1 Planning and Learning with Tabular Methods

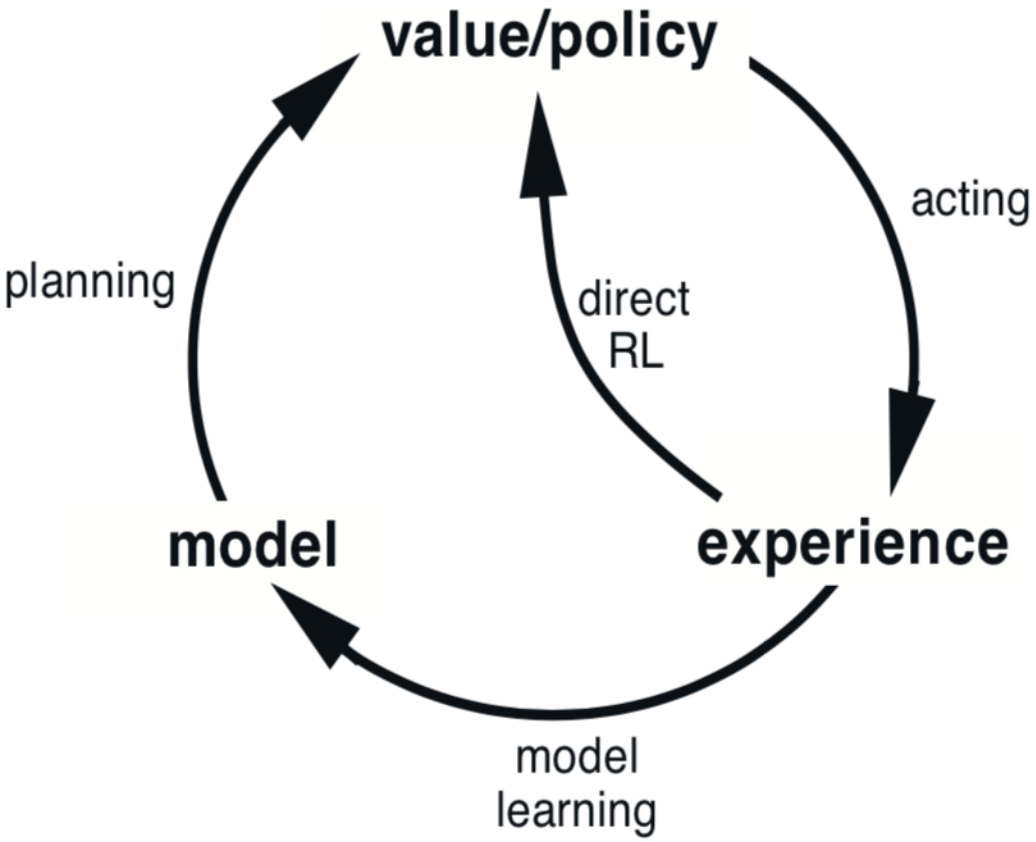
1.1 Models and Planning

A *model* of the environment is anything that an agent can use to predict how the environment will respond to its actions. A *distribution model* is one that characterises the distribution of possible environmental changes, whereas a *sample model* is one that produces sample behaviour. Distribution models are in some sense stronger, in that they can be used to produce samples of the behaviour of the environment, but it is often easier to reproduce sample responses than to model the response distribution.

Models can be used to simulate the environment and hence simulate experience. We use the term *planning* to refer to a computational process that uses a model for improving a policy. The kind of planning that we consider here falls under the name *state-space planning*, since it is a search through the state space for an optimal policy or path to a goal. (Planning as we consider it here is essentially just learning from simulated experience.)

1.2 Dyna: Integrated Planning, Acting and Learning

Within a planning agent, real experience can be used to improve the model or to directly improve the value function and policy. The former we call *model learning* and the latter we call *direct reinforcement learning*. The use of a learned model to improve the value function and policy is sometimes called *indirect reinforcement learning*. The figure below illustrates this duality.



Dyna-Q

Dyna-Q uses one-step tabular Q-learning to learn from both real and simulated experience. (It is typical to use the same update rule for both types of experience.) The idea is that a model and a value function are learned simultaneously from real experience, and the model is then used for further planning. An algorithm is given below. Note that, although not shown in this way, the planning and direct learning can run concurrently.

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
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

1.3 When the Model is Wrong

Of course, the model we are learning may be incorrect, meaning that planning results in a sub-optimal policy. If the environment's dynamics are non-stationary, then this will be an issue for the agent.

In some cases, the suboptimal policy results in discovery and correction of model error, since if the model leads to optimistic estimates for action-values the agent will take these actions and realise its modelling error. The situation can be more difficult when values are underestimated, since in this case the agent may never choose to have experience that would correct its model.

