Reinforcement Learning

Planning and Learning

Stefano Albrecht, Pavlos Andreadis

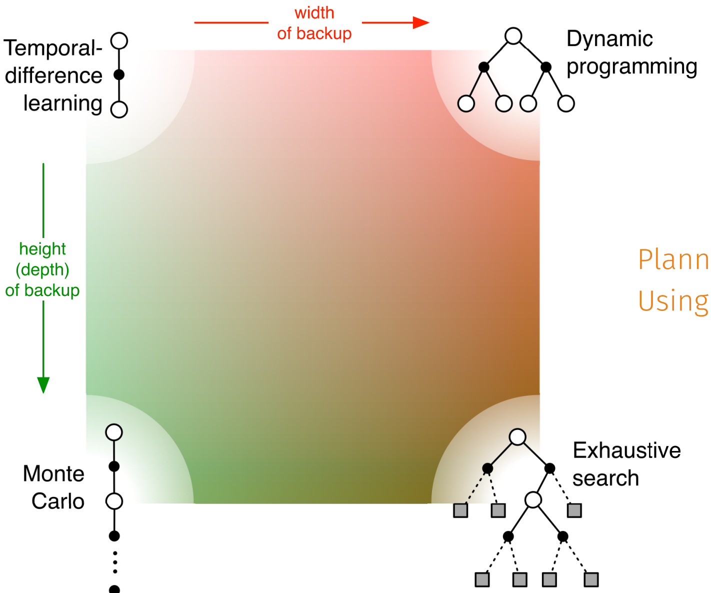
4 February 2020

THE UNIVERSITY of EDINBURGH
informatics

Lecture Outline

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

Unified View

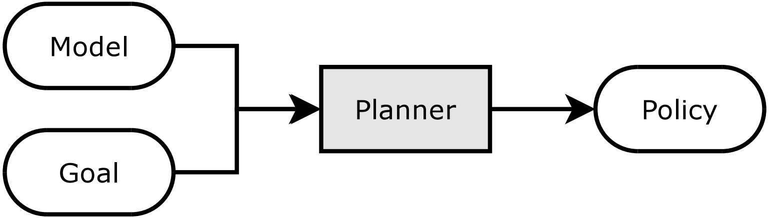


Planning:

Using a model

Planning

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

Model: anything the agent can use to predict how the environment will respond to its actions

- Distribution model: description of all possibilities and their probabilities

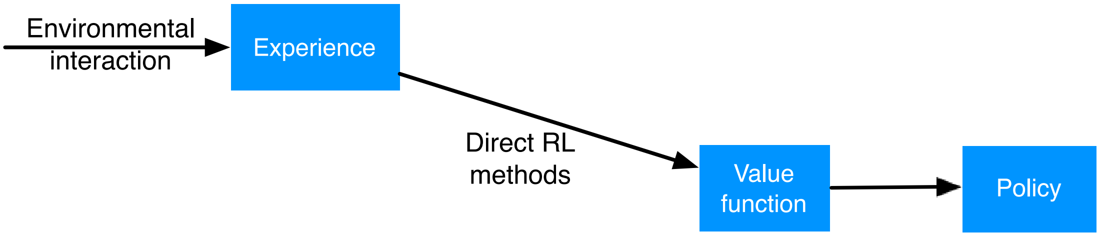
$$p(s', r|s, a) \quad \text{for all} \quad s, a, s', r$$

