Reinforcement Learning Tutorial

Episodic Model-Free Reinforcement Learning with Discrete State/Action Spaces

Freek Stulp, March 2015

Aims of this tutorial

This tutorial is an accompaniment to the course IN104 "project informatique" at ENSTA-ParisTech. The aim of this course is to conduct a mid-size software project. The topic of the project I supervise is reinforcement learning, because there are some interesting software engineering aspects, e.g. understanding the interface/implementation difference by using different agent/environment implementations with the same interface, and using spiral software development cycles to implement increasingly complex agents and environments. Thus, this tutorial is not meant to be a complete, but rather enables students to independently understand those concepts that are required to implement reinforcement learning agents of increasing sophistication.

1 Introduction

In reinforcement learning (RL), an agent must learn to select actions that optimize rewards. This form of learning is especially useful when we have a clear idea of what the outcome of the actions should be, but do not know which actions should be performed to achieve that outcome. For instance, it is clear to me that the outcome of chess is to checkmate the opponent, but that does not mean I always know the appropriate moves to achieve that outcome. Robotics is another good example. When flipping a pancake (`https://www.youtube.com/watch?v=W_gxLKSsSIE`) or playing ball-in-cup (`https://www.youtube.com/watch?v=BeU1KT-wzDM`), we know well *what* the robot needs to do, but it is very difficult to program the robot so that it knows *how* to do it.

In RL, desired outcomes are expressed in terms of rewards. In chess, one could give a big reward for winning the game, and smaller rewards for capturing individual pieces. By using only the points in Atari games as a reward, agents developed by researchers at Google Deepmind are now able to learn sophisticated gaming strategies for playing Atari games using model-free RL: `https://www.youtube.com/watch?v=nwx96e7qck0`.

The basic structure assumed in most RL problem is the **agent-environment interface**. An environment has a certain state (e.g. the configuration of chess pieces on the board), which is (partially) observable to the agent. The agent performs actions (e.g. moves pieces) which change the state of the environment. It receives rewards for certain states (or state-action transitions).

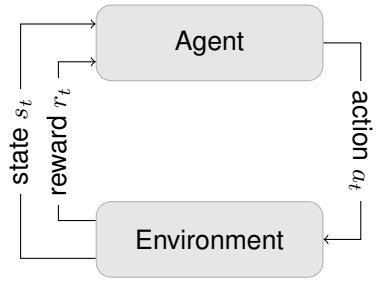


Figure 1: Agent-environment interface

The rest of this tutorial is structured as follows:

2 Problem Formalization: where the agent-environment interface and the RL problem are formally defined

3 State Values: where we explain how the agent is able to estimate the value of being in a certain state

4 State-Action Values: where we explain how the agent is able to estimate the value of performing a certain action in a certain state. This allows the agent to choose the best action in each state.

5 Temporal Differencing: this is considered to be one of the key contributions of RL research. It reformulates the task of estimating values as a recursion, which enables faster learning.

2 Problem Formalization

Two important components of a RL are the possible states of the environment, and the actions the agent can take in this environment. In discrete RL, both the states and actions are finite. In the example below, we see a 2×4 grid world, with 8 states. In each state, the agent can go up, right, down or left, so that there are 4 actions. More formally, we define the **state space** S and **action space** A for this environment as follows:

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$$
$$A = \{a_0, a_1, a_2, a_3\} = \{a_{\text{UP}}, a_{\text{RIGHT}}, a_{\text{LEFT}}, a_{\text{LEFT}}\}$$

s_0	s_1	s_2	s_3
s_4	s_5	s_6	s_7

The **transition function** T defines the effects of an agent's actions on the environment. In the 2D grid above, going down in s_1 will take you to s_5. But going down in s_5 will make you stay in s_5 because you cannot go outside the grid. Formally, the transition function maps a state and an action to a new state $T : S \times A \mapsto S$. For our example, the transition function is:

$$\begin{aligned} T : & S \times A \mapsto S = \{ \\ & T(s_0, a_{\text{UP}}) \mapsto s_0, T(s_0, a_{\text{RIGHT}}) \mapsto s_1, \\ & T(s_0, a_{\text{LEFT}}) \mapsto s_4, T(s_0, a_{\text{LEFT}}) \mapsto s_0, \\ & T(s_1, a_{\text{UP}}) \mapsto s_1, T(s_1, a_{\text{RIGHT}}) \mapsto s_2, \\ & T(s_1, a_{\text{LEFT}}) \mapsto s_5, T(s_1, a_{\text{LEFT}}) \mapsto s_0, \\ & T(s_2, a_{\text{UP}}) \mapsto s_2, \text{etc.} \\ & \} \end{aligned}$$

In this tutorial, we consider **model-free RL**, which means that the agent does not know the transition function of the environment, and thus does not know the effect its actions have on the world. Therefore, it observes numbered states s_i, but does not know how to get from one state to another.

