07-280 Notes
Pre-reading: Reinforcement Learning (Overview)

Carnegie Mellon University
School of Computer Science

Contents

1 What Is Reinforcement Learning? **2**

1.1 Setup: MDPs, Returns, and Prince's House 2

2 Planning with a Known Model: Value Iteration **3**

3 Learning from Experience **3**

3.1 Monte Carlo Policy Evaluation 3

3.2 Temporal Difference (TD) Learning 3

3.3 Q-Learning 4

4 Policy Gradients **4**

5 Summary **4**

1 What Is Reinforcement Learning?

In supervised learning, we learn from a dataset of inputs and correct outputs. **Reinforcement learning** (RL) has no answer key. Instead, an **agent** takes **actions** in an **environment**, and after each action it sees a new **state** and a **reward**, a number saying how good that step was. The goal is to learn to choose actions that collect as much reward as possible over time.

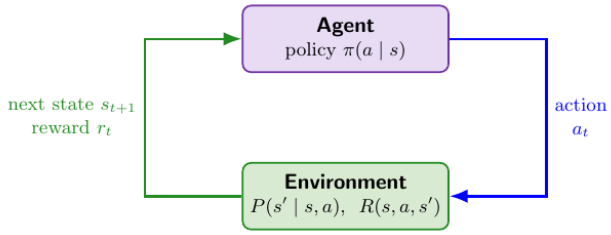


Figure 1: The agent–environment loop.

Unlike labels, rewards don't say what the best action was, they may arrive many steps late, and the data depends on the agent's own actions. (This is a short overview; the Reinforcement Learning course notes have full derivations and more worked examples.)

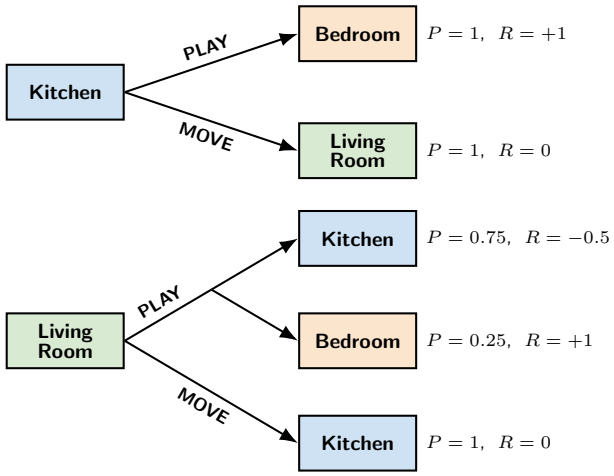
1.1 Setup: MDPs, Returns, and Prince's House

Recap. A **Markov decision process** (MDP) has states $\mathcal{S}$, actions $\mathcal{A}$, transition probabilities $P(s' \mid s, a)$, rewards $R(s, a, s')$, and a discount factor $\gamma \in [0, 1]$. A **policy** $\pi(a \mid s)$ says how the agent picks actions.

The **return** is the discounted sum of rewards, $G = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots$. The goal of RL is a policy with the largest expected return.

Running example. Prince the dog is in the Kitchen (K), Living Room (L), or Bedroom (B, terminal: the episode ends). In K and L he can `Move` or `Play`. Playing in the Kitchen always earns a treat ($+1$) and ends in the Bedroom. Playing in the Living Room is a gamble: 25% of the time a treat ($+1$, Bedroom), 75% of the time he knocks over a lamp (-0.5) and is sent to the Kitchen. Moving walks to the other room with reward 0. We use $\gamma = 0.8$.

s	a	s' (P)	R
K	`Move`	L (1.0)	0
K	`Play`	B (1.0)	$+1$
L	`Move`	K (1.0)	0
L	`Play`	K (0.75)	-0.5
L	`Play`	B (0.25)	$+1$



2 Planning with a Known Model: Value Iteration

$V(s)$ is the expected return from state s, and $Q(s,a)$ is the expected return from taking action a in s. They are linked by the **Bellman equation**:

$$Q(s,a) = \sum_{s'} P(s' \mid s,a)\,[R(s,a,s') + \gamma V(s')]\,, \qquad V(s) = \max_a Q(s,a)$$

In words: *the value of an action is the reward now plus the discounted value of wherever you land*, averaged over where you might land.

If we know P and R, **value iteration** computes the optimal values by starting from $V_0 = 0$ and repeatedly applying the Bellman equation as an update:

$$V_{k+1}(s) = \max_a \sum_{s'} P(s' \mid s,a)\,[R(s,a,s') + \gamma V_k(s')]$$

Example: after one iteration, $V_1(K) = 1$ (play for a treat) and $V_1(L) = 0$. The second iteration for the Living Room compares `Play` and `Move`:

$$V_2(L) = \max\big(\underbrace{0.75\,[-0.5 + 0.8(1)] + 0.25\,[1 + 0.8(0)]}_{\texttt{Play}},\ \underbrace{0.8\,V_1(K)}_{\texttt{Move}}\big) = \max(0.475,\ 0.8) = 0.8$$

The values then stop changing: $V(K) = 1$ and $V(L) = 0.8$. The optimal policy is `Play` in the Kitchen and `Move` in the Living Room: walking to the Kitchen beats the gamble.

