Lecture 07: Planning and Learning with Tabular Methods

Oliver Wallscheid

Recap: RL Agent Taxonomy

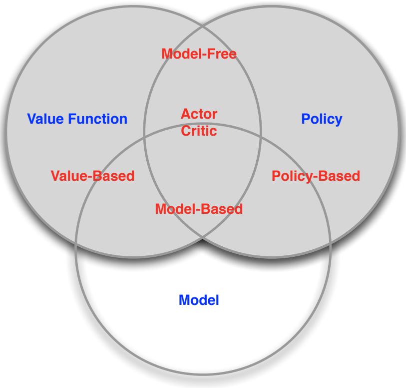


Fig. 7.1: Main categories of reinforcement learning algorithms (source: D. Silver, Reinforcement learning, 2016. CC BY-NC 4.0)

- Up to now: independent usage of model-free and model-based RL
- Today: integrating both strategies (on finite state & action spaces)

Table of Contents

Model-based RL

- Plan/predict value functions and/or policy from a model.
- Requires an a priori model or to learn a model from experience.
- Solves control problems by planning algorithms such as
  - Policy iteration or
  - Value iteration.

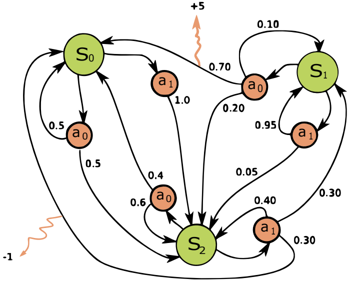


Fig. 7.2: A model for discrete state and action space problems is generally an MDP (source: www.wikipedia.org, by Waldoalvarez CC BY-SA 4.0)

What is a Model?

- A model $\mathcal{M}$ is an MDP tuple $\langle\mathcal{X},\mathcal{U},\boldsymbol{\mathcal{P}},\mathcal{R},\gamma\rangle$.
- In particular, we require the
  - state-transition probability

$$\boldsymbol{\mathcal{P}} = \mathbb{P}\left[\boldsymbol{X}_{k+1} = \boldsymbol{x}_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] \tag{7.1}$$

  - and the reward probability

$$\mathcal{R} = \mathbb{P}\left[R_{k+1} = r_{k+1} | \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] . \tag{7.2}$$

- State space $\mathcal{X}$ and action space $\mathcal{U}$ is assumed to be known.
- Discount factor γ might be given by environment or engineer's choice.
- What kind of model is available?
  - If $\mathcal{M}$ is perfectly known a priori: true MDP.
  - If $\hat{\mathcal{M}} \approx \mathcal{M}$ needs to be learned: approximated MDP.

Model Learning / Identification

- Ideally, the model $\mathcal{M}$ is exactly known a priori (e.g., gridworld case).
- On the contrary, a model might be too complex to derive or not exactly available (e.g., real physical systems).
  - Objective: estimate model $\hat{\mathcal{M}}$ from experience $\{X_0, U_0, R_1, \ldots, X_T\}$.
- This is a supervised learning / system identification task:

$$\{X_0, U_0\} \rightarrow \{X_1, R_1\}$$
$$\vdots$$
$$\{X_{T-1}, U_{T-1}\} \rightarrow \{X_T, R_T\}$$

- Simple tabular / look-up table approach (with $n(x, u)$ visit count):

$$\begin{aligned} \hat{p}^u_{xx'} &= \frac{1}{n(x,u)} \sum_{k=0}^{T} 1(X_{k+1} = x' | X_k = x, U_k = u), \\ \hat{\mathcal{R}}^u_x &= \frac{1}{n(x,u)} \sum_{k=0}^{T} 1(X_k = x | U_k = u) r_{k+1}. \end{aligned} \tag{7.3}$$

Distribution vs. Sample Models

- A model based on $\mathcal{P}$ and $\mathcal{R}$ is called a distribution model.
  - Contains descriptions of all possibilities by random distributions.
  - Has full explanatory power, but is still rather complex to obtain.
- Alternatively, use sample models to receive realization series.
  - Remember black jack examples: easy to sample by simulation but hard to model a full distributional MDP.

Fig. 7.3: Depending on the application distribution models are easily available or not (source: Josh Appel on Unsplash)

Model-free RL

- Learn value functions and/or policy directly from experience.
- Requires no model at all (policy can be considered an implicit model).
- Solves control problems by learning algorithms such as
  - Monte-Carlo,
  - Sarsa or
  - Q-learning.

