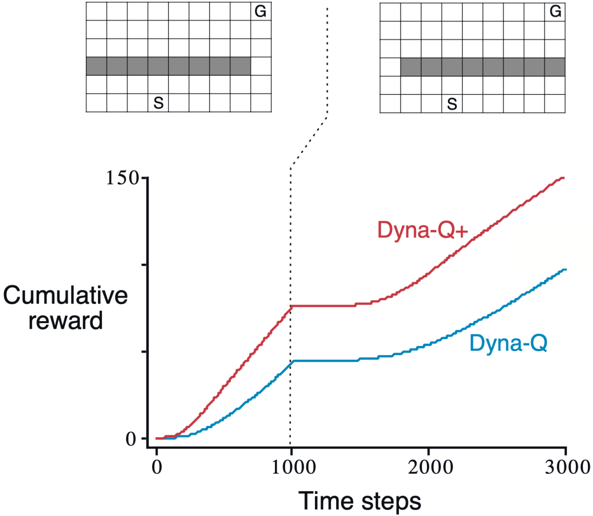
for some small $\kappa > 0$. $\tau(s, a)$ denotes the times have elapsed since (s, a) was last tried

Dyna-Q with an Inaccurate Model



- The changed environment is **easier**
- The left environment was used for the first 3000 steps
- The right environment was used for the rest

Dyna-Q with an Inaccurate Model (Cont'd)



- The changed environment is **harder**
- The left environment was used for the first 1000 steps
- The right environment was used for the rest

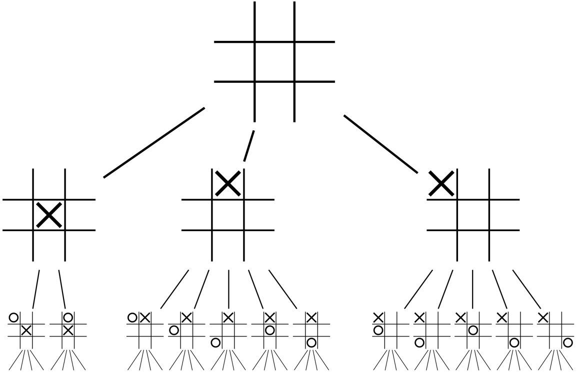
Model-based Methods

- First, we learn a **model** (reward and state transition functions) based on data
- Next, we can implement **planning** based on the learned model
- Alternatively, we can **integrate planning with learning** (Dyna)
- Finally, we can implement **Monte Carlo tree search** for decision-time planning

Two Ways of Planning

- **Background planning**
  - Planning is used well **before** an action is selected
  - Need to select actions fo each state, not **current** state
  - Examples: policy iteration and value iteration in Lecture 3
- **Decision-time planning**
  - Planning is started and completed **after** encountering each new state S_t
  - As a computation to **determine** A_t
  - On the next step planning begins anew with S_{t+1} to produce A_{t+1}, and so no

Game Trees



- Game trees: data structures to represent a game
- Exhaustive search can be **computationally intensive**
- Solutions bought by **Monte Carlo tree search**

Monte-Carlo Tree Search (Evaluation)

- Given a model $\mathcal{M}$
- Simulate K episodes from current states S_t using current policy π

$$\left\{S_t, A_t^k, R_t^k, S_{t+1}^k, A_{t+1}^k, R_{t+1}^k, \cdots, S_T^k\right\}_{k=1}^{K} \sim \mathcal{M}, \pi$$

- Build a search tree containing visited states and actions
- Evaluate states $Q(s, a)$ by mean return of episodes from s, a

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbb{I}(S_u = s, A_u = a) G_u \rightarrow Q^{\pi}(s, a)$$

- After search is finished, select current action with maximum value in search tree

$$A_t = \underset{a \in \mathcal{A}}{\arg\max}\, Q(S_t, a)$$

Monte-Carlo Tree Search (Simulation)

- In MCTS, the simulation policy (rollout policy) π that simulates data **improves**
- Repeat (each simulation)
  - **Evaluate** states $Q(s, a)$ by Monte-Carlo evaluation
  - **Improve** simulation policy, e.g., by ε-greedy(Q)
  - **Monte-Carlo control** applied to **simulated experience**
- Converges to the optimal search tree, $Q(s, a) \rightarrow Q^{\pi^{\text{opt}}}(s, a)$

Lecture Outline

1. What is Model-based RL
2. How to Implement Model-based RL
3. **Mastering the Game of Go**

Case Study: the Game of Go

- Invented in China over 2500 years ago
- The **hardest** classic board game
- Much harder than chess:
  - Go has larger number of legal moves than chess ($\approx$250 v.s. $\approx$35)
  - Go involve more moves than chess ($\approx$ 150 v.s. $\approx$ 80)
  - Traditional game-tree search fails in Go

