07-280 Notes
Reinforcement Learning: Full Course Notes

Carnegie Mellon University
School of Computer Science

NOTE: This is not the pre-reading. These are the full-length course notes on reinforcement learning, with complete derivations and worked examples, for reference and review. For the required pre-reading (and its quiz), read the shorter *Pre-reading: Reinforcement Learning (Overview)* instead. Everything in the pre-reading is covered here in more depth.

Contents

1 From Supervised Learning to Reinforcement Learning **2**

1.1 Markov Decision Processes (Recap) . . . 2

1.2 Running Example: Prince's House . . . 3

2 Planning: Solving a Known MDP **3**

2.1 Value Functions and Bellman Equations . . . 3

2.2 Value Iteration . . . 4

3 Learning from Experience **5**

3.1 Monte Carlo Policy Evaluation . . . 5

3.2 Temporal Difference (TD) Learning . . . 6

3.3 Q-Learning . . . 7

3.4 Exploration vs. Exploitation . . . 8

4 Policy Gradients **8**

4.1 What Do We Want to Compute? . . . 8

4.2 Why Is This Gradient Tricky? . . . 8

4.3 The Log-Derivative Trick . . . 8

4.4 REINFORCE . . . 9

5 Summary **9**

6 References **9**

1 From Supervised Learning to Reinforcement Learning

So far in this course, someone has always handed us the right answers. We had a dataset of inputs and measured outputs, $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, and we searched for the hypothesis function that minimized the empirical risk:

$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} \hat{R}(h)$$

Reinforcement learning (RL) is *without* these answer keys. Instead, an **agent** takes **actions** in an **environment**. After each action, the environment responds with a new **state** and a **reward**, a single number saying how good that step was. The agent's goal is to learn to choose actions that collect as much reward as possible over time.

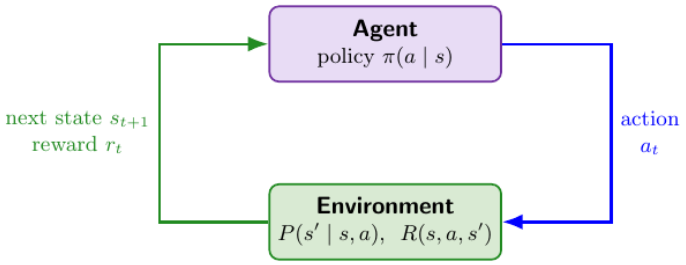


Figure 1: agent–environment loop

For example, the agent might be a game player (reward +1 for a win), a walking robot (reward for distance traveled without falling), or a language model solving math problems (reward 1 for a correct answer, which we'll see in the Deep RL prereading).

Three things make RL different from the supervised learning we've done so far:

1. **Rewards, not labels.** A reward says how good an action was, but not what the best action would have been.
2. **Delayed consequences.** An action now might only pay off many steps later, so the agent has to figure out which earlier actions deserve the credit.
3. **The data depends on the agent.** In supervised learning, the dataset is fixed. In RL, the states the agent sees depend on the actions it chooses, so changing the policy changes the data.

1.1 Markov Decision Processes (Recap)

Quick recap (from the MDP notes). A **Markov decision process** (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ has a set of **states** $\mathcal{S}$, a set of **actions** $\mathcal{A}$, **transition probabilities** $P(s' \mid s, a)$, a **reward** $R(s, a, s')$ for taking action a in state s and landing in s', and a **discount factor** $\gamma \in [0, 1]$. The **Markov property** says the next state depends only on the current state and action, not on the rest of the history. A **policy** $\pi(a \mid s)$ gives the probability of picking action a in state s; a deterministic policy always picks one action, written $\pi(s)$.

Note: some sources, including the deep RL readings we'll see later, write the reward as $r(s, a) = \sum_{s'} P(s' \mid s, a)\, R(s, a, s')$, the *expected* immediate reward for taking action a in state s. It's the same information, averaged over where you land.