3 Learning from Experience

Usually we *don't* know P and R; the agent only sees samples (s, a, r, s') by acting. The methods below are **tabular**: they keep one table entry per state or state-action pair. (The Deep RL prereading replaces the table with a neural network.)

3.1 Monte Carlo Policy Evaluation

To estimate $V^\pi(s)$ for a fixed policy π: run full episodes, compute the actual return G from each visited state, and **average** them. For example, if Prince always plays and the returns observed from L are 0.3, 1, and 0.3, then $V(L) \approx (0.3 + 1 + 0.3)/3 \approx 0.53$. Monte Carlo must wait until an episode ends before updating.

3.2 Temporal Difference (TD) Learning

TD learning also estimates V^π, but updates after *every single transition*. The Bellman equation averages over every possible next state, which needs P; TD replaces that average with the one next state it actually saw, giving the target $r + \gamma V(s')$:

$$V(s) \leftarrow V(s) + \alpha\,[r + \gamma V(s') - V(s)]$$

where α is the learning rate. The bracket is the **TD error** $\delta = r + \gamma V(s') - V(s)$, the "surprise": if things went better than expected ($\delta > 0$), $V(s)$ goes up; if worse, it goes down. The learning rate sets how much to trust one sample: $\alpha = 1$ replaces the old estimate with the new target, while $\alpha = 0$ ignores new samples and never learns. This is a gradient descent step on the squared error $\frac{1}{2}\,(\text{target} - V(s))^2$, the regression loss from earlier in the course. TD **bootstraps**: its target uses the current estimate $V(s')$ instead of waiting for the real return.

Example: with $\alpha = 0.5$, $V(L) = 0.375$, and $V(K) = 0.5$, Prince observes $(L, \texttt{Play}, -0.5, K)$. The target is $-0.5 + 0.8(0.5) = -0.1$, so $V(L) \leftarrow 0.375 + 0.5(-0.1 - 0.375) = 0.1375$.

Monte Carlo vs. TD. This is TD's last update to $V(L)$ on the same three episodes that gave Monte Carlo 0.53 (the true value is 0.475). TD did worse here because it bootstrapped from $V(K) = 0$ before learning the Kitchen is worth about 1. In exchange, TD updates every step instead of waiting for the episode to end; with more data, both find the true value.

3.3 Q-Learning

TD tells us how good a policy is, but not which action to take. **Q-learning** learns $Q(s, a)$ directly, so the best action is just $\text{argmax}_a\, Q(s, a)$. It's the same idea as TD, with $V(s')$ replaced by the best next Q-value:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

The target uses the *largest* Q-value in the next state, even if the agent actually takes a different action next. So Q-learning is **off-policy**: it learns the optimal values while exploring.

Example: $\alpha = 0.5$, $\gamma = 0.8$, $Q(L, \texttt{Play}) = 0$, $Q(K, \texttt{Play}) = 0.5$, $Q(K, \texttt{Move}) = 0$. Prince observes $(L, \texttt{Play}, -0.5, K)$. The target is $-0.5 + 0.8 \max(0.5, 0) = -0.1$, so $Q(L, \texttt{Play}) \leftarrow 0 + 0.5(-0.1 - 0) = -0.05$.

Exploration vs. exploitation. Always taking the best-looking action (**exploitation**) can miss actions that are actually better, so the agent also needs **exploration**. With **ϵ-greedy**, the agent takes a random action with probability ϵ, and otherwise the action with the largest Q-value.

4 Policy Gradients

Instead of learning values, **policy gradient** methods adjust a policy $\pi_\theta(a \mid s)$ with parameters θ (for example, a neural network's weights) by gradient *ascent* on the expected return. The **REINFORCE** algorithm runs the policy, then updates

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a \mid s)\, G$$

Here $\nabla_\theta \log \pi_\theta(a \mid s)$ points toward making action a more likely, and the return G sets how hard to push.

Baselines. If every return is positive, REINFORCE pushes *every* action up, bad ones included. Subtracting a **baseline** $b(s)$, a "normal" return from state s, fixes this: the update is weighted by $G - b(s)$. If an action did better than normal ($G > b$), it becomes more likely; if worse ($G < b$), less likely. A common choice is $b(s) = V(s)$. Then $G - V(s)$ estimates the **advantage** $A(s, a) = Q(s, a) - V(s)$: how much better action a did than the policy does on average from s.

5 Summary

Method	Needs P, R?	Learns
Value iteration	yes	optimal values V^*
Monte Carlo	no	V^π, by averaging full-episode returns
TD learning	no	V^π, updating after each step
Q-learning	no	optimal Q^* (off-policy)
REINFORCE	no	the policy π_θ directly

v1.0