Lecture 6: Planning and Learning

Friday, December 3, 2021

Reinforcement Learning, Winter Term 2021/22

Joschka Boedecker, Gabriel Kalweit, Jasper Hoffmann

Neurorobotics Lab
University of Freiburg

Lecture Overview

Lecture Overview

1. Recap
2. Model Learning
3. Dyna
4. Simulation-based Search
5. Wrapup

Recap

- TD is a combination of Monte Carlo and dynamic programming ideas
- Similar to MC methods, TD methods learn directly raw experiences without a dynamic model
- TD learns from *incomplete* episodes by bootstrapping
- Bootstrapping: update estimated based on other estimates without waiting for a final outcome (update a guess towards a guess)

Simplest temporal-difference learning algorithm: $TD(0)$

Update value $V(S_t)$ towards the *estimated* return $R_{t+1} + \gamma V(S_{t+1})$.

$$V(s_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

- $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*

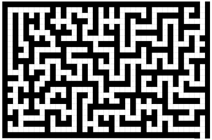
Lecture Overview

Components of RL Systems

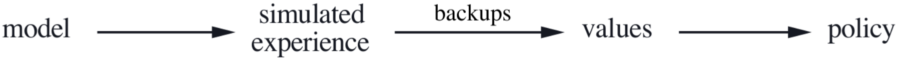
- Policy: defines the behaviour of the agent
  - is a mapping from a state to an action
  - can be stochastic: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$
  - or deterministic: $\pi(s) = a$
- Value-function: defines the expected value of a state or an action
  - $v_\pi(s) = \mathbb{E}[G_t|S_t = s]$ and $q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a]$
  - can be used to evaluate states or to extract a good policy
- Model: defines the transitions between states in an environment
  - p yields the next state and reward
  - $p(s', r|s, a) = \mathsf{Pr}\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\}$

Learning Models

- Depending on the task, the dynamics model can be much easier than the value-function or the policy
- We can estimate it via supervised learning methods
- However, the model can also be more complex than policy and value-function
- In practice, modelling state-changes can even be easier than the global state
- In a nutshell:
  - Learning a model: data-efficient, hard to extract an optimal policy
  - Learning a value function: less data-efficient, easier to extract an optimal policy
  - Learning a policy: data-inefficient, directly estimate an optimal policy



- Given a state and an action, a model generates the next state and the corresponding reward (can also be used to generate sequences of states and rewards)
- It can either give the probabilities of all possible next states and rewards (*distribution model*), or only one (*sample model*)
- Which one was used in Dynamic Programming?
- Extracting a policy from a model is called *planning*
- Here: *state-space planning*



Models and Planning

- Planning: Uses simulated experience generated by a model
- Learning: Uses real experiences from the environment
- But we can also apply learning methods to simulated experience

Random-sample one-step tabular Q-planning

Loop forever:
1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$

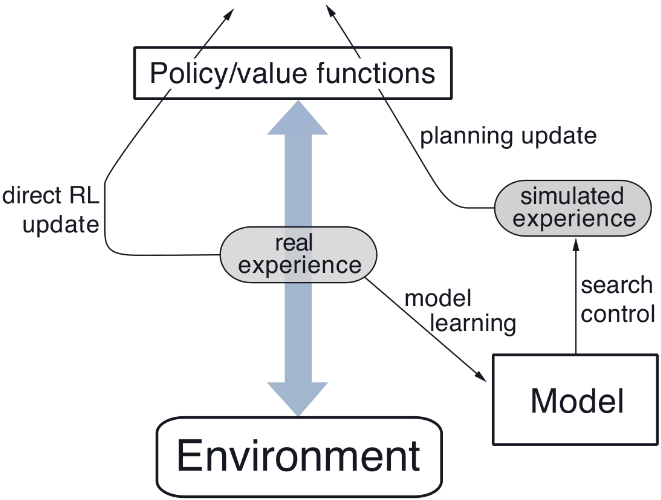
- Converges to the optimal policy *for the model*

