Objectives of this chapter:

- Introduce Temporal Difference (TD) learning
- Focus first on policy evaluation, or prediction, methods
- Then extend to control methods by following the idea of Generalized Policy Iteration (GPI)
The neurotransmitter dopamine seems to implement something very similar to the so-called *temporal difference error* that gives TD methods their name.
Two pieces of motivation: II

Human-level control through deep reinforcement learning

Volodymyr Mnih*, Koray Kavukcuoglu*, David Silver*, Andrei A. Rusu1, Joel Veness1, Marc G. Bellemare1, Alex Graves1, Martin Riedmiller1, Andreas K. Fidjeland1, Georg Ostrovski1, Stig Petersen1, Charles Beattie1, Amir Sadik1, Ioannis Antonoglou1, Helen King1, Dharshan Kumaran1, Daan Wierstra1, Shane Legg1 & Demis Hassabis1

Nature, February 2015
6.1 TD Prediction

Policy Evaluation (the prediction problem):
for a given policy \( \pi \), compute the state-value function \( V^\pi \)

Recall:
Simple every-visit Monte Carlo method:
\[
V(S_t) \leftarrow V(S_t) + \alpha \left[ G_t - V(S_t) \right]
\]

\textcolor{red}{\textbf{target}}: the actual return after time \( t \)

The simplest TD method, TD(0):
\[
V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]
\]

\textcolor{red}{\textbf{target}}: an estimate of the return
Simple Monte Carlo

\[ V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)] \]

where \( R_t \) is the actual return following state \( S_t \).
Simplest TD Method

\[ V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \]
cf. Dynamic Programming

\[ V(S_t) \leftarrow E_{\pi} \left\{ R_{t+1} + \gamma V(S_t) \right\} \]
TD methods bootstrap and sample

- **Bootstrapping**: update involves an *estimate*
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps

- **Sampling**: update does not involve an *expected value*
  - MC samples
  - DP does not sample
  - TD samples
Monte Carlo method suitable for nonstationary environments is
\[ V(s_t) \leftarrow V(s_t) + \gamma G_t, \]
where \( G_t \) is the actual return following time \( t \), and \( \gamma \) is a constant step-size parameter (c.f., Equation 2.4). Let us call this method constant-\( \gamma \)MC.

Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to \( V(s_t) \) (only \( G_t \) is known), TD methods need wait only until the next time step. At time \( t+1 \) they immediately form a target and make a useful update using the observed reward \( R_{t+1} \) and the estimate \( V(s_{t+1}) \). The simplest TD method, known as TD(0), is
\[ V(s_t) \leftarrow V(s_t) + \gamma R_{t+1} + V(s_{t+1}) - V(s_t). \]

In effect, the target for the Monte Carlo update is \( G_t \), whereas the target for the TD update is \( R_{t+1} + V(s_{t+1}) \).

Because the TD method bases its update in part on an existing estimate, we say that it is a bootstrapping method, like DP. We know from Chapter 3 that
\[ v(\pi(s)) = \mathbb{E}_\pi[G_t \mid S_t = s] \]
\[ = \mathbb{E}_\pi\left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \]
\[ = \mathbb{E}_\pi\left[ R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s \right] \]
\[ = \mathbb{E}_\pi[R_{t+1} + \gamma v(\pi(S_{t+1})) \mid S_t = s]. \]

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because \( v(\pi(S_{t+1})) \) is not known and the current estimate, \( V(s_{t+1}) \), is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) and it uses the current estimate instead of the true \( v(\pi) \). Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.
Thus, TD methods combine the sampling of Monte Carlo with the bootstrapping of expected values in (6.4) instead. The TD target is an estimate for both reasons: it samples the environment, but because the expected values, which are assumed to be completely provided by a model of the environment, are used in place of the real expected return. The DP target is an estimate not because of the expected value in (6.3) is not known; a sample return is used as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used as a target. Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas TD(0) is a special case of TD(\(\gamma\)).