Finally, the **reward function** maps states to rewards. In our example for instance, we may give the agent a reward of 100 for moving into s_0, but a penalty of -1 when it moves to any other state (negative rewards are usually called penalties).

$$R : S \mapsto R = \{R(s_0) \mapsto 100, R(s_{i>0}) \mapsto -1\}$$

Together, the state space S, action space A, transition function T and reward function R, i.e. $\langle S, A, T, R \rangle$, are known as a **Markov Decision Process (MDP)**. 'Markov' refers to the 'Markov property', which poses that future states depend only on the current state, not previous states. In RL, this means that all the relevant information for making decisions is capture in a state s.

The agent determines its actions with its **policy** π, which maps states to actions: $\pi : S \mapsto A$. For instance, an agent may always decide to go right ($\pi(s_*) \mapsto a_{\text{RIGHT}}$), or to always do a random action. Roughly speaking, the

aim of the agent is to determine a policy that optimizes the rewards it can expect to receive.

This policy is known as the **optimal policy**, and is denoted π^*. Because going to s_0 yields a reward of 100 in our example, the optimal policy should clearly choose a_{LEFT} in s_1 rather than for instance a_{RIGHT}, which will lead to a penalty of -1. In subsequent sections, we will define more precisely what the agent should optimize, and how it can determine the optimal policy from interacting with the environment.

In this tutorial, we consider **episodic RL**. This means that the environment contains at least on **terminal state**, in which the interaction with the environment ends. In a marathon for instance, the terminal state is the finish line. After an episode ends, the agent usually starts with a new episode. Reaching the finish line in a marathon doesn't preclude you from running another marathon later on; but you'll have to start at the beginning again.

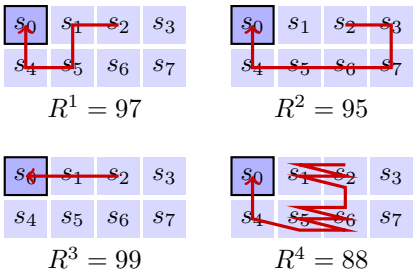
2.1 Summary

- RL problems are often modeled as a Markov Decision Process (MDP), which consists of:
  - the state space S
  - the action space A
  - the transition function $T : S \times A \mapsto S$
  - the reward function $R : S \times A \mapsto \mathcal{R}$
- In episodic RL, interactions with the environment always end in a terminal state
- The agent acts through its policy, which maps states to actions $\pi : S \mapsto A$
- In RL, the agent's aim is to find the optimal policy π^*, which optimizes the future rewards it can expect to receive.

3 State Values (V)

Suppose our agent has a policy that returns randomly chosen actions. Below, we see four possible episodes, in which the agent interacts with the environment until it reaches the terminal state s_0 and the episode ends.

In the upper left episode, the agent receives a penalty of -1 when going to s_1, s_5, s_4, and a reward of 100 when going to the terminal state s_0. The **return of an episode** is defined as the sum over the rewards in an episode: $R = \sum_{t=1}^{N} r_t$, which in this first episode is $R^1 = -1 - 1 - 1 + 100 = 97$.



The returns of the other three episodes, which also all start in s_2, are 95, 99 and 88. The reason that these returns are different is because the agent chooses different random actions in each episode. A fundamental question in RL is: *What is the return we can expect to receive, on average, when starting an episode in a certain state?* If we are able to answer this question, we are very close to our aim of *optimizing* expected returns.

In RL the answer to this question is known as the **value of a state**. In general, the value of a state is defined as the expected return of a state as follows:

$$V^{\pi}(s) = E\{R|s, \pi\} \qquad \text{Value definition} \qquad (1)$$

Given the four episodes above, the average (expected) return is $\frac{97+95+99+88}{4}$, which is 94.