Lecture Overview

Dyna

- Real experience can be used to optimize the value function (or the policy)
  - *directly* (model-free RL) or
  - *indirectly* (model-based RL) via a model
- *Indirect methods* are often more data-efficient
- But they introduce additional bias through the model
- Idea of Dyna: try to combine the best of both worlds

Dyna

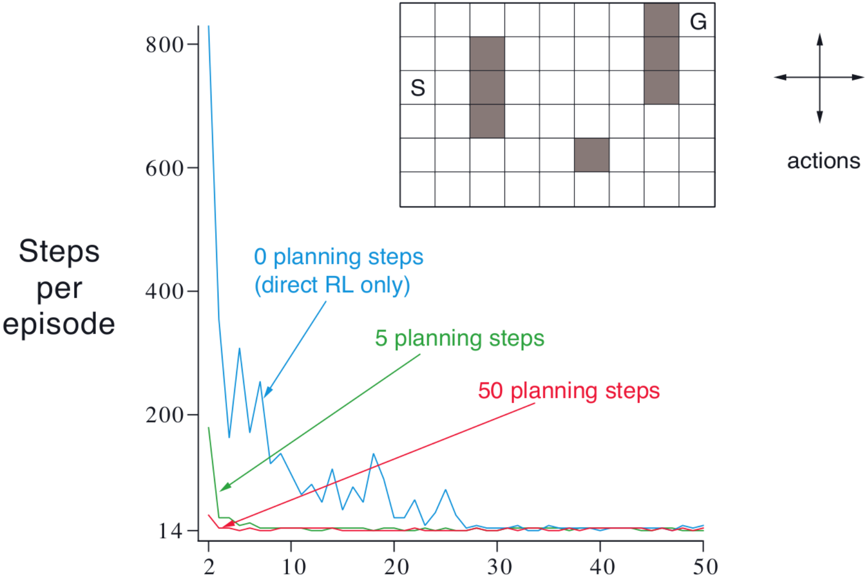

Policy/value functions
planning update
direct RL
update
simulated
experience
real
experience
model
learning
search
control
Model
Environment

Dyna

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon$-greedy(S,Q)
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) Loop repeat n times:
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Dyna



Steps per episode
800
600
400
200
14
2
10
20
30
40
50
0 planning steps (direct RL only)
5 planning steps
50 planning steps
S
G
actions

When the model is wrong

- Models can be incorrect (limited number of samples, environment has changed, function approximation)
- Especially in areas where the agent has not explored
- There can be a *Distribution Mismatch* when the agent enters new areas of the state-action space
- When the model is incorrect, the planning process is likely to find a suboptimal policy

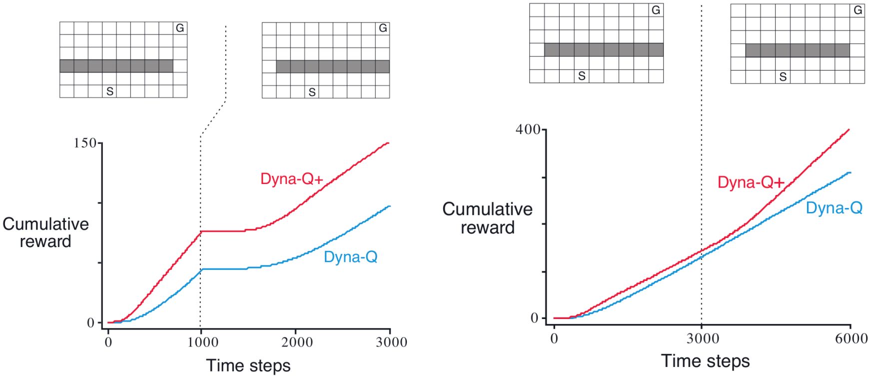
When the model is wrong

- Dyna-Q+: Add an exploration bonus for transitions that have not been visited recently
- Let r be the reward, κ the weight of the exploration bonus and τ the number of time steps in which a certain transition has not been visited
- Then Dyna-Q+ modifies the internal reward function to:

$$r + \kappa\sqrt{\tau}$$

- To which exploration technique from Lecture 01 (Bandits) does this share great similarity?