Trajectory and return: running a policy in the environment produces a **trajectory**, the sequence of states, actions, and rewards $\tau = (s_0, a_0, r_0, s_1, \ldots)$. The **return** adds up the rewards along the

way, discounted by how far in the future they arrive (here $r_t = R(s_t, a_t, s_{t+1})$ is the reward received at step t):

$$G_0 = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t \geq 0} \gamma^t r_t$$

A treat now is worth more than the same treat later. With γ close to 0 the agent is short-sighted; with γ close to 1 it cares about the far future almost as much as the present.

The RL objective: find the policy with the largest expected return,

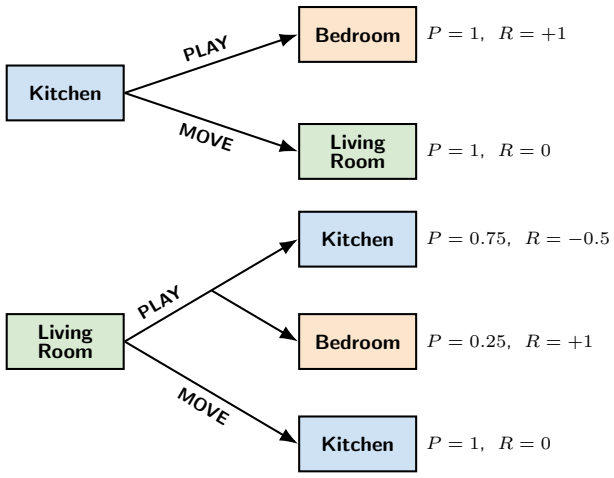
$$J(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t \geq 0} \gamma^t R(s_t, a_t, s_{t+1}) \right], \qquad \pi^* = \operatorname*{argmax}_{\pi} J(\pi)$$

Compare this with empirical risk minimization: in both cases we search over a set of models (hypothesis functions or policies) for the one that optimizes an objective. The differences are that we maximize instead of minimize, and that the expectation is over trajectories generated by the policy itself.

1.2 Running Example: Prince's House

Prince the dog lives in a house with three rooms: the Kitchen (K), the Living Room (L), and the Bedroom (B). In the Kitchen and Living Room he can either `Move` or `Play`. Playing in the Kitchen always earns a treat ($+1$) and then a nap in the Bedroom. Playing in the Living Room is a gamble: 25% of the time he gets a treat and naps, but 75% of the time he knocks over a lamp (-0.5) and gets sent to the Kitchen. Moving just walks him to the other room. The Bedroom is a **terminal state**: once Prince gets there, the episode ends. 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)	$+1$



On average, playing in the Living Room earns $0.75(-0.5) + 0.25(+1) = -0.125$ right away. Every other action has only one possible outcome.

2 Planning: Solving a Known MDP

2.1 Value Functions and Bellman Equations

How good is it to be in a state? **Value functions** answer that.

- $V^\pi(s)$: the expected return if you start in state s and follow policy π.
- $Q^\pi(s, a)$: the expected return if you start in state s, take action a first, and then follow π.

They are connected by the **Bellman equations**:

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

$$V^\pi(s) = \sum_{a} \pi(a \mid s)\, Q^\pi(s,a)$$

In words: *the value of a decision is the reward you get now, plus the discounted value of wherever you land.*

The **optimal** values assume you act as well as possible from here on:

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

And once you know Q^*, acting optimally is easy: pick the action with the largest Q-value.

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

Note: this is why so many RL algorithms try to learn Q. A good Q function hands you a good policy for free.

2.2 Value Iteration

If we know P and R, we can compute the optimal values directly without ever acting in the environment. This is called **planning**. **Value iteration** repeatedly applies the Bellman equation as an update:

1. Initialize $V_0(s) = 0$ for every state.
2. For every state s and action a, compute

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

   and set $V_{k+1}(s) = \max_a Q_{k+1}(s,a)$.
3. Repeat step 2 until the values stop changing.

Then read off the policy with **policy extraction**: $\pi(s) = \operatorname{argmax}_a Q(s,a)$.