Dyna-Q+

The aforementioned issue of model error, especially in non-stationary environments, is the general problem of exploration versus exploitation. There is probably no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent keeps track of the time elapsed since it last visited each state-action pair, then increases the reward from visiting these pairs in *simulated* experience to $r + \kappa\sqrt{\tau}$, where r is the modeled reward for the transition, τ is the number of time-steps since the last time the state-action pair was visited and κ is a small constant. This increases computational complexity, but has the benefit of encouraging the agent to try actions that it hasn't taken in a long time.

1.4 Prioritised Sweeping

In the Dyna-Q algorithm given above, planning is done using uniform sweeps of the state-action space. This could be very wasteful, for instance because it is possible that there are many parts of

the state-action space that are irrelevant to the optimal policies. It is also the case that uniformly distributed planning updates could waste effort on states whose value functions have not changed recently, which is wasted computation.

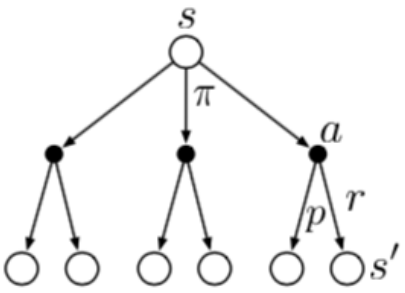
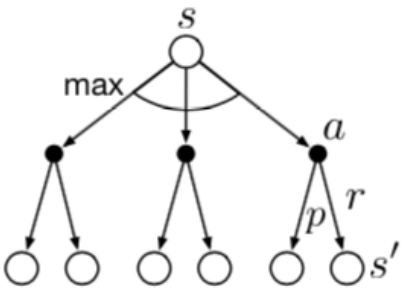
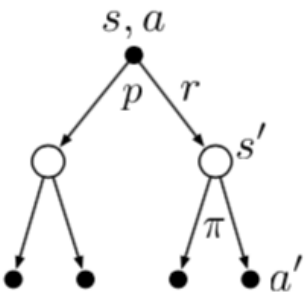
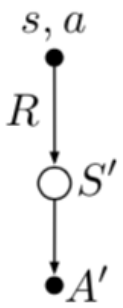
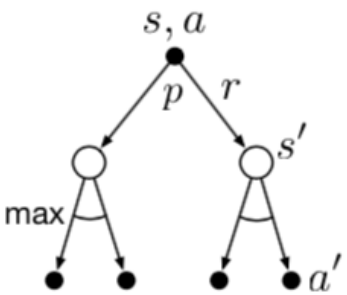
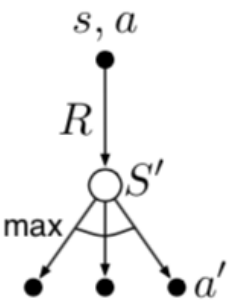
Prioritised sweeping focuses updates on the state-action pairs whose estimated values are likely to change the most from the most recent experience. Q queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritised by the size of the change. During planning, the state-action pair that is first in the queue is updated and removed from the queue first, then it's predecessors are updated and removed (if the update would be significant), and so on. An algorithm for deterministic environments is given below.

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

1.5 Expected vs. Sample Updates

This section is about the relative benefits of expected and sample updates. Expected updates consider all possible outcomes, while sample updates use only sample experience of particular outcomes. In the absence of a distribution model, expected updates are not possible. A summary of all one-step updates considered are given below.

Value estimated	Expected updates (DP)	Sample updates (one-step TD)
$v_\pi(s)$	 policy evaluation	 TD(0)
$v_*(s)$	 value iteration	
$q_\pi(s,a)$	 q-policy evaluation	 Sarsa
$q_*(s,a)$	 q-value iteration	 Q-learning

Since expected updates do not directly suffer from sampling error (error could propagate through model estimation in planning), they are more computationally intensive. However, they are not always optimal. In problems with large state-spaces or branching factors, sample updates are often much more efficient. This means that one can do many sample updates in the same computational time as an expected update, in turn meaning that the sample updates produce more accurate value estimates in the given time.

1.6 Trajectory Sampling

As discussed previously, distributing updates uniformly during planning is often sub-optimal. This is because for many tasks, the majority of possible updates will be on irrelevant or low-probability trajectories.

We could generate experience and updates in planning by interacting the current policy with the model, then only updating the simulated trajectories. We call this *trajectory sampling*. Naturally, trajectory sampling generates updates according to the on-policy distribution.

Focusing on the on-policy distribution could be beneficial because it causes uninteresting parts of the space to be ignored, but it could be detrimental because it causes the same parts of the space to be updated repeatedly. It is often the case the distributing updates according to the on-policy distribution is preferable to using the uniform distribution for larger problems.

1.7 Real-time Dynamic Programming

Real-time dynamic programming (RTDP) is a on-policy, trajectory-sampling version of value-iteration DP. This is DP value iteration, but with the updates distributed according to the on-policy distribution. As such, it is a form of asynchronous DP.

Due to the trajectory sampling, RTDP allows us to skip portions of the state space that are not relevant to the current policy (in terms of the prediction problem). For the control problem (finding an optimal policy) all we really need is an *optimal partial policy*, which is a policy that is optimal on the relevant states and specifies arbitrary actions on the others.