Rules of Go

- Two players place down white and black stones **alternately**
- Stones are **captured** according to simple rules

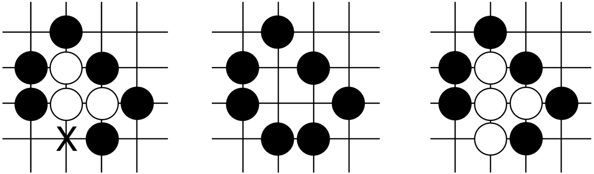


Figure: Left: the three white stones are not surrounded because point X is unoccupied. Middle: if black places a stone on X, the three white stones are captured and removed from the board. Right: if white places a stone on point X first, the capture is blocked.

- The game ends when neither player wishes to place another stone
- The player with more **territory** wins the game

Two-Player Zero-Sum Markov Games

- Simplest extension of MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{B}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
- $\mathcal{A}$ and $\mathcal{B}$ are actions spaces of first and second players
- $\mathcal{R}$ is reward function. In Go,
  - $R_t = 0$ for all non-terminal steps
  - $R_T = 1$ if Black wins and -1 otherwise
- Let π and ν be policies of the first and second players
- The state-value function depends on both π and ν

$$V^{\pi,\nu}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \middle| S_0 = s, A_t \sim \pi, B_t \sim \nu\right]$$

Nash Equilibrium

- At each state s, the two players aim to solve two **minimax** problems

$$\arg\max_{\pi} V^{\pi}(s) = \arg\max_{\pi}\min_{\nu} V^{\pi,\nu}(s)$$
$$\arg\min_{\nu} V^{\nu}(s) = \arg\min_{\nu}\max_{\pi} V^{\pi,\nu}(s)$$

- Under **Markov** and **time-homogeneity** assumptions, there exist stationary policies π^* (ν^*) whose values are no worse (better) than any history dependent policy

$$V^{\pi^*,\nu^*}(s) = \arg\max_{\pi}\min_{\nu} V^{\pi,\nu}(s) = \arg\min_{\nu}\max_{\pi} V^{\pi,\nu}(s)$$

  similar to the existence of the optimal stationary policy theorem in Lecture 2
- These policies reach a **Nash equilibrium** [Morgenstern and Von Neumann, 1953], i.e., no player can play better by changing his/her own policy

Prisoner's Dilemma

	Prisoner B stays silent (cooperates)	**Prisoner B betrays (defects)**
Prisoner A stays silent (cooperates)	Each serves 1 year	Prisoner A: 3 years Prisoner B: goes free
Prisoner A betrays (defects)	Prisoner A: goes free Prisoner B: 3 years	Each serves 2 years

Mutual defection is the only Nash equilibrium

Bellman Optimally Equation

- Bellman optimal equation for the **state value**

$$V^{\pi^*,\nu^*}(S_t) = \max_{a} \min_{b} \mathbb{E}\left[R_t + \gamma V^{\pi^*,\nu^*}(S_{t+1}) \middle| S_t, A_t = a, B_t = b \right]$$

- Bellman optimal equation for the **state-action value**

$$Q^{\pi^*,\nu^*}(S_t, A_t, B_t) = \mathbb{E}\left[R_t + \gamma \max_{a} \min_{b} Q^{\pi^*,\nu^*}(S_{t+1}, a, b) \middle| S_t, A_t = a, B_t = b \right]$$

- The values can be learned similarly to standard TD/Q learning algorithms for MDP [see e.g., Fan et al., 2020]

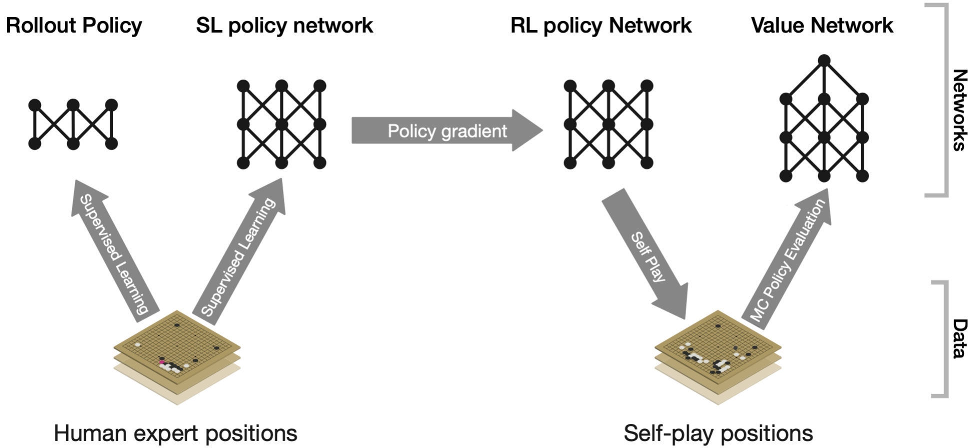
AlphaGo


THE ULTIMATE GO CHALLENGE
GAME 3 OF 3
27 MAY 2017
VS
AlphaGo
Winner of Match 3
Ke Jie
RESULT B + Res