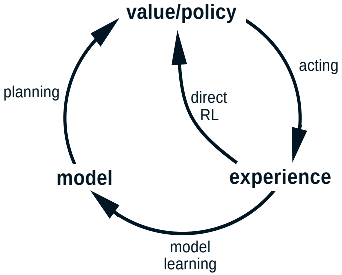


Fig. 7.4: If a perfect a priori model is not available, RL can be realized directly or indirectly (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Advantages & Drawbacks: Model-free vs. Model-based RL

Pro model-based / indirect RL:

- Efficiently uses limited amount of experience (e.g., by replay).
- Allows integration of available a priori knowledge.
- Learned models might be re-used for other tasks (e.g., monitoring).

Pro model-free / direct RL:

- Is simpler to implement (only one task, not two consequent ones).
- Not affected by model bias / error during model learning.

Table of Contents

The General Dyna Architecture (1)

- Proposed by R. Sutton in 1990's
- General framework with many different implementation variants

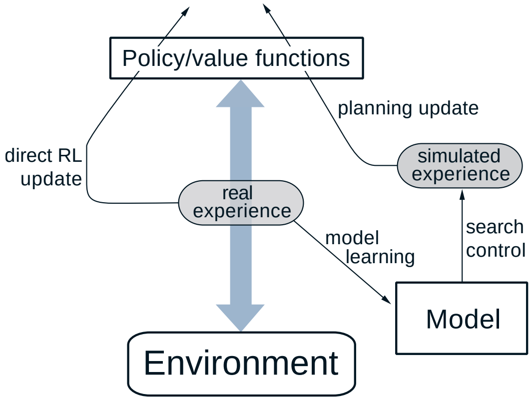
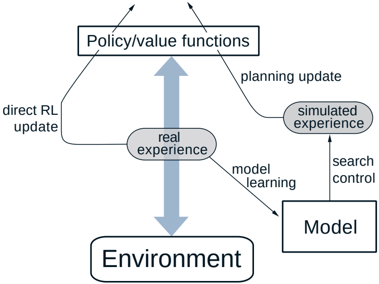


Fig. 7.6: Dyna framework (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

The General Dyna Architecture (2)

- Direct RL update: any model-free algorithm: Q-learning, Sarsa, ...
- Model learning:
  - In tabular case: simple distribution estimation as in (7.3)
  - Simple experience buffer to re-apply model-free algorithm
  - For large or continuous state/action spaces: function approximation by supervised learning / system identification (next lecture)
- Search control: strategies for selecting starting states and action to generate simulated experience



Algorithmic Implementation: Dyna-Q

parameter: $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $n \in \{\mathbb{N}|n \geq 1\}$ (planning steps per real step)
init: $\hat{q}(x, u)$ arbitrary (except terminal) and $\hat{\mathcal{M}}(x, u)$ $\forall$ $\{x \in \mathcal{X}, u \in \mathcal{U}\}$
for $j = 1, 2, \dots$ *episodes* **do**
  Initialize x_0;
  $k \leftarrow 0$;
  repeat
    Choose u_k from x_k using a soft policy derived from $\hat{q}(x, u)$;
    Take action u_k, observe r_{k+1} and x_{k+1};
    $\hat{q}(x_k, u_k) \leftarrow \hat{q}(x_k, u_k) + \alpha\left[r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k)\right]$;
    $\hat{\mathcal{M}}(x_k, u_k) \leftarrow \{r_{k+1}, x_{k+1}\}$ (assuming deterministic env.);
    for $i = 1, 2, \dots$ *n* **do**
      $\tilde{x}_i \leftarrow$ random previously visited state;
      $\tilde{u}_i \leftarrow$ random previously taken action in $\tilde{x}_i$;
      $\{\tilde{r}_{i+1}, \tilde{x}_{i+1}\} \leftarrow \hat{\mathcal{M}}(\tilde{x}_i, \tilde{u}_i)$;
      $\hat{q}(\tilde{x}_i, \tilde{u}_i) \leftarrow \hat{q}(\tilde{x}_i, \tilde{u}_i) + \alpha\left[\tilde{r}_{i+1} + \gamma \max_u \hat{q}(\tilde{x}_{i+1}, u) - \hat{q}(\tilde{x}_i, \tilde{u}_i)\right]$;
    $k \leftarrow k + 1$;
  until x_k *is terminal*;

Algo. 7.1: Dyna with Q-learning (Dyna-Q)

Remarks on Dyna-Q Implementation

The specific Dyna-Q characteristics are:

- Direct RL update: Q-learning,
- Model: simple memory buffer of previous real experience,
- Search strategy: random choices from model buffer.