When the model is wrong



Prioritized Sweeping

- Update action-values for state-action pairs with high priority
- Here: we want to work back from states whose values have changed

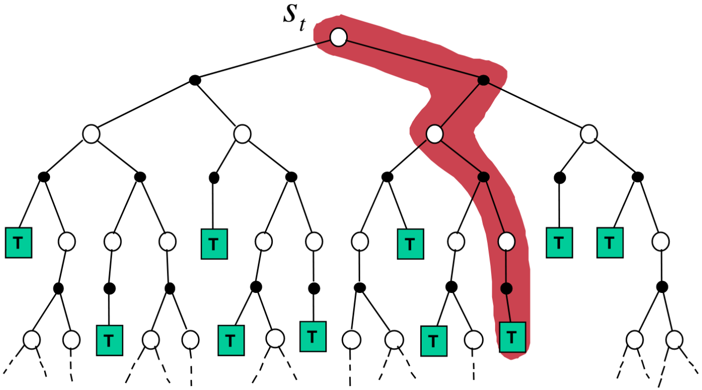
Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow policy(S,Q)$
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Model(S,A) \leftarrow R, S'$
  (e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
  (f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
  (g) Loop repeat n times, while $PQueue$ is not empty:
    $S, A \leftarrow first(PQueue)$
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
      $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
      $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
      if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

Lecture Overview

Simulation-based Search

- Forward search paradigm using sample-based planning
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



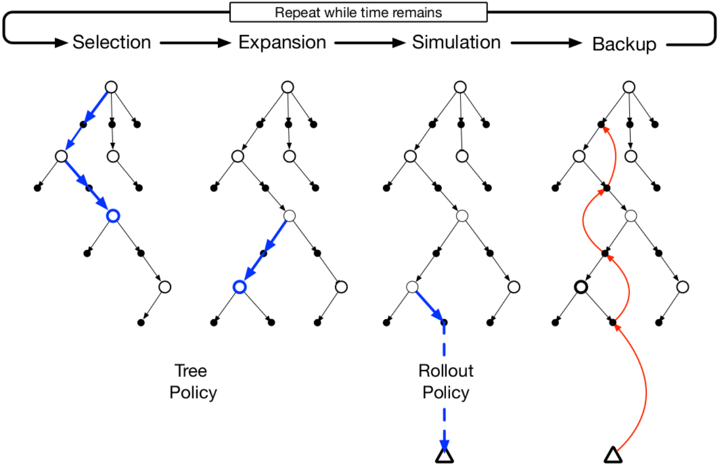
Monte Carlo Tree Search

- Build a search tree containing visited states and actions using the model (simulate episodes from current state)
- Two policies: tree policy (improving, e.g. ϵ-greedy) and out-of-tree rollout policy (random)
- Monte-Carlo control applied to simulated experience
- One of the key ingredients of AlphaGo (2016)

Monte Carlo Tree Search

1. Selection: starting at the root, traverse to a leaf node following the *tree policy*
2. Expansion: expand the tree by one or multiple child nodes reached from the selected leaf in some iterations
3. Simulation: simulate an episode following the *rollout policy*
4. Backup: update the action-values for all nodes visited **in the tree**

Monte Carlo Tree Search



Temporal-Difference Search

- Simulation-based search
- Using TD instead of MC
- MCTS applies MC control to sub-MDP from now
- TD search applies SARSA to sub-MDP from now

Lecture Overview

Summary by Learning Goals

Having heard this lecture, you can now...

- Explain the difference between model-based and model-free RL
- Explain how to make use of models in RL
- Explain the Dyna-architecture
- Explain simulation-based forward search and its variants MCTS and TD Search