In general, finding an optimal policy with on-policy trajectory-sampling control method (e.g. Sarsa) requires visiting all state action pairs infinitely many times in the limit. This is true for RTDP as well, but there are certain types of problems for which RTDP is guaranteed to find ann optimal partial policy without visiting all states infinitely often. This is an advantage for problems with very large state sets.

The particular tasks for which this is the case are *stochastic optimal path problems* (which are generally framed in terms of cost minimisation rather than reward maximisation). They are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards. For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided: 1) the initial value of every goal state is zero, 2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, 3) all rewards for transitions from non-goal states are strictly negative, and 4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to zero).

1.8 Planning at Decision Time

The type of planning we have considered so far is the improvement of a policy or value function based on simulated experience. This is not focussed on interaction with the environment and is called *background planning*.

An alternative type of planning, *decision time planning*, is the search (sometimes many actions deep) of possible future trajectories given the current state.

1.9 Heuristic Search

In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes discussed throughout this book. The backing up stops at the state–action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

1.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. Rollout algorithms start in a given state, then estimate the value of the state by averaging simulated returns from that state after following a given policy, called the *rollout policy*. The action with the highest estimated value is selected and the process is repeated. This is useful when one knows a policy but needs to average over some stochasticity in the environment.

1.11 Monte Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a successful example of decision time planning. It is a rollout algorithm that accumulates value estimates from the Monte Carlo simulations in order to guide the search. A variant of MCTS was used in AlphaGo.

A basic version of MCTS follows the following steps, starting at the current state:

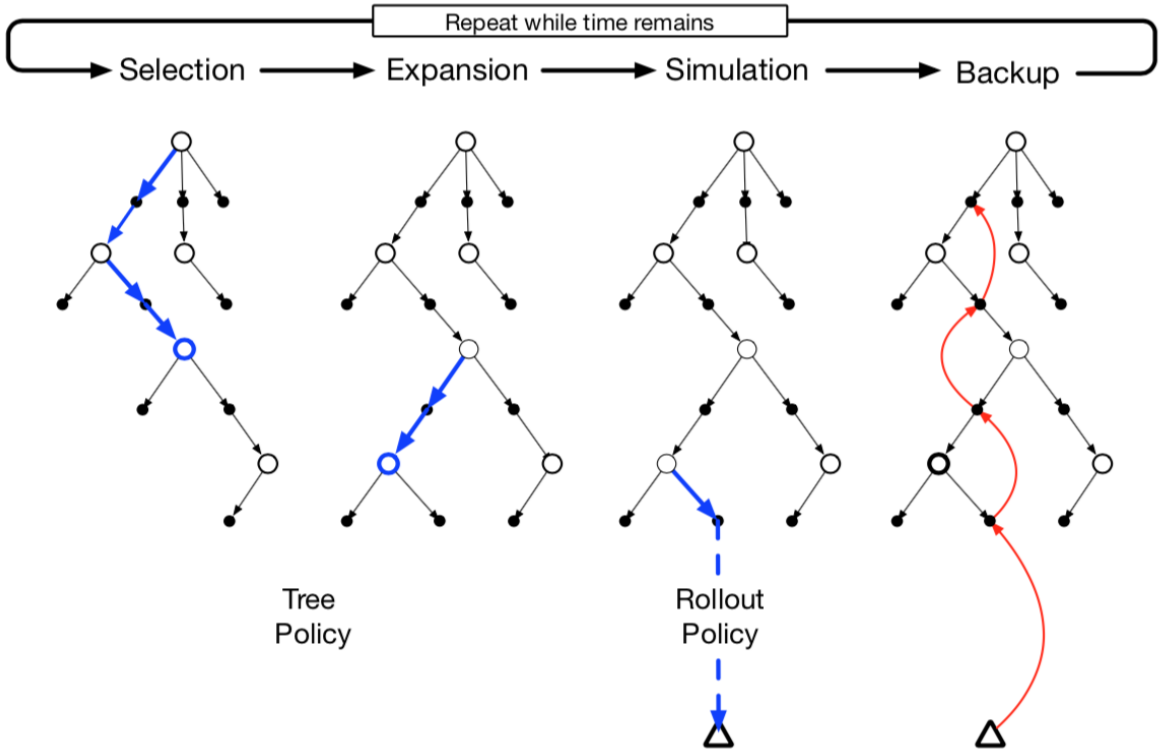
1. **Selection.** Starting at the root node, a *tree policy* based on action-values attached to the edges of the tree (that balances exploration and exploitation) traverses the tree to select a leaf node.

2. **Expansion.** On some iterations (depending on the implementation), the tree is expanded from the selected leaf node by adding one of more child nodes reached from the selected node via unexplored actions.

3. **Simulation.** From the selected node, or from one if its newly added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.

4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialise, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree.

The figure below illustrates this process. MCTS executes this process iteratively, starting at the current state, until no more time is left or computational resources are exhausted. An action is then taken based on some statistics in the tree (e.g. largest action-value or most visited node). After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.



Summary of Part I