Moreover:

- Number of Dyna planning steps n is to be delimited from n-step bootstrapping (same symbol, two interpretations).
- Without the model $\hat{\mathcal{M}}$ one would receive one-step Q-learning.
- The model-based learning is done n times per real environment interaction:
  - Previous real experience is re-applied to Q-learning.
  - Can be considered a background task: choose $\max n$ s.t. hardware limitations (prevent turnaround errors).
- For stochastic environments: use a distributional model as in (7.3).
  - Update rule then may be modified from sample to expected update.

Maze Example (1)

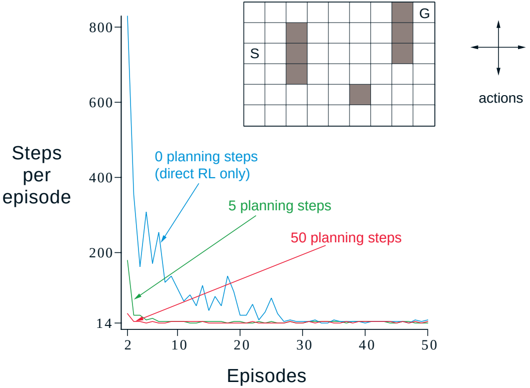


Fig. 7.7: Applying Dyna-Q with different planning steps n to simple maze (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Maze with obstacles (gray blocks)
- Start at S and reach G
- $r_T = +1$ at G
- Episodic task with $\gamma = 0.95$
- Step size $\alpha = 0.1$
- Exploration $\varepsilon = 0.1$
- Averaged learning curves

Maze Example (2)

- Blocks without an arrow depict a neutral policy (equal action values).
- Black squares indicate agent's position during second episode.
- Without planning ($n = 0$), each episodes only adds one new item to the policy.
- With planning ($n = 50$), the available experience is efficiently utilized.
- After the third episode, the planning agent found the optimal policy.

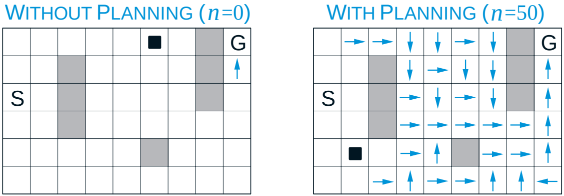


Fig. 7.8: Policies (greedy action) for Dyna-Q agent halfway through second episode (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

What if the Model is Wrong?

Possible model error sources:

- A provided a priori model may be inaccurate (expert knowledge).
- Environment behavior changes over time (non-stationary).
- Early-stage model is biased due to learning process.
- If function approximators are used: generalization error (cf. lecture 08 and following).

Consequences:

- Model errors are likely to lead to a suboptimal policy.
- If lucky: errors are quickly discovered and directly corrected by default, random exploration.
- Nevertheless, more intelligent exploration / correction strategies might be useful (compared to random actions as in ε-greedy strategies).

The Blocking Maze Example

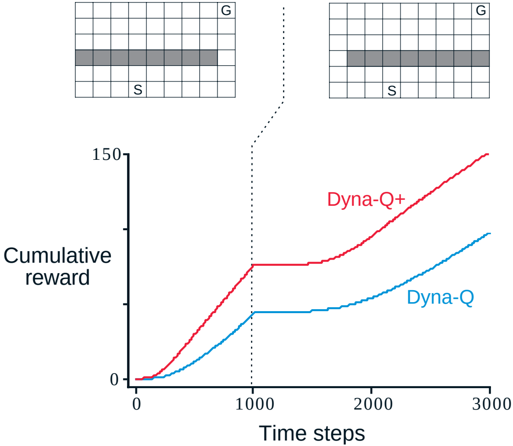


Fig. 7.9: Maze with a changing layout after 1000 steps illustrates a model error (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Maze with a changing obstacle line
- Start at S and reach G
- $r_T = +1$ at G
- Dyna-$Q+$ encourages intelligent exploration (upcoming slides)
- Dyna-Q requires more steps in order to overcome the blockade
- Averaged learning curves