- Simulation (sample) model: produces sample outcomes

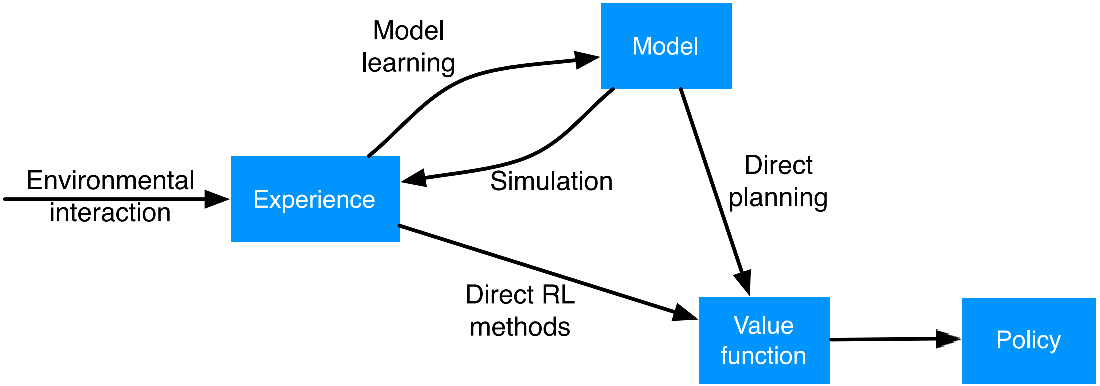
$$\hat{p}(s, a) \to (\mathcal{S}, \mathcal{R}) \quad \text{s.t.} \quad \Pr\{\hat{p}(s, a) = (s', r)\} = p(s', r|s, a)$$

- Simulation model usually easier to specify than distribution model

Model-free RL



Model-based RL

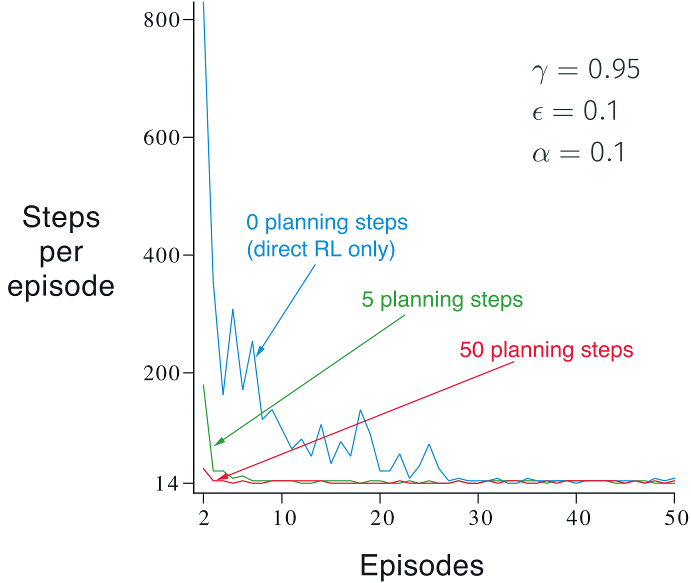


Dyna-Q: Integrating Planning, Learning, Acting

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ ← **direct RL**
(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment) ← **model learning**
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

← **planning**

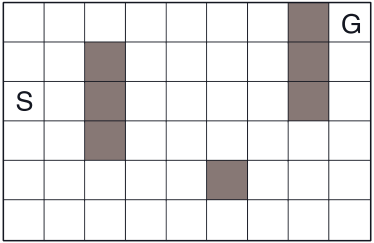
Dyna-Q in Maze Example



$\gamma = 0.95$

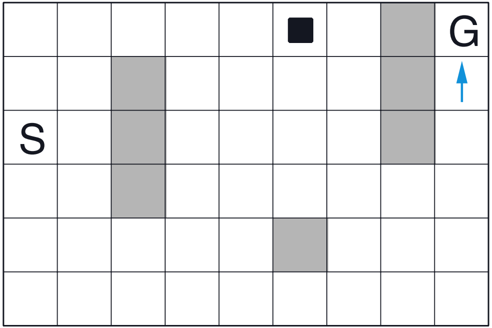
$\epsilon = 0.1$

$\alpha = 0.1$

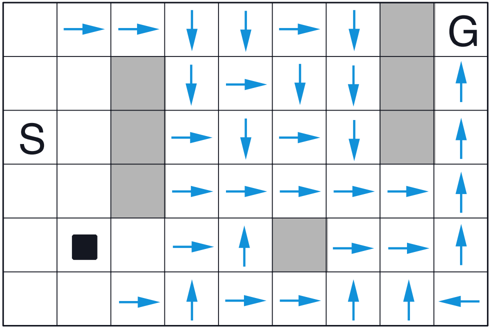


Greedy policy halfway through second episode:

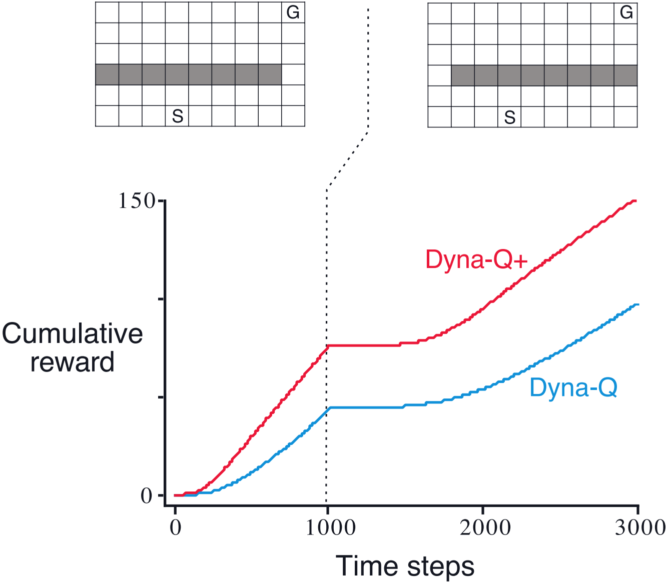
WITHOUT PLANNING (n=0)



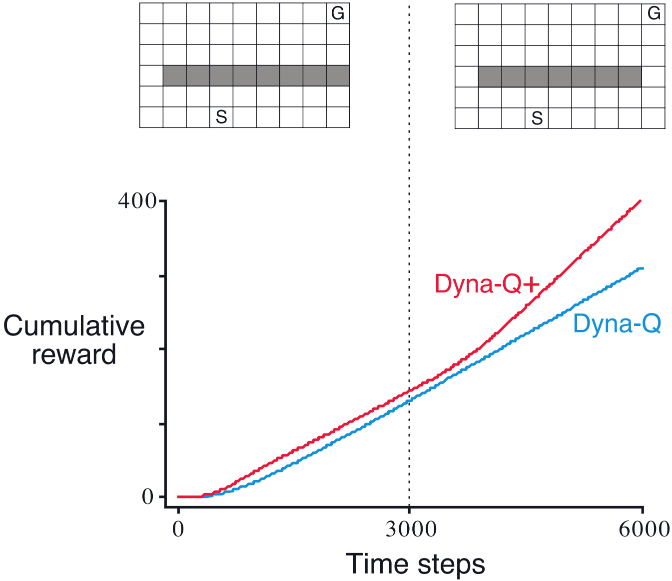
WITH PLANNING (n=50)



When the Model is Wrong: Blocking Maze



When the Model is Wrong: Shortcut Maze



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

Rollout Planning

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
- Focus usually on finding best action A_t for state S_t

Rollout Planning with Forward Updating

Rollout Q-planning with forward updating:

Given: simulation model *Model*
Initialise: $Q(s, a)$ for all s, a
for $t = 0, 1, 2, 3, \ldots$ **do**
  $S_t \leftarrow$ current state
  for n times (n rollouts) **do**
    $S \leftarrow S_t$
    while S is non-terminal (or fixed-length rollouts) **do**
      select action A based on $Q(S, \cdot)$ with some exploration // *e.g.* ϵ*-greedy*
      $(R, S') \sim Model(S, A)$
      Q-update: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
      $S \leftarrow S'$
  select action A_t greedily from $Q(S_t, \cdot)$

Rollout Planning

If model **correct** and under Q-learning conditions (all (s, a) infinitely visited and standard α-reduction), rollout planning learns optimal policy

If model **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?