Example: value iteration on Prince's house ($\gamma = 0.8$; the Bedroom is terminal, so $V(B) = 0$ always).

Iteration	$V(K)$	$V(L)$	$V(B)$
V_0	0	0	0
V_1	1	0	0
V_2	1	0.8	0
V_3	1	0.8	0

The interesting step is $k = 2$, when the Kitchen's treat first becomes visible from the Living Room (in each max, the first entry is `Play` and the second is `Move`):

$$V_2(K) = \max\big(1,\ 0.8\,V_1(L)\big) = \max(1,\ 0) = 1$$
$$V_2(L) = \max\big(0.75\,[-0.5 + 0.8(1)] + 0.25\,[1 + 0.8(0)],\ 0.8\,V_1(K)\big) = \max(0.475,\ 0.8) = 0.8$$

At $k = 3$ nothing changes (see the table), so we're done. The optimal policy is $\pi^*(K) = \texttt{Play}$ and $\pi^*(L) = \texttt{Move}$. Rather than gamble in the Living Room, it's better to walk to the Kitchen and play there, even though the treat arrives one step later and is worth only 0.8 instead of 1.

Exercise: suppose the lamp gets bolted down, so getting in trouble costs 0 instead of -0.5. What is the best action in the Living Room now? (Answer: `Play`, since $Q(L, \texttt{Play}) = 0.25 + 0.8(0.75)(1) = 0.85 > 0.8$.)

3 Learning from Experience

Value iteration needs the full table of P and R. In the real world, we usually don't have it. Prince doesn't know the odds of knocking over the lamp; he has to try things and see what happens. From now on, all the agent gets is **samples** of experience:

$$(s, a, r, s') = \text{(state, action taken, reward received, next state)}$$

Note: in a sample, r is the reward Prince *actually received*. Playing in the Living Room gives either -0.5 or $+1$, never the average -0.125.

The methods in this section are **tabular**: they keep a table with one entry per state, $V(s)$, or one per state-action pair, $Q(s, a)$, just like the tables we filled in for Prince. The Deep RL prereading replaces the table with a neural network.

3.1 Monte Carlo Policy Evaluation

Goal: estimate V^π for a fixed policy π, using only samples.

The most direct idea: follow π until the episode ends, then compute the actual return G from each state you visited. $V^\pi(s)$ is the *expected* return from s, so averaging the returns you observed from s estimates it. This is **Monte Carlo policy evaluation** ("Monte Carlo" just means estimating something by averaging random samples).

Example: Prince follows the "always play" policy, with $\gamma = 0.8$. We observe three episodes:

- Episode 1: $(L, \texttt{Play}, -0.5, K)$, then $(K, \texttt{Play}, +1, B)$
- Episode 2: $(L, \texttt{Play}, +1, B)$
- Episode 3: $(L, \texttt{Play}, -0.5, K)$, then $(K, \texttt{Play}, +1, B)$

The returns from each visited state are:

- From L: episodes 1 and 3 give $G = -0.5 + 0.8(1) = 0.3$, and episode 2 gives $G = 1$.
- From K: episodes 1 and 3 both give $G = 1$.

Averaging: $V(K) = 1$ and $V(L) = (0.3+1+0.3)/3 \approx 0.53$. The true values for this policy (computed from the Bellman equation with P and R) are $V^\pi(K) = 1$ and $V^\pi(L) = 0.75\,[-0.5 + 0.8(1)] + 0.25\,[1 + 0.8(0)] = 0.475$, so three episodes already get us close.

3.2 Temporal Difference (TD) Learning

Goal: same as Monte Carlo, but update after every single step.

The Bellman equation for V^π uses $\sum_{s'} P(s' \mid s, a)\, V(s')$, an average over every possible next state. We don't know P, but we do know the one next state s' we actually landed in. So we use the **sample target**

$$\text{target} = r + \gamma V(s')$$

as a noisy estimate of what $V(s)$ should be.