AlphaGo Pipeline

- Based on a novel version of **Monte-Carlo tree search** (MCTS)
- Combined with a **policy** and a **value function** learned by RL with function approximation provided by deep CNN
- Simulate trajectories and generate the search tree using the **rollout** policy
- **Expand** search tree by selecting unexplored actions according to a **policy network**
- Policy network trained previously via supervised learning to predict moves contained in a database of nearly 30 million human expert moves
- Evaluate state-action value based on simulated returns (MC) and a **value** network
- Value network trained previously via RL

AlphaGo Pipeline (Cont'd)



Input of Neural Networks

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Policy Network

- Training the **SL policy network** took approximately 3 weeks using distributed implementation of SGD on 50 processors
- The SL policy network achieved **57**% accuracy; best accuracy achieved by other methods **44**%
- The **RL policy network** is trained on a million games in a single day
- The final RL policy won more than **80**% of games played against the SL policy
- It won **85**% of games played against a Go program using MCTS that simulated 100,000 games per move

Value Network

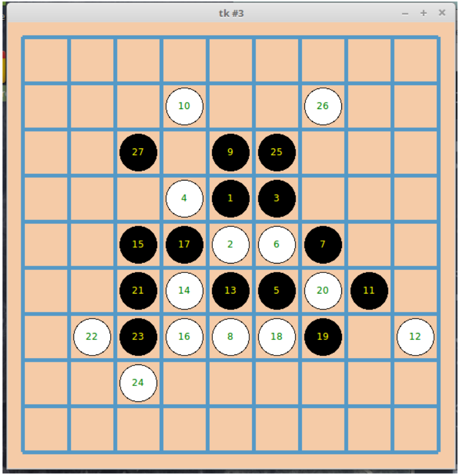
- The **value network** used **Monte Carlo policy evaluation** based on data obtained from a large number of self-play games played using the RL policy
- To avoid overfitting and instability, and to reduce the strong correlations between positions encountered in self-play, the dataset consists of **30** million positions, each chosen randomly from a unique self-play game
- Training was done using **50** million mini-batches each of **32** positions drawn from this data set
- Training took one week on **50** GPUs

Rollout Policy

- The **rollout policy** was learned prior to play by a simple linear network trained by supervised learning from a corpus of **8** million human moves
- In principle, the SL or RL policy networks could have been used in the rollouts, but the forward propagation through these deep networks took **too much time** for either of them to be used in rollout simulations
- The rollout policy network allowed approximately 1,000 complete game simulations per second to be run on each of the processing threads

AlphaGo Zero on Gomoku

`https://github.com/initial-h/AlphaZero_Gomoku_MPI`

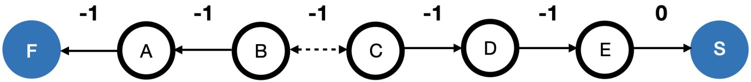


Summary

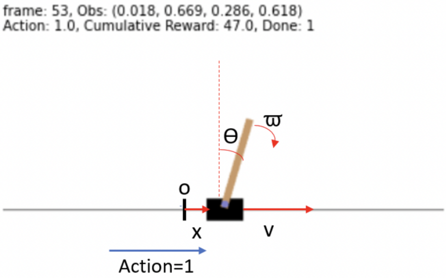
- Model-based/Model free learning
- Integrating planning and learning
- Dyna-Q/Dyna-Q^+
- Simulation-based search
- Background/Decision-time planning
- Monte Carlo Tree Search
- Two-player zero-sum Markov games
- Nash equilibrium
- AlphaGo

Seminar Exercise

- Solution to HW8 (Deadline: Wed 12:00 pm)



- Advantage Actor-Critic (with deep neural networks) to CartPole



- Implementation of Dyna-Q algorithm

References I

Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for Dynamics and Control*, pages 486–489. PMLR, 2020.

Oskar Morgenstern and John Von Neumann. *Theory of games and economic behavior*. Princeton university press, 1953.

Jonas Rothfuss, Fabio Ferreira, Simon Walther, and Maxim Ulrich. Conditional density estimation with neural networks: Best practices and benchmarks. *arXiv preprint arXiv:1903.00954*, 2019.

Brian L Trippe and Richard E Turner. Conditional density estimation with bayesian normalising flows. *arXiv preprint arXiv:1802.04908*, 2018.

Christopher K Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

Questions