Chapter 8: Planning and Learning

Objectives of this chapter:

- To think more generally about uses of environment models
- Integration of (unifying) planning, learning, and execution
- “Model-based reinforcement learning”
DP with Distribution models

In Chapter 4, we assumed access to a model of the world

- These models describe all possibilities and their probabilities
- We call them **Distribution models**
  - e.g., \( p(s', r | s, a) \) for all \( s, a, s', r \)

In Dynamic Programing we sweep the states:

- in each state we consider all the possible rewards and next state values
- the model describes the next states and rewards and their associated probabilities
- using these values to update the value function

In Policy Iteration, we then improve the policy using the computed value function
Chapter 8: Planning and Learning

Objectives of this chapter:

- To think more generally about uses of environment models
- Integration of (unifying) planning, learning, and execution
- “Model-based reinforcement learning”
Paths to a policy

Model-based RL
Sample Models

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

- **Sample model**, a.k.a. a simulation model
  - produces sample experiences for given $s, a$
    - sampled according to the probabilities
  - allows reset, exploring starts
  - often much easier to come by

- Both types of models can be used mimic or simulate experience: to produce *hypothetical experience*
Consider modeling the sum of two dice

- A *distribution model* would produce all possible sums and their probabilities of occurring
- A *sample model* would produce an individual sum drawn according to the correct probability distribution

When we solved the Gambler’s problem with value iteration, we used the distribution model

When you solved the Gambler’s problem with Monte-Carlo, you implemented a sample model in your environment code
Planning

Planning: any computational process that uses a model to create or improve a policy

We take the following (unusual) view:

- update value functions using both real and simulated experience
- all state-space planning methods involve computing value functions, either explicitly or implicitly
- they all apply updates from simulated experience
Planning Cont.

- Classical DP methods are state-space planning methods
- Heuristic search methods are state-space planning methods
- A planning method based on Q-learning:

**Random-sample one-step tabular Q-planning**

Do forever:
1. Select a state, $S \in S$, and an action, $A \in A(s)$, 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 $S, A, R, S'$:
   \[ Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)] \]
Paths to a policy

Dyna
Two uses of real experience:

- **model learning**: to improve the model
- **direct RL**: to directly improve the value function and policy

Improving value function and/or policy via a model is sometimes called **indirect RL**. Here, we call it **planning**.
Direct (model-free) vs. Indirect (model-based) RL

- **Direct methods**
  - simpler
  - not affected by bad models

- **Indirect methods**:
  - make fuller use of experience: get better policy with fewer environment interactions

But they are very closely related and can be usefully combined:

- planning, acting, model learning, and direct RL can occur simultaneously and in parallel
The Dyna Architecture

Agent

Environment

Model

Policy/value functions

planning update

simulated experience

search control

real experience

model learning

direct RL update

real experience

direct RL update

The Dyna Architecture
The Dyna-Q Algorithm

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

**direct RL**

**model learning**

**planning**
Demo
A simple maze: problem description

- 47 states, 4 actions, deterministic dynamics
- Obstacles and walls
- Rewards are 0 except +1 for transition into goal state
- $\gamma = 0.95$, discounted episodic task

Agent parameters:
- $\alpha = 0.1$, $\epsilon = 0.1$
  - Initial action-values were all zero

Let’s compare one-step tabular Q-learning and Dyna-Q with different values of $n$
Dyna-Q on a Simple Maze

rewards = 0 until goal, when = 1
Dyna-Q Snapshots: Midway in 2nd Episode

**WITHOUT PLANNING** $(n=0)$

**WITH PLANNING** $(n=50)$
Notice during the demo, that the one-step Q-learning agent and Dyna-Q agent appeared to be equally reactive.

Updating the value function and selecting a new action is very fast.

Thus, there is usually some time left over.

We can use that time to run a planning loop.

- Planning with n=5 helps a lot.

What are other ways we could integrate planning with learning and acting?