How much should we trust one sample? Consider two extremes:

1. **Trust only the new sample**: set $V(s) \leftarrow r + \gamma V(s')$. This throws away everything we learned before, and one unlucky sample (Prince knocks over the lamp) would wreck our estimate.
2. **Ignore the new sample**: keep $V(s)$ as is. This is never fooled by an unlucky sample, but it also never learns anything.

TD learning takes a weighted average of the old estimate and the new target, controlled by the **learning rate** $\alpha \in (0, 1]$:

$$V(s) \leftarrow (1 - \alpha)\, V(s) + \alpha\, [r + \gamma V(s')]$$

(Question: which values of α give the two extremes above?)

Rearranging gives the more common form:

$$V(s) \leftarrow V(s) + \alpha \underbrace{[r + \gamma V(s') - V(s)]}_{\delta \text{ (TD error)}}$$

The **TD error** δ measures the surprise: if things went better than expected ($\delta > 0$), raise $V(s)$; if worse ($\delta < 0$), lower it. The name "temporal difference" comes from δ being a difference between value estimates at two consecutive times, s and s'.

Note: this update is gradient descent in disguise! Treat $V(s)$ as a prediction and $r + \gamma V(s')$ as the target, and use the squared error loss $\ell = \frac{1}{2}\,(\text{target} - V(s))^2$. The gradient with respect to $V(s)$ is $-\,(\text{target} - V(s))$, so a gradient descent step with step size α gives exactly the TD update.

Example: Prince follows the "always play" policy. We use the same three episodes as the Monte Carlo example, with $\alpha = 0.5$, $\gamma = 0.8$, and all values starting at 0 ($V(B)$ stays 0, since the Bedroom is terminal).

Step	Sample	Target $r + \gamma V(s')$	$V(K)$	$V(L)$
–	(initial)	–	0	0
1	$(L, \texttt{Play}, -0.5, K)$	$-0.5 + 0.8(0) = -0.5$	0	-0.25
2	$(K, \texttt{Play}, +1, B)$	$1 + 0.8(0) = 1$	0.5	-0.25
3	$(L, \texttt{Play}, +1, B)$	$1 + 0.8(0) = 1$	0.5	0.375
4	$(L, \texttt{Play}, -0.5, K)$	$-0.5 + 0.8(0.5) = -0.1$	0.5	0.1375
5	$(K, \texttt{Play}, +1, B)$	$1 + 0.8(0) = 1$	0.75	0.1375

Each update is $V(s) \leftarrow V(s) + 0.5\,[\text{target} - V(s)]$:

$$\begin{aligned}
&\text{Step 1: } V(L) \leftarrow 0 + 0.5(-0.5 - 0) = -0.25 \\
&\text{Step 2: } V(K) \leftarrow 0 + 0.5(1 - 0) = 0.5 \\
&\text{Step 3: } V(L) \leftarrow -0.25 + 0.5(1 - (-0.25)) = 0.375 \\
&\text{Step 4: } V(L) \leftarrow 0.375 + 0.5(-0.1 - 0.375) = 0.1375 \\
&\text{Step 5: } V(K) \leftarrow 0.5 + 0.5(1 - 0.5) = 0.75
\end{aligned}$$

Monte Carlo vs. TD: compare with the true values $V^\pi(K) = 1$ and $V^\pi(L) = 0.475$. TD **bootstraps**: its target uses our current estimate $V(s')$ instead of the actual return. That lets it update after every step, but its targets are only as good as those estimates. Here, TD's early updates to $V(L)$ used $V(K) = 0$, before it had learned the Kitchen is worth about 1, so it ended at $V(L) = 0.1375$ while Monte Carlo, which waits for real returns, got 0.53. With more episodes, both settle near the true values.

Note: with a constant α, the estimates keep bouncing forever. Shrinking α over time makes them converge.

3.3 Q-Learning

Instead of learning V, learn Q directly. Then improving the policy needs no model at all: just pick $\text{argmax}_a\, Q(s, a)$.