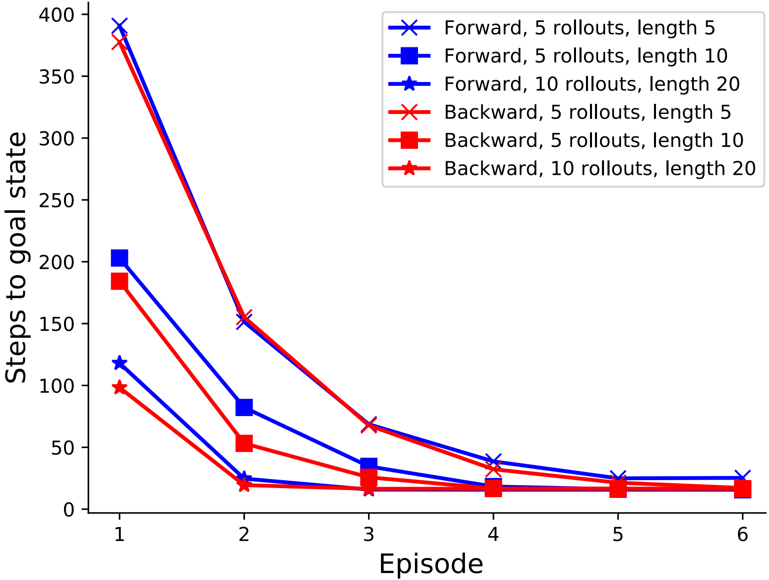
$\Rightarrow$ Back-propagate rewards

Rollout Planning with Backward Updating (Back-Propagation)

Rollout Q-planning with backward updating:

Given: simulation model *Model*
Initialise: $Q(s, a)$ for all s, a; LIFO stack $Trace = \{\}$
for $t = 0, 1, 2, 3, \ldots$ **do**
  $S_t \leftarrow$ current state
  for n times (n rollouts) **do**
    $S \leftarrow S_t$
    while S is non-terminal (or fixed-length rollouts) **do** *// Rollout*
      select action A based on $Q(S, \cdot)$ with some exploration
      $(R, S') \sim Model(S, A)$
      push (S, A, R, S') to $Trace$
      $S \leftarrow S'$
    while $Trace$ not empty **do** *// Backprop*
      pop (S, A, R, S') from $Trace$
      $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
  select action A_t greedily from $Q(S_t, \cdot)$

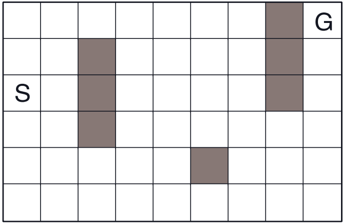
Rollout Planners in Maze Example



$\gamma = 0.95$

$\epsilon = 0.1$

$\alpha = 0.1$



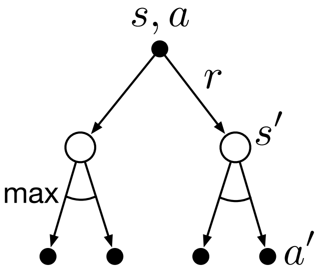
Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS):

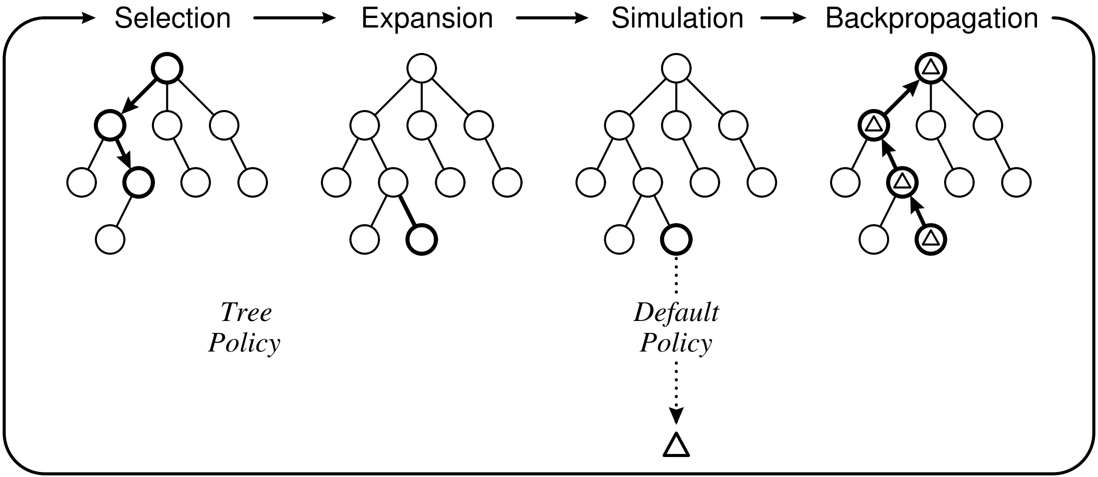
- General, efficient rollout planner
- Stores partial Q as tree and asymmetrically expands tree based on most promising actions