Planning of this form is *anytime*.
When the model is wrong

- So far we have considered models, that:
  - Start empty and are always updated with correct info.
- The model can be wrong! Because:
  - environment might be stochastic and we have only seen a few samples
  - the environment has changed
- Planning is likely to compute a suboptimal policy in this case

- Imagine the world changed, and:
  - The suboptimal policy leads to discovery and correction of the modeling error
Consider the Gridworld with a partition wall
Dyna-Q can find the policy for reaching the goal state
Then after 1000 time steps, we change the world:
- we block the path of the existing planned policy
- and open up a new path

Figure 8.5: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration.
When the Model is Wrong: Blocking Maze

The changed environment is harder
A change for the better

- If the world changes, such that a better path is possible:
  - the formerly correct policy will not reveal the improved situation
  - the modeling error may not be detected for a long time!
When the Model is Wrong: Shortcut Maze

The changed environment is easier
What is Dyna-Q+?

- Uses an “exploration bonus”:
  - Keeps track of time since each state-action pair was tried for real
  - An extra reward is added for transitions caused by state-action pairs related to how long ago they were tried: the longer unvisited, the more reward for visiting

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

- The agent actually “plans” how to visit long unvisited states
The conflict between exploration and exploitation

- Exploration in planning: trying actions that improve the model
  - Make it more accurate
  - Make it a better match with the environment
  - Proactively discover when the model is wrong

- Exploitation: behaving optimally with respect to the current model

- Simple heuristics can be effective
Prioritizing Search Control

Consider the second episode in the Dyna maze

- The agent has successfully reached the goal once…

In larger problems, the number of states is so large that unfocused planning would be extremely inefficient

**Figure 8.6:** Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent.

Agent took about 25 episodes to reach ("-"optimal performance, whereas the \( n = 5 \) agent took about five episodes, and the \( n = 50 \) agent took only three episodes. Figure 8.6 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the \( n = 0 \) and \( n = 50 \) agents halfway through the second episode. Without planning (\( n = 0 \)), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the episode's end will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained.

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

**Exercise 8.1** The nonplanning method looks particularly poor in Figure 8.6 because it is a one-step method; a method using eligibility traces would do better. Do you think an eligibility trace method could do as well as the Dyna method? Explain why or why not.
Large maze and random search control

(Peng and Williams, 1993)
Prioritized Sweeping

Which states or state-action pairs should be generated during planning?

Work backwards from states whose values have just changed:

- Maintain a queue of state-action pairs whose values would change a lot if backed up, prioritized by the size of the change
- When a new backup occurs, insert predecessors according to their priorities
- Always perform backups from first in queue

Moore & Atkeson 1993; Peng & Williams 1993
improved by McMahan & Gordon 2005; Van Seijen 2013
Prioritized Sweeping

Initialize $Q(s, a)$, $Model(s, a)$, for all $s, a$, and $PQueue$ to empty

Do forever:

(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow policy(S, Q)$
(c) Execute 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) Repeat $n$ times, while $PQueue$ is not empty:
   
   $S, A \leftarrow first(PQueue)$
   $R, S' \leftarrow Model(S, A)$
   
   $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

Repeat, 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 $P$
Prioritized Sweeping vs. Dyna-Q

Both use $n=5$ backups per environmental interaction.
Rod Maneuvering (Moore and Atkeson 1993)

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps. The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately 1/20 of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of $20 \times 20$ positions. The figure below shows the obstacles and the shortest solution from start to goal, found by prioritized sweeping.

This problem is deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. Figure reprinted from Moore and Atkeson (1993).

Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state–action pair has been experienced and of what the next states were. It is natural then to update each pair not with a sample update, as we have been using so far, but with an expected update, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses expected updates, which in stochastic environments may waste lots of computation on low-probability transitions. As we show in the following section, sample updates can in many cases get closer to the true value function with less computation despite the variance introduced by sampling. Sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact. This idea was taken to what may be its logical limit in the “small backups” introduced by van Seijen and Sutton (2013). These are updates along a single transition, like a sample update, but based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done. In this section we have emphasized backward focusing, but this
Planning is a form of state-space search
- a massive computation which we want to control to maximize its efficiency

Prioritized sweeping is a form of search control
- focusing the computation where it will do the most good

But can we focus better?

Can we focus more tightly?

Small backups are perhaps the smallest unit of search work
- and thus permit the most flexible allocation of effort
CHAPTER 8. PLANNING AND LEARNING WITH TABULAR METHODS

is just one strategy. For example, another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called forward focusing. Peng and Williams (1993) and Barto, Bradtke and Singh (1995) have explored versions of forward focusing, and the methods introduced in the next few sections take it to an extreme form.

8.5 Expected vs. Sample Updates

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of expected and sample updates.