The **Q-learning** update:

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

This is the TD update with $V(s')$ replaced by $\max_{a'} Q(s', a')$, the value of acting *optimally* from s'. It's the sample-based version of value iteration.

Note: Q-learning is **off-policy**. The target uses the best next action, not the action Prince actually takes next. So Q-learning learns the optimal Q^* even while Prince wanders around trying random things.

Example: $\alpha = 0.5$, $\gamma = 0.8$, all Q-values start at 0, and $Q(B, \cdot) = 0$ since the Bedroom is terminal. Prince explores and we observe these six samples:

Step	Sample	Target	$Q(K, \texttt{Play})$	$Q(K, \texttt{Move})$	$Q(L, \texttt{Play})$	$Q(L, \texttt{Move})$
–	(initial)	–	0	0	0	0
1	$(K, \texttt{Play}, +1, B)$	1	0.5	0	0	0
2	$(L, \texttt{Move}, 0, K)$	0.4	0.5	0	0	0.2
3	$(L, \texttt{Play}, -0.5, K)$	-0.1	0.5	0	-0.05	0.2
4	$(K, \texttt{Move}, 0, L)$	0.16	0.5	0.08	-0.05	0.2
5	$(K, \texttt{Play}, +1, B)$	1	0.75	0.08	-0.05	0.2
6	$(L, \texttt{Move}, 0, K)$	0.6	0.75	0.08	-0.05	0.4

The targets $r + \gamma \max_{a'} Q(s', a')$, using the table values just before each step:

$$\begin{aligned}
&\text{Step 1: } 1 + 0.8 \max_{a'} Q(B, a') = 1 + 0 = 1 \\
&\text{Step 2: } 0 + 0.8 \max\left(Q(K, \texttt{Play}), Q(K, \texttt{Move})\right) = 0.8 \max(0.5, 0) = 0.4 \\
&\text{Step 3: } -0.5 + 0.8 \max(0.5, 0) = -0.1 \\
&\text{Step 4: } 0 + 0.8 \max\left(Q(L, \texttt{Play}), Q(L, \texttt{Move})\right) = 0.8 \max(-0.05, 0.2) = 0.16 \\
&\text{Step 5: } 1 + 0 = 1 \\
&\text{Step 6: } 0 + 0.8 \max(0.75, 0.08) = 0.6
\end{aligned}$$

and each update is $Q(s,a) \leftarrow Q(s,a) + 0.5\,[\text{target} - Q(s,a)]$. For example, step 6 gives $Q(L, \texttt{Move}) \leftarrow 0.2 + 0.5(0.6 - 0.2) = 0.4$.

After just six samples, the greedy policy is `Play` in the Kitchen ($0.75 > 0.08$) and `Move` in the Living Room ($0.4 > -0.05$). That's exactly the optimal policy value iteration found, and Q-learning got there without ever seeing the table of P and R! The values themselves are still below the optimal Q^* values (1, 0.64, 0.475, 0.8); more samples would bring them closer.

3.4 Exploration vs. Exploitation

If Prince always took the action that currently looks best (**exploitation**), an early unlucky result could convince him to never try an action that's actually great. He also needs **exploration**: trying actions just to learn about them.

The simplest fix is **ϵ-greedy**: with probability ϵ, take a random action; otherwise, take the action with the largest Q-value. A common schedule starts with large ϵ (explore a lot early on) and shrinks it over time (exploit once you know more).

Note: tables only work for small problems. Chess has around 10^{43} states, far too many to store a Q-value for each. The Deep RL prereading replaces the table with a neural network.

4 Policy Gradients

TD learning and Q-learning learn *values*, then derive a policy from them. **Policy gradient** methods skip the values and adjust the policy directly.