The Shortcut Maze Example

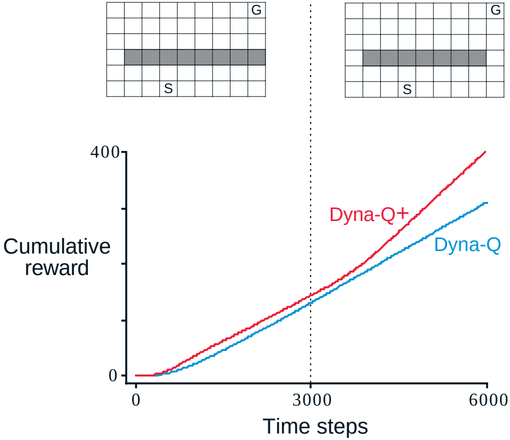


Fig. 7.10: Maze with an additional shortcut after 3000 steps (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Maze opens a shortcut after 3000 steps
- Start at S and reach G
- $r_T = +1$ at G
- Dyna-Q with random exploration is likely not finding the shortcut
- Dyna-$Q+$ exploration strategy is able to correct internal model
- Averaged learning curves

Dyna-$Q+$ Extensions

Compared to default Dyna-Q in Algo. 7.1, Dyna-$Q+$ contains the following extensions:

- Search heuristic: add $\kappa\sqrt{\tau}$ to regular reward.
  - τ: is the number of time steps a state-action transition has not been tried.
  - κ: is a small scaling factor $\kappa \in \{\mathbb{R}|0 < \kappa\}$.
  - Agent is encouraged to keep testing all accessible transitions.
- Actions for given states that had never been tried before are allowed for simulation-based planning.
  - Initial model for that: actions lead back to same state without reward.

Table of Contents

Background and Idea

- Dyna-Q randomly samples from the memory buffer.
  - Many planning updates maybe pointless, e.g., zero-valued state updates during early training.
  - In large state-action spaces: inefficient search since transitions are chosen far away from optimal policies.
- Better: focus on important updates.
  - In episodic tasks: backward focusing starting from the goal state.
  - In continuing tasks: prioritize according to impact on value updates.
- Solution method is called prioritized sweeping.
  - Build up a queue of every state-action pair whose value would change significantly.
  - Prioritize updates by the size of change.
  - Neglect state-action pairs with only minor impact.

Algorithmic Implementation: Prioritized Sweeping

parameter: $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $n \in \{\mathbb{N}|n \geq 1\}$, $\theta \in \{\mathbb{R}|\theta \geq 0\}$
init: $\hat{q}(x, u)$ arbitrary and $\hat{\mathcal{M}}(x, u)$ $\forall$ $\{x \in \mathcal{X}, u \in \mathcal{U}\}$, empty queue $\mathcal{Q}$
for $j = 1, 2, \ldots$ *episodes* **do**
    Initialize x_0 and $k \leftarrow 0$;
    repeat
        Choose u_k from x_k using a soft policy derived from $\hat{q}(x, u)$;
        Take action u_k, observe r_{k+1} and x_{k+1};
        $\hat{\mathcal{M}}(x_k, u_k) \leftarrow \{r_{k+1}, x_{k+1}\}$ (assuming deterministic env.);
        $P \leftarrow |r_{k+1} + \gamma \max_u \hat{q}(x_{k+1}, u) - \hat{q}(x_k, u_k)|$;
        if $P > \theta$ **then** insert $\{x_k, u_k\}$ in $\mathcal{Q}$ with priority P;
        for $i = 1, 2, \ldots$ *n while queue* $\mathcal{Q}$ *is not empty* **do**
            $\{\tilde{x}_i, \tilde{u}_i\} \leftarrow \arg\max_P(\mathcal{Q})$;
            $\{\tilde{r}_{i+1}, \tilde{x}_{i+1}\} \leftarrow \hat{\mathcal{M}}(\tilde{x}_i, \tilde{u}_i)$;
            $\hat{q}(\tilde{x}_i, \tilde{u}_i) \leftarrow \hat{q}(\tilde{x}_i, \tilde{u}_i) + \alpha \left[\tilde{r}_{i+1} + \gamma \max_u \hat{q}(\tilde{x}_{i+1}, u) - \hat{q}(\tilde{x}_i, \tilde{u}_i)\right]$;
            for $\forall \{\overline{x}, \overline{u}\}$ *predicted to lead to* $\tilde{x}_i$ **do**
                $\overline{r} \leftarrow$ predicted reward for $\{\overline{x}, \overline{u}, \tilde{x}_i\}$;
                $P \leftarrow |\overline{r} + \gamma \max_u \hat{q}(\tilde{x}_i, u) - \hat{q}(\overline{x}, \overline{u})|$;
                if $P > \theta$ **then** insert $\{\overline{x}, \overline{u}\}$ in $\mathcal{Q}$ with priority P;
        $k \leftarrow k + 1$;
    until x_k *is terminal*;