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of updates for approximating the four value functions:

- $v_\pi(s)$: Expected updates (DP)
- $v_*(s)$: Sample updates (one-step TD)
- $q_\pi(s, a)$: Expected updates (DP)
- $q_*(s, a)$: Sample updates (one-step TD)

Value estimated Expected updates (DP) Sample updates (one-step TD)

\[ v_\pi(s) \]
\[ s \]
\[ s' \]
\[ a \]
\[ r \]
\[ p \]
\[ \pi \]

policy evaluation

\[ v_*(s) \]
\[ s \]
\[ s' \]
\[ a \]
\[ r \]
\[ p \]
\[ \max \]

value iteration

\[ q_\pi(s, a) \]
\[ s, a \]
\[ s' \]
\[ p \]
\[ r \]
\[ \pi \]

q-policy evaluation

\[ q_*(s, a) \]
\[ s, a \]
\[ s' \]
\[ a' \]
\[ \max \]

q-value iteration

\[ v_\pi(s) \]
\[ s \]
\[ A \]
\[ R \]
\[ S' \]

TD(0)

\[ v_*(s) \]
\[ s \]
\[ A \]
\[ R \]
\[ S' \]

TD(0)

\[ q_\pi(s, a) \]
\[ s, a \]
\[ A' \]
\[ R \]
\[ S' \]

Sarsa

\[ q_*(s, a) \]
\[ s, a \]
\[ A' \]
\[ R \]
\[ S' \]

Q-learning

Figure 8.7: Diagrams of all the one-step updates considered in this book.
Full vs. Sample Backups

Figure 8.8 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for expected and sample updates for a variety of branching factors, \( b \). The case considered is that in which all \( b \) successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. In this case, sample updates reduce the error according to \( \frac{1}{t} \) where \( t \) is the number of sample updates that have been performed (assuming sample averages, i.e., \( \frac{1}{t} \)). The key observation is that for moderately large \( b \) the error falls dramatically with a tiny fraction of \( b \) updates. For these cases, many state–action pairs could have their values improved dramatically, to within a few percent of the effect of an expected update, in the same time that a single state–action pair could undergo an expected update.

The advantage of sample updates shown in Figure 8.8 is probably an underestimate of the real effect. In a real problem, the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly.

8.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state–action) space, updating each state (or state–action pair) once per sweep. This is problematic on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, updates can be distributed any way one likes (to assure convergence, all states or state–action pairs must be visited in the limit an infinite number of times; although an exception to this is discussed in Section 8.7 below), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state–action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the on-policy
Trajectory Sampling

- **Trajectory sampling**: perform updates along simulated trajectories
- This samples from the on-policy distribution
- Advantages when function approximation is used (Part II)
- Focusing of computation:
  can cause vast uninteresting parts of the state space to be ignored:

![Diagram of trajectory sampling](image)
Trajectory Sampling Experiment

- one-step full tabular updates
- uniform: cycled through all state-action pairs
- on-policy: backed up along simulated trajectories
- 200 randomly generated undiscounted episodic tasks
- 2 actions for each state, each with $b$ equally likely next states
- 0.1 prob of transition to terminal state
- expected reward on each transition selected from mean 0 variance 1 Gaussian
Heuristic Search

- Used for action selection, not for changing a value function (=heuristic evaluation function)
- Backed-up values are computed, but typically discarded
- Extension of the idea of a greedy policy — only deeper
- Also suggests ways to select states to backup: smart focusing:

![Diagram of heuristic search process](image-url)
Summary of Chapter 8

- Emphasized close relationship between planning and learning
- Important distinction between distribution models and sample models
- Looked at some ways to integrate planning and learning
  - synergy among planning, acting, model learning
- Distribution of backups: focus of the computation
  - prioritized sweeping
  - small backups
  - sample backups
  - trajectory sampling: backup along trajectories
  - heuristic search
- Size of backups: full/sample; deep/shallow
Summary of Part I: Dimensions

Core ideas common to all methods:
- estimation of value functions
- backing-up updates

Backups can be
- fat or skinny (expected or sample)
- short or tall (depth)

Problem dimensions:
- prediction vs control
- action values vs state values
- on-policy vs off-policy
- episodic vs continuing

Other method dimensions:
- online vs offline

Temporal-difference learning
Dynamic programming
Monte Carlo
Exhaustive search