4.1 What Do We Want to Compute?

We give the policy parameters θ, written $\pi_\theta(a \mid s)$. In deep RL, θ are the weights of a neural network that outputs action probabilities. The objective is the expected return, now written as a function of θ:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\right], \qquad R(\tau) = \sum_t r_t$$

We want to do gradient *ascent*:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

4.2 Why Is This Gradient Tricky?

In supervised learning, the data distribution $\mathcal{D}$ is fixed and θ appears inside the loss, so the gradient moves right inside the expectation:

$$\nabla_\theta \, \mathbb{E}_{z\sim\mathcal{D}}\left[\ell(h_\theta, z)\right] = \mathbb{E}_{z\sim\mathcal{D}}\left[\nabla_\theta \ell(h_\theta, z)\right]$$

In RL it's the other way around. The return $R(\tau)$ doesn't contain θ at all; θ changes *which trajectories happen*. And we can't differentiate through the environment, since we don't even know P.

4.3 The Log-Derivative Trick

(Optional: this derivation is not on the exam.)

The fix uses one identity from calculus, $\nabla_\theta \log p_\theta(x) = \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}$:

$$\begin{aligned}\nabla_\theta \, \mathbb{E}_{x\sim p_\theta}[f(x)] &= \nabla_\theta \sum_x p_\theta(x) f(x) = \sum_x \nabla_\theta p_\theta(x)\, f(x) \\ &= \sum_x p_\theta(x)\, \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)}\, f(x) = \mathbb{E}_{x\sim p_\theta}\left[\nabla_\theta \log p_\theta(x)\, f(x)\right]\end{aligned}$$

Applying this to trajectories τ, the environment's transition probabilities $P(s' \mid s, a)$ don't depend on θ, so they drop out of the gradient. Only the policy's log-probabilities $\log \pi_\theta(a_t \mid s_t)$ remain, so we never need to know P!

4.4 REINFORCE

Putting it together gives the **REINFORCE** algorithm:

1. Run the current policy π_θ to collect N trajectories.
2. Estimate the gradient:

$$\widehat{\nabla J} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) R \left(\tau^{(i)} \right)$$

3. Update $\theta \leftarrow \theta + \alpha \, \widehat{\nabla J}$ and repeat.

Intuition: $\nabla_\theta \log \pi_\theta(a \mid s)$ points in the direction that makes action a more likely. Multiplying by the return means actions from high-return trajectories get pushed up more, and actions from negative-return trajectories get pushed down. It's trial and error, with calculus.

Note: an action at time t can't change rewards that already happened before time t. So in practice we replace $R(\tau)$ with the **reward-to-go** $G_t = \sum_{k \geq t} r_k$, which only counts rewards from time t onward. This gives the same gradient on average, with less noise.

Baselines: if every return is positive, REINFORCE pushes *every* action up, bad ones included. The fix is to subtract a **baseline** $b(s)$, a "normal" return from state s, and use $(G_t - b(s_t))$ in place of G_t. Now an action is pushed up only if it did *better than normal*. A common choice is $b(s) = V^\pi(s)$, which can be learned with TD.

5 Summary

Method	Needs P, R?	Learns	Update
Value iteration	yes	V^*	$V(s) \leftarrow \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V(s') \right]$
Monte Carlo	no	V^π	$V(s) \leftarrow$ average of observed returns G from s
TD learning	no	V^π	$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]$
Q-learning	no	Q^*	$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
REINFORCE	no	π_θ	$\theta \leftarrow \theta + \alpha \, \nabla_\theta \log \pi_\theta(a \mid s) \, (G - b)$

Value iteration, TD learning, and Q-learning all use the Bellman idea: *value now = reward now + discounted value later*. Value iteration uses the exact average over next states; TD and Q-learning replace that average with samples. Monte Carlo skips the Bellman idea and simply averages the returns it actually observed. REINFORCE takes a different route, adjusting the policy directly by making good actions more likely.

6 References

- Williams (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*.
- Watkins & Dayan (1992). Q-learning. *Machine Learning*.

- Sutton & Barto (2018). *Reinforcement Learning: An Introduction*, 2nd edition. MIT Press. (Chapters 5, 6, and 13 cover Monte Carlo, TD, and policy gradients.)

Parts of these notes are adapted from earlier 07-280 MDP and RL notes and from CMU 10-703 (Deep Reinforcement Learning) lecture materials.