Remarks on Prioritized Sweeping Implementation

The specific prioritized sweeping characteristics are:

- Direct RL update: Q-learning,
- Model: simple memory buffer of previous real experience,
- Search strategy: prioritized updates based on predicted value change.

Moreover:

- θ is a hyperparameter denoting the update significance threshold.
- Prediction step regarding $\tilde{x}_i$ is a backward search in the model buffer.
- For stochastic environments: use a distributional model as in (7.3).
  - Update rule then may be modified from sample to expected update.

Comparing Against Dyna-Q on Simple Maze Example

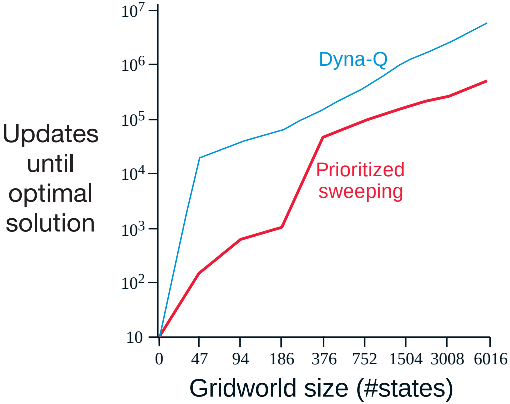


Fig. 7.11: Comparison of prioritized sweeping and Dyna-Q on simple maze (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

- Environment framework as in Fig. 7.7
- But: changing maze sizes (number of states)
- Both methods can utilize up to $n = 5$ planning steps
- Prioritized sweeping finds optimal solution 5-10 times quicker

Table of Contents

Update Rule Alternatives

- Dyna updates (search strategy) are not bound to Q-learning during planning and can be exchanged in many ways (see Fig. 7.12).
- Even evaluating a fixed policy π in terms of $v_\pi(x)$ and $q_\pi(x, u)$ is possible.

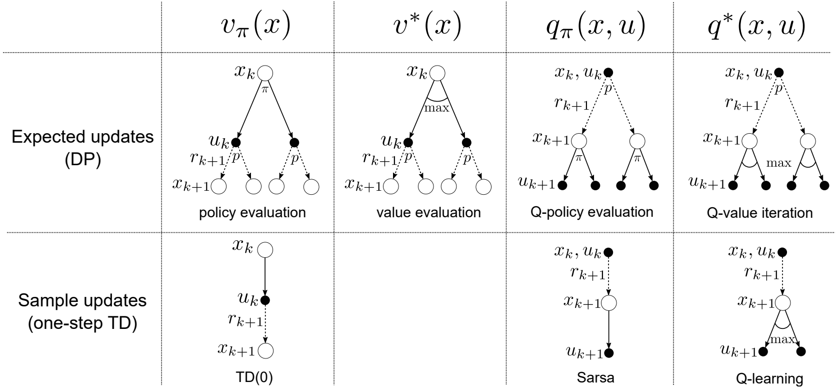


Fig. 7.12: Possible one-step updates: alternatives for Dyna

Advantages & Drawbacks: Expected vs. Sampled Updates

Pro expected updates:

- Delivers more accurate value estimates (no sampling error).

Pro sample updates:

- Is computational cheaper (e.g., distributional model not required).

Leads to trade-off:

- Estimation accuracy vs. computational burden.
- Evaluate decision on given problem, i.e., how many state-action pairs have to be evaluated for a new expected update?
- Utilize branching factor b metric: corresponds to number of possible next states x' with $p(x'|x, u) > 0$.
  - If expected update is available, this will be roughly as accurate as b samplings.
  - If only incomplete expected update is available, prefer sampling solution (often applies to large state-action spaces).