Q is recursive tree structure:

$$Q(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') \mid S_t = a, A_t = a]$$



Phases of Monte Carlo Tree Search



Browne et al. (2012)

General MCTS Method

MCTS-Search(S_t):

Find node v_0 with $state(v_0) = S_t$ (or create new node)
while within computational budget **do**
    $v_l \leftarrow TreePolicy(v_0)$
    $\Delta \leftarrow DefaultPolicy(state(v_l))$
    $Backprop(v_l, \Delta)$
return $action(BestChild(v_0))$ *// e.g. most visited child; highest expected return*

- Backprop works just as before
- Tree policy can be any exploration policy (balancing exploration and exploitation)

Upper Confidence Bounds for Trees

Upper Confidence Bounds for Trees (UCT):

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

UCB recap: estimate upper bound on action value:

$$A \leftarrow \begin{cases} a \text{ if not tried before in } S \\ \arg\max_a Q(S, a) + c\sqrt{\log N(S)/N(S, a)} \end{cases}$$

- $N(S)$ 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):

```
G ← 0
while S is non-terminal do
    A ← random action (uniformly)
    (R, S′) ~ Model(S, A)
    G ← R + γG
    S ← S′
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)

Offline Planning

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

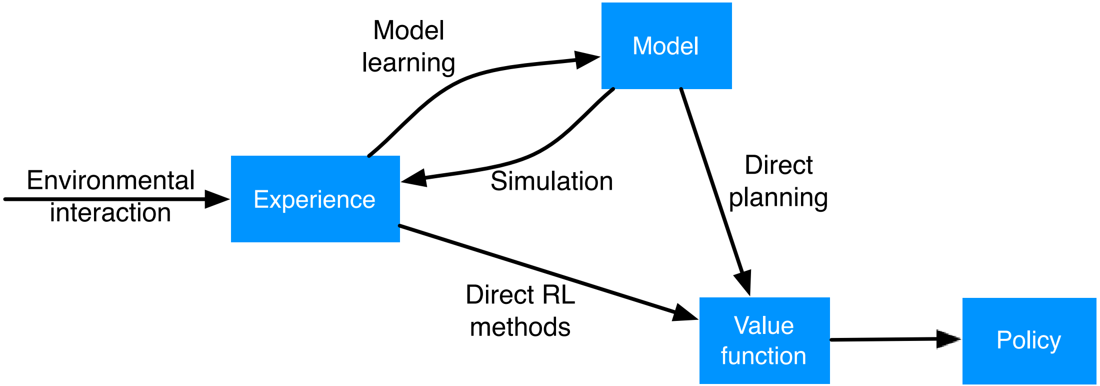
Imagine you are given an MDP for a chess game against a specific opponent:

Online planning:

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

Rollout planning (including MCTS) are suitable for online planning

Paths to a Policy: Model-Based RL



Reading

Required:

- RL book, chapter 8 (8.1–8.6, 8.8–8.11)

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
- UCT paper: L. Kocsis and C. Szepesvari (2006). Bandit based Monte-Carlo Planning. European Conference on Machine Learning
- T. Vodopivec, S. Samothrakis, B. Ster (2017). On Monte Carlo Tree Search and Reinforcement Learning. Journal of Artificial Intelligence Research, Vol. 60