**Tabular TD(0) for estimating \(v_\pi\)**

Input: the policy \(\pi\) to be evaluated  
Initialize \(V(s)\) arbitrarily (e.g., \(V(s) = 0, \forall s \in S^+\))  
Repeat (for each episode):  
  Initialize \(S\)  
  Repeat (for each step of episode):  
    \(A \leftarrow \text{action given by } \pi\text{ for } S\)  
    Take action \(A\), observe \(R, S'\)  
    \(V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]\)  
    \(S \leftarrow S'\)  
  until \(S\) is terminal
## Example: Driving Home

The rewards in this example are the elapsed times on each leg of the journey. Thus as follows:

<table>
<thead>
<tr>
<th>State</th>
<th>Elapsed Time (minutes)</th>
<th>Predicted Time to Go</th>
<th>Predicted Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaving office, friday at 6</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>reach car, raining</td>
<td>5</td>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>exiting highway</td>
<td>20</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>2ndary road, behind truck</td>
<td>30</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>entering home street</td>
<td>40</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>arrive home</td>
<td>43</td>
<td>0</td>
<td>43</td>
</tr>
</tbody>
</table>

We are not discounting ($\gamma = 1$) and thus the return for each state is the actual time to go from that state. The value of each state is the expected time to go.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.3. The arrows show the changes in predictions recommended by the constant-MC method (6.1), for $\gamma = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t V(S_t)$, at this time is eight.

If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the negative of the elapsed time. But since we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.
Driving Home

Changes recommended by Monte Carlo methods ($\alpha=1$)

Changes recommended by TD methods ($\alpha=1$)

![Graph showing predicted total travel time vs. situation, with actual outcomes and changes recommended by Monte Carlo and TD methods.](image-url)
6.2 Advantages of TD Prediction Methods

- TD methods do not require a model of the environment, only experience
- TD, but not MC, methods can be fully incremental
  - You can learn **before** knowing the final outcome
    - Less memory
    - Less peak computation
  - You can learn **without** the final outcome
    - From incomplete sequences
- Both MC and TD converge (under certain assumptions to be detailed later), but which is faster?
Random Walk Example

Values learned by TD(0) after various numbers of episodes
TD and MC on the Random Walk

Data averaged over 100 sequences of episodes
**Optimality of TD(0)**

**Batch Updating**: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.

Compute updates according to TD(0), but only update estimates after each complete pass through the data.

For any finite Markov prediction task, under batch updating, TD(0) converges for sufficiently small $\alpha$.

Constant-$\alpha$ MC also converges under these conditions, but to a different answer!
Random Walk under Batch Updating

After each new episode, all previous episodes were treated as a batch, and algorithm was trained until convergence. All repeated 100 times.
You are the Predictor

Suppose you observe the following 8 episodes:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

$V(A)$?

$V(B)$?
You are the Predictor

\[ V(A) ? \]
You are the Predictor

- The prediction that best matches the training data is $V(A) = 0$
  - This minimizes the mean-square-error on the training set
  - This is what a batch Monte Carlo method gets
- If we consider the sequentiality of the problem, then we would set $V(A) = 0.75$
  - This is correct for the maximum likelihood estimate of a Markov model generating the data
  - i.e., if we do a best fit Markov model, and assume it is exactly correct, and then compute what it predicts (how?)
  - This is called the certainty-equivalence estimate (CEE)
  - This is what TD(0) gets
  - Note: directly calculating the CEE is often not feasible
6.4 Sarsa: On-Policy TD Control

First Step: Learning An Action-Value Function

Estimate $Q^\pi$ for the current behavior policy $\pi$.

After every transition from a nonterminal state $S_t$, do this:

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$  

If $S_{t+1}$ is terminal, then $Q(S_{t+1}, A_{t+1}) = 0$. 
6.4 Sarsa: On-Policy TD Control

Turn this into a control method by always updating the policy to be (more) greedy with respect to the current estimate:

**Sarsa: An on-policy TD control algorithm**

Initialize $Q(s, a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

- Initialize $S$
- Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)

Repeat (for each step of episode):

- Take action $A$, observe $R, S'$
- Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma Q(S', A') - Q(S, A) \right]$$

$S \leftarrow S'$; $A \leftarrow A'$;

until $S$ is terminal
Example: Windy Gridworld

undiscounted, episodic, reward = −1 until goal
Results of Sarsa on the Windy Gridworld
6.5 Q-Learning: Off-Policy TD Control

Q-learning: estimated action-value function Q directly approximates \( q^* \), independent of the policy being followed.

\[
Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]
\]

What is the backup diagram?
One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989), defined by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R + \gamma \max_a Q(s'_{t+1}, a) - Q(s_t, a_t)]$$

In this case, the learned action-value function, $Q$, directly approximates $q^\star$, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, $Q$ has been shown to converge with probability 1 to $q^\star$. The Q-learning algorithm is shown in procedural form in the box below.

