Lecture Outline

- Planning in reinforcement learning
- Dyna-Q
- Rollout planning
- Monte Carlo tree search
- Offline vs online planning
Unified View

Planning: Using a model
Planning: any process which uses a model of the environment to compute a plan of action (policy) to achieve a specified goal

- Dynamic programming is planning: uses model $p(s', r|s, a)$
Model: anything the agent can use to predict how environment will respond to actions
**Model**: anything the agent can use to predict how environment will respond to actions

- **Distribution model**: description of all possibilities and their probabilities

\[
p(s', r | s, a) \quad \text{for all } s, a, s', r
\]
**Model:** anything the agent can use to predict how environment will respond to actions

- **Distribution model:** description of all possibilities and their probabilities

  \[ p(s', r | s, a) \text{ for all } s, a, s', r \]

- **Simulation (sample) model:** produces sample outcomes

  \[(s', r) \sim \hat{p}(s, a) \text{ s.t. } \Pr\{\hat{p}(s, a) = (s', r)\} = p(s', r | s, a)\]

  Simulation model usually easier to specify than distribution model
Paths to a Policy: Model-Free RL

Model-free RL

Environmental interaction → Experience → Value function → Policy

Direct RL methods
Model-based RL

Paths to a Policy: Model-Based RL
Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in S$ and $a \in A(s)$

Do forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy($S, Q$)
(c) Execute action $A$; observe resultant reward, $R$, and state, $S'$
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat $n$ times:
   $S \leftarrow$ random previously observed state
   $A \leftarrow$ random action previously taken in $S$
   $R, S' \leftarrow Model(S, A)$
   $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
Dyna-Q in Maze Example

\[ \gamma = 0.95 \]
\[ \epsilon = 0.1 \]
\[ \alpha = 0.1 \]

Steps per episode

0 planning steps (direct RL only)
5 planning steps
50 planning steps
Greedy policy halfway through second episode:

**WITHOUT PLANNING (n=0)**

**WITH PLANNING (n=50)**
When the Model is Wrong: Blocking Maze

Cumulative reward

Time steps

Dyna-Q+

Dyna-Q
When the Model is Wrong: Shortcut Maze

Cumulative reward vs. Time steps for Dyna-Q and Dyna-Q+.
Dyna-Q+ uses an *exploration bonus* heuristic:

- Keeps track of time since each state-action pair was tried in real environment

- Bonus reward is added for transitions caused by state-action pairs related to how long ago they were tried:

\[ R + \kappa \sqrt{\tau} \]

  time since last visiting the state-action pair

- Incentive to re-visit “old” state-action pairs
Dyna-Q uses model to reuse past experiences

Rollout planning:

- Use model to simulate ("rollout") future trajectories
- Each trajectory starts at current state $S_t$
- Find best action $A_t$ for state $S_t$
Rollout Q-planning with forward updating:

1. Given: simulation model \( Model \)
2. Initialise: \( Q(s, a) \) for all \( s, a \)
3. \textbf{for} \( t = 0, 1, 2, 3, \ldots \) \textbf{do}
4. \( S_t \leftarrow \text{current state} \)
5. \textbf{for} \( n \) rollouts \textbf{do}
6. \( S \leftarrow S_t \)
7. \textbf{while} \( S \) is non-terminal (or fixed-length rollouts) \textbf{do}
8. select action \( A \) based on \( Q(S, \cdot) \) with some exploration // e.g. \( \epsilon \)-greedy
9. \( (R, S') \sim Model(S, A) \)
10. Q-update: \( Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)] \)
11. \( S \leftarrow S' \)
12. select action \( A_t \) greedily from \( Q(S_t, \cdot) \)
If model is **correct** and under Q-learning conditions (all \((s, a)\) infinitely visited and standard \(\alpha\)-reduction), rollout planning learns optimal policy.
If model is **correct** and under Q-learning conditions (all \((s, a)\) infinitely visited and standard \(\alpha\)-reduction), rollout planning learns *optimal policy*

If model is **incorrect**, learned policy likely sub-optimal on real task

- Can range from slightly sub-optimal to failing to solve real task (examples?)
If model is **correct** and under Q-learning conditions (all \((s, a)\) infinitely visited and standard \(\alpha\)-reduction), rollout planning learns *optimal policy*

If model is **incorrect**, learned policy likely sub-optimal on real task
- Can range from slightly sub-optimal to failing to solve real task (examples?)

Next: can we use rewards from rollouts more effectively?