Example for Sampled vs. Expected Updates

- Artificial prediction task where all b successor states are equally likely
- Method initialization such that RMS error is always one
- Sample updates perform particularly well for large branching factors b (takeaway message: if facing large stochastic problems, use sampling)

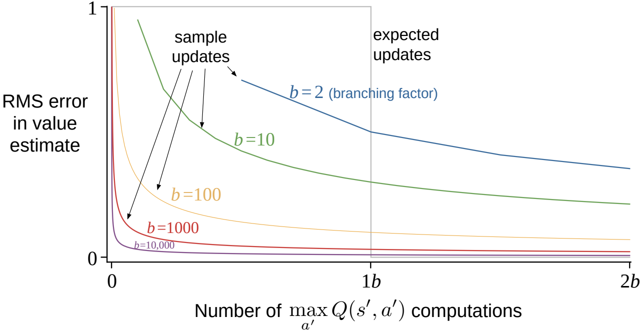


Fig. 7.13: Comparison of expected vs. sampled updates (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Alternatives on Distributing the Updates

Recap:

- Dynamic programming: sweep through the entire state(-action) space
- Dyna-Q: random uniform sampling
- Mutual problem: irrelevant updates decrease computational efficiency

Alternative: update according to on-policy distribution

- Based on sampling along the encountered state(-action) pairs (trajectory sweeping)
- Based on explicit on-policy distribution
- In both cases: ignore vast, uninteresting parts of the problem space at the risk of updating same old parts all over again

Exemplary Update Distribution Comparison

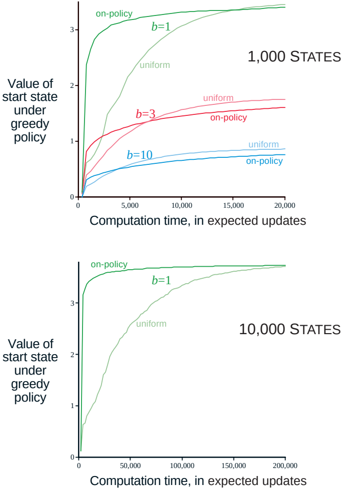


Fig. 7.14: Update distribution comparison (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Example:

- Two actions per state
- Both actions led to b next states
- $10\,\%$ probability of transition to terminal state
- reward per transition: $\mathcal{N}(\mu = 0, \sigma^2 = 1)$
- Task: estimate start state value
- 200 randomly generated undiscounted episodic runs

Table of Contents

Background Planning vs. Planning at Decision Time

Background Planning (discussed so far):

- Gradually improves policy or value function if time is available.
- Backward view: re-apply gathered experience.
- Feasible for fast execution: policy or value estimate are available with low latency (important, e.g., for real-time control).

Planning at decision time[1] (not yet discussed alternative):

- Select single next future action through planning.
- Forward view: predict future trajectories starting from current state.
- Typically discards previous planning outcomes (start from scratch after state transition).
- If multiple trajectories are independent: easy parallel implementation.
- Most useful if fast responses are not required (e.g., turn-based games).

[1] Can be interpreted as *model predictive control* in an engineering context.

Heuristic Search

- Develop tree-like continuations from each state encountered.
- Approximate value function at leaf nodes (using a model) and back up towards the current state.
- Choose action according to predicted trajectory with highest value.
- Predictions are normally discarded (new search tree in each state).

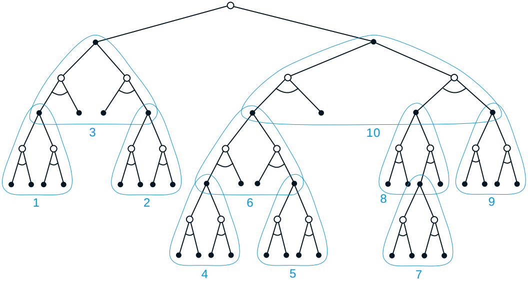


Fig. 7.15: Heuristic search tree with exemplary order of back-up operations (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Rollout Algorithms

- Similar to heuristic search, but: simulate trajectories following a rollout policy.
- Use Monte Carlo estimates of action value only for current state to evaluate on best action.
- Gradually improves rollout policy but optimal policy might not be found if rollout sequences are too short.
- Predictions are normally discarded (new rollout in each state).