Q-learning: An off-policy TD control algorithm

Initialize $Q(s, a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(terminal-state, \cdot) = 0$

Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Take action $A$, observe $R, S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(s'_{t+1}, a) - Q(S, A)]$
    $S \leftarrow S'$
  until $S$ is terminal
Example: Cliffwalking

\[ r = -1 \]

\[ r = -100 \]

\[ \epsilon \text{-greedy, } \epsilon = 0.1 \]
6.6 Expected SARSA

- Idea: don’t consider maximum over next state action pairs but use the expected value:

\[
Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right]
\]

\[
\leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]
\]

\[
Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right]
\]

\[
\leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]
\]

Q-learning

Expected SARSA
A. Cliff Walking

We begin by testing Hypothesis 1 using the cliff walking task. The agent has to move from the start state [S] to the square right of the goal and takes a ‘left’ action, it ends up in the square just above the goal. Each resulting in a reward of -1. The result of an action is a movement of 1 square in the corresponding direction plus an additional movement in the ‘up’ direction, corresponding to a movement of 1 square in the direction determined by the square below it. The reward for reaching the goal is 0, while for all others is -1.

As in the cliff walking task, we consider the policy to get worse in the long run. This is because the optimal value of Q-learning is also lower than 1 is the greedy sense, i.e. walking towards the goal with a detour.

That the optimal value of Q-learning is also lower than 1 is expected for a deterministic problem. If we also show the performance of Q-learning on this problem.

Let us consider the learning algorithm that is just like Q-learning except that instead of using an episode-based algorithm, that is, taking the discount factor into account (which we have denoted by the parameter $\gamma$), we compute the average return $R_t = \sum_{i=1}^{N} \left( \sum_{j=1}^{N} R_i \right)$ for each action.

10 runs, respectively.

We averaged the results over 50,000 runs and averaged the results over numerous independent runs, respectively. The solid circles mark the best performance as a function of the learning rate $\alpha$, each resulting in a reward of -1. The result of an action is a movement of 1 square in the corresponding direction plus an additional movement in the ‘up’ direction, corresponding to a movement of 1 square in the direction determined by the square below it. The reward for reaching the goal is 0, while for all others is -1.

Finally, we present results in other domains verifying the advantages of Expected Sarsa in a broader setting. All results come close to the performance of Expected Sarsa only for values in case of Expected Sarsa and Q-learning. This implies that the Q values of Sarsa diverge.

For completeness, we also show the performance of Q-learning on this problem.
Problem: maximum of estimated values is used as an estimate of the maximum value. This can lead to a positive bias.
Double Q Learning:

- How to fix the maximization bias problem?
- Origin of problem: we are using the same samples to estimate which action value is maximal and what the value of the maximum is
- Solution: use independent sets of samples for both aspects
  - split samples into two independent sets and estimate all values twice: Q1(a) and Q2(a)
  - use, e.g., the Q1 estimates to select the best action and the Q2 estimates to determine the value of this maximum.
Double Q-Learning:

### Double Q-learning

- Initialize $Q_1(s, a)$ and $Q_2(s, a), \forall s \in S, a \in A(s)$, arbitrarily
- Initialize $Q_1(terminal-state, \cdot) = Q_2(terminal-state, \cdot) = 0$

**Repeat (for each episode):**

- Initialize $S$

**Repeat (for each step of episode):**

  - Choose $A$ from $S$ using policy derived from $Q_1$ and $Q_2$ (e.g., $\varepsilon$-greedy in $Q_1 + Q_2$)
  - Take action $A$, observe $R, S'$
  - With 0.5 probability:
    
    $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left( R + \gamma Q_2(S', \text{argmax}_a Q_1(S', a)) - Q_1(S, A) \right)$
  - else:
    
    $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left( R + \gamma Q_1(S', \text{argmax}_a Q_2(S', a)) - Q_2(S, A) \right)$

  - $S \leftarrow S'$

**until $S$ is terminal**
Usually, a state-value function evaluates states in which the agent can take an action.

But sometimes it is useful to evaluate states after agent has acted, as in tic-tac-toe.

Why is this useful?