⇒ Back-propagate rewards
Rollout Planning with Backward Updating (Back-Propagation)

Rollout Q-planning with backward updating:
1: Given: simulation model \textit{Model}
2: Initialise: \( Q(s, a) \) for all \( s, a \); LIFO stack \( \text{Trace} = \{\} \)
3: \textbf{for} \( t = 0, 1, 2, 3, \ldots \) \textbf{do}
4: \( S_t \leftarrow \) current state
5: \textbf{for} \( n \) rollouts \textbf{do}
6: \( S \leftarrow S_t \)
7: \textbf{while} \( S \) is non-terminal (or fixed-length rollouts) \textbf{do} \hspace{1em} // Rollout
8: \hspace{1em} select action \( A \) based on \( Q(S, \cdot) \) with some exploration
9: \hspace{1.5em} \( (R, S') \sim \text{Model}(S, A) \)
10: \hspace{1em} push \( (S, A, R, S') \) to \( \text{Trace} \)
11: \hspace{1em} \( S \leftarrow S' \)
12: \textbf{while} \( \text{Trace} \) not empty \textbf{do} \hspace{1em} // Backprop
13: \hspace{1.5em} pop \( (S, A, R, S') \) from \( \text{Trace} \)
14: \hspace{2em} \( Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)] \)
15: \hspace{1.5em} select action \( A_t \) greedily from \( Q(S_t, \cdot) \)
Rollout Planners in Maze Example

Steps to goal state vs. Episode

- Forward, 5 rollouts, length 5
- Forward, 5 rollouts, length 10
- Forward, 10 rollouts, length 20
- Backward, 5 rollouts, length 5
- Backward, 5 rollouts, length 10
- Backward, 10 rollouts, length 20

\[ \gamma = 0.95 \]
\[ \epsilon = 0.1 \]
\[ \alpha = 0.1 \]
Monte Carlo Tree Search (MCTS):

- General, efficient rollout planning with backward updating
- Stores partial $Q$ as tree and asymmetrically expands tree based on most promising actions

$Q$ is recursive tree structure:

$$Q(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') | S_t = a, A_t = a]$$
Phases of Monte Carlo Tree Search

Selection → Expansion → Simulation → Backpropagation

Tree Policy

Default Policy

Browne et al. (2012)
General MCTS Method

MCTS-Search($S_t$):

1. Find node $v_0$ with $\text{state}(v_0) = S_t$ (or create new node)

2. while within computational budget do

3. $v_I \leftarrow \text{TreePolicy}(v_0)$ // Select node in tree and expand

4. $\Delta \leftarrow \text{DefaultPolicy}(\text{state}(v_I))$ // Simulation steps

5. $\text{Backprop}(v_I, \Delta)$

6. return $\text{action}(\text{BestChild}(v_0))$ // e.g. highest expected return; most visited child

- Tree policy can be any exploration policy
- Backprop works just as before
Upper Confidence Bounds for Trees (UCT):

- Popular MCTS variant — easy to use and often effective
- Uses UCB action selection as tree policy, and \( \alpha = 1/N(S, A) \)

UCB recap: estimate upper bound on action value:

\[
A \leftarrow \begin{cases} 
    a, & \text{if } a \text{ never tried in } S \\
    \arg \max_a Q(S, a) + c \sqrt{\log N(S)/N(S, a)} & \text{otherwise}
\end{cases}
\]

- \( N(S) \) is number of times state \( S \) has been visited
- \( N(S, a) \) is number of times action \( a \) was selected in \( S \)
Simulation Step

Simulation step gives estimate of return at state, e.g.:

Random-DefaultPolicy(S):

1: $G \leftarrow 0$
2: while $S$ is non-terminal do
3: $A \leftarrow$ random action (uniformly)
4: $(R, S') \sim Model(S, A)$
5: $G \leftarrow R + \gamma G$
6: $S \leftarrow S'$
7: return $G$

Possible improvements:

- Average over multiple simulations
- Use domain-specific heuristic to
  - select better actions than random
  - evaluate state directly (e.g. in Chess we know that some states are better than others)
Imagine you are given an MDP for a chess game against a specific opponent

**Offline planning:**

- Use MDP to find best policy **before** the actual chess game takes place (offline)
- Use as much time as needed to find policy
- Policy is **complete**: gives optimal action for all possible states

Dyna-Q and dynamic programming are suitable for offline planning
Online Planning

Imagine you are given an MDP for a chess game against a specific opponent

**Online planning:**

- Use MDP to find best policy *during* the actual chess game (online)
- Limited compute time budget at each state (e.g. seconds/minutes in chess)
- Policy usually *incomplete*: gives optimal action for *current* state

Rollout planning (including MCTS) is suitable for online planning
Paths to a Policy: Model-Based RL

1. Experience
2. Model
   - Model learning
3. Simulation
4. Direct planning
5. Value function
   - Direct RL methods
6. Policy

Environmental interaction
Required:

- RL book, Chapter 8 (8.1–8.3, 8.10–8.11)

Optional:

- Browne et al. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games, Vol. 4, No. 1