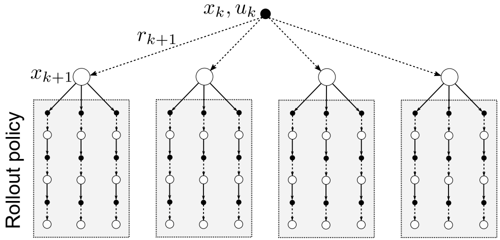


Fig. 7.16: Simplified processing diagram of rollout algorithms

Monte Carlo Tree Search (MCTS)

- Rollout algorithm, but:
  - accumulates values estimates from former MC simulations,
  - makes use of an informed tree policy (e.g., ε-greedy).

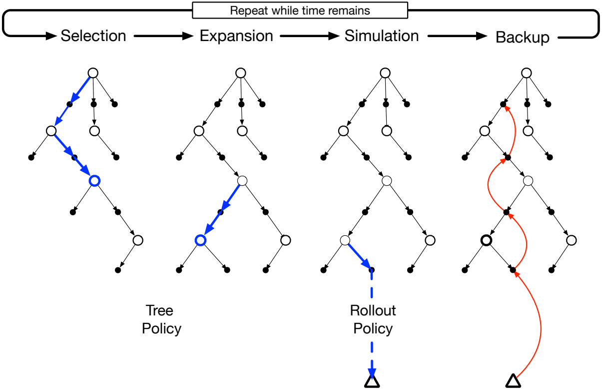


Fig. 7.17: Basic building blocks of MCTS algorithms (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)

Basic MCTS Procedure

Repeat the following steps while prediction time is available:

1. Selection: Starting at root node, use a tree policy (e.g., ε-greedy) to travel through the tree until arriving at a leaf node.
   - The tree policy exploits auspicious tree regions while maintaining some exploration.
   - It is improved and (possibly) extended in every simulation run.
2. Expansion: Add child node(s) to the leaf node by evaluating unexplored actions (optional step).
3. Simulation: Simulate the remaining full episode using the rollout policy starting from the leaf or child node (if available).
   - The rollout policy could be random, pre-trained or based on model-free methods using real experience (if available).
4. Backup: Update the values along the traveled trajectory but only saves those within the tree policy.

Further MCTS Remarks

What is happening after reaching the feasible simulation runs?

- After time is up, MCTS picks an appropriate action regarding the root node, e.g.:
  - The action visited the most times during all simulation runs or
  - The action having the largest action value.
- After transitioning to a new state, the MCTS procedure re-starts:
  - Either with a new tree incorporating only the root node or
  - by re-utilizing the applicable parts from the previous tree.

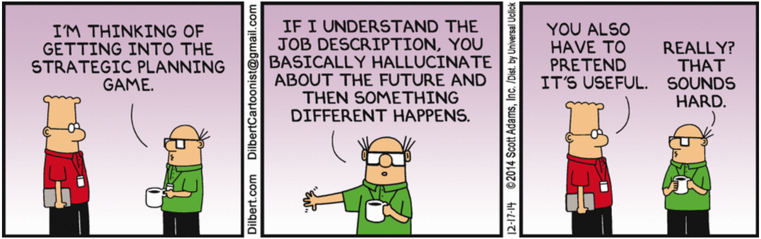
Further reading on MCTS:

- MCTS-based algorithms are not limited to game applications but were able to achieve outstanding success in this field.
  - Famous AlphaGo (cf. Keynote lecture from D. Silver)
- More in-depth lectures on MCTS can be found (among others) here:
  - Stanford Online: CS234
  - MIT OpenCourseWare
  - Extensive slide set from M. Sebag at Universite Paris Sud

Summary: What You've Learned Today

- Model-free RL is easy to implement and cannot suffer any model learning error while model-based approaches use a limited amount of experience much more efficient.
- Integrating these two RL branches can be achieved using the Dyna framework (background planning) incorporating the steps:
  - Direct RL updates (any model-free approach, e.g., Q-learning),
  - Model learning: use real experience to improve model predictions,
  - Search control: strategies on how to generate simulated experience.
- The Dyna framework allows many different algorithms such as Dyna-$Q(+)$ or prioritized sweeping.
  - Learning efficiency is much increased compared to pure model-based/free approaches.
  - Many degrees of freedom regarding internal update rules exist.
- In contrast, planning at decision time predicts future trajectories starting from the current state (forward view).
  - Rather computationally expensive leading to high latency responses.
  - The Monte Carlo tree search rollout algorithm is a well-known example.

The End for Today



Thanks for your attention and have a nice week!