Two state-action combinations may deterministically lead to the same “afterstate” before the environment responds stochastically. This represents an opportunity for generalization.
Summary

- TD prediction
- Extend prediction to control by employing some form of GPI
  - On-policy control: Sarsa
  - Off-policy control: Q-learning
- These methods *bootstrap* and *sample*, combining aspects of DP and MC methods
- Most widely used RL methods
- So far: *one-step, tabular, model free*
- Possible to use TD methods for pure prediction tasks
Unified View
Chapter 7: Multi-Step Bootstrapping

Only a brief sketch of some key ideas is given here:

- Idea: “interpolate” between Monte Carlo (MC) methods and Temporal Difference (TD) Methods
- Optimal performance is often found in the middle
- allows bootstrapping time interval to be different from (longer than) the time interval of action choice
n-step backups for estimating a state value function

One-step TD and MC lie on opposite ends of a spectrum!
We can also use the backups in between.
7.2. \( n \)-STEP SARSA

Figure 7.2: Performance of \( n \)-step TD methods as a function of \( n \), for various values of \( n \), on a 19-state random walk task (Example 7.1). Note that methods with an intermediate value of \( n \) worked best. This illustrates how the generalization of TD and Monte Carlo methods to \( n \)-step methods can potentially perform better than either of the two extreme methods.

Exercise 7.1 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of \( n \)? How about the change in left-side outcome from 0 to 1 made in the larger walk? Do you think that made any difference in the best value of \( n \)?

7.2 \( n \)-step Sarsa

How can \( n \)-step methods be used not just for prediction, but for control? In this section we show how \( n \)-step methods can be combined with Sarsa in a straightforward way to produce an on-policy TD control method. The \( n \)-step version of Sarsa we call \( n \)-step Sarsa, and the original version presented in the previous chapter we henceforth call one-step Sarsa, or Sarsa(0).

The main idea is to simply switch states for actions (state–action pairs) and then use an "\( \varepsilon \)-greedy policy. The backup diagrams for \( n \)-step Sarsa, shown in Figure 7.3, are like those of \( n \)-step TD (Figure 7.1), strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather a state. We redefine \( n \)-step returns in terms of estimated action values:

\[
G(n)_t = R_{t+1} + R_{t+2} + \cdots + R_{t+n} + \min_{A \in \mathcal{A}} Q_{t+n+1}(S_{t+n+1}, A) - Q_t(S_t, A)
\] (7.4)
n-step SARSA

different variants of SARSA are possible!

R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction
How can this speed up learning a policy?

Path taken

Action values increased by one-step Sarsa

Action values increased by 10-step Sarsa

Figure 7.4: Gridworld example of the speedup of policy learning due to the use of $n$-step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the $G$. In this example the values were all initially 0, and all rewards were zero except for a positive reward at $G$. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and $n$-step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the $n$-step method strengthens the last $n$ actions of the sequence, so that much more is learned from the one episode.
Chapter 8: Planning and Learning with Tabular Methods

Only a brief sketch of some key ideas is given here:

- Idea: with a model you can learn from simulated experience (as we already did in Dynamic Programming)
- Two kinds of models:
  - distribution models: produce full distribution of successor states and rewards (strong)
  - sample models: only produce samples (weak)
- Planning (here): use a model to improve a policy

![Diagram](model -> planning -> policy)
Two “paths” to improve the policy

- Model learning: Using real experience to improve the model.
- Direct reinforcement learning (direct RL): Using real experience directly to improve the value function and policy.

Experience can improve value and policy functions either directly or indirectly via the model. Indirect methods make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. Direct methods are much simpler and are not affected by biases in the design of the model.

Human beings have been criticized for ignoring the indirect methods, with some arguments that they are always superior to direct methods. Others argue that direct methods are responsible for most human and animal learning. Related debates concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision-making. Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained.
Simple Tabular Dyna-Q Algorithm

Tabular Dyna-Q

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) $A \leftarrow \epsilon$-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 $S$
   - $R, S' \leftarrow Model(S, A)$
   - $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
8.2. DYNA: INTEGRATING PLANNING, ACTING, AND LEARNING

Figure 8.3: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps ($n$) per real step. The task is to travel from $S$ to $G$ as quickly as possible.

Agents varied in the number of planning steps, $n$, they performed per real step. For each $n$, the curves show the number of steps taken by the agent in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of $n$, and its data are not shown in the figure. After the first episode, performance improved for all values of $n$, but much more rapidly for larger values. Recall that the $n=0$ agent is a nonplanning agent, utilizing only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values ($\lambda$ and $\gamma$) were optimized for it. The nonplanning agent took about 25 episodes to reach ($\lambda$-optimal performance, whereas the $n=5$ agent took about five episodes, and the $n=50$ agent took only three episodes.

Figure 8.4 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.
Where and when to plan

☐ Where should we plan?
  - where it matters most! If the value of a state has changed much, then the values of possible predecessor states should probably also be backed up -> prioritized sweeping
  - where it is most urgent, i.e., for the current state! We can use planning also as a part of action selection

☐ When should we plan?
  - when we have time for it! E.g., before you make a move in chess. Or when you sleep?