3.1 Monte-Carlo Estimation (Batch)

In the previous section, $V^{\pi}(s_2) = 94$ is not the true expected return. Rather, it is an *estimate* $\overline{V}^{\pi}(s_2)$ based on four observations. Estimating the value function by averaging over actually observed returns[1] leads to a class of algorithms known as a **Monte-Carlo methods**. The pseudo-code for our first algorithm for estimating $V_{\pi}(s)$ using Monte-Carlo estimation is:

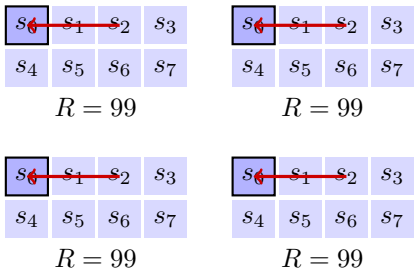
[1] Observed returns are also known as "sampled returns" or "rollouts"

```
# State we want to  determine the value for
state = 2
# To record the sum over the returns of each
# episode
total_returns = 0.0
# Number of episodes
n_episodes = 0

while True:
    # Run episode, starting in state 'state'
    current_return = run_episode(state)
    total_returns += current_return
    n_episodes += 1

    # Value is the average return per episode
    # (gets more and more accurate over time)
    value[state] = total_returns/n_episodes
```

We can estimate the values of the other states also, which yields the value function $V : S \mapsto R$ below[2].

T	85.04	76.04	72.43
90.21	81.85	75.15	72.08

The value function above demonstrates that each state has a different value. But the value also depends on the policies. For instance, consider a policy π_{LEFT} that always returns the action a_{LEFT} (except in s_4, where it returns a_{UP}). If we again run four episodes starting in s_2, we will have the following.



T	100.0	99.0	98.0
100.0	99.0	98.0	97.0

So in each episode, the return is exactly the same: 99. Therefore, the return we can expect to get on average is also 99. And thus the value of this state is $V(s_2) = 99$. For the random policy however, $V(s_2) = 76.04$. Because the value function depends on the policy, we index it with

[2] Note that the terminal state s_0 does not have a value, because we cannot start an episode in a terminal state. Thus, we cannot have an expected return in this state.

the policy, i.e. $V_\pi(s)$. So now we can distinguish between $V_{\pi_{RANDOM}}(s_2) = 76.04$ and $V_{\pi_{LEFT}}(s_2) = 99.0$. The values for all states for π_{LEFT} are shown below:

3.2 Monte-Carlo Estimation (Incremental)

The method for estimating the value function we have used so far is *batch*. That means that it takes a batch of N returns, and estimates the value of a state with:

$$\overline{V}^\pi(s) = \frac{\sum R^1, R^2, R^3, \dots R^N}{N} \quad \text{Batch estimation} \quad (2)$$

An alternative is to use incremental estimation, where we update the estimate of the value incrementally, and throw away the return after it has been used for this update

$$\overline{V}^\pi(s) = \overline{V}^\pi(s) + \alpha[R - \overline{V}(s_t)] \text{ Incremental estim.} \quad (3)$$

Here, $0 < \alpha < 1$ is a learning rate. High values of α lead to faster learning, but may cause undesirable oscillations. The pseudo-code for incremental Monte-Carlo updates is listed below.

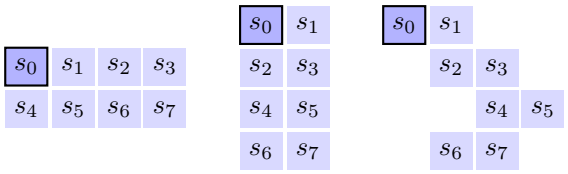
```
# State we want to  determine the value for
state = 2

while True:
    # Run episode, starting in state 'state'
    R = run_episode(state)

    # Incrementally update value with the return
    # (gets more and more accurate over time)
    value[state] += alpha*(R - value[state])
```

The advantages of the incremental method is that no extra storage is required to keep track of the number of updates and the sum over all of the returns. We will see a much more important advantage in Section 5.

3.3 From Values to Policies?

A key idea in RL is that knowing the value function, or at least having a good estimate of it, will allow you to make better decisions in the future. For instance, if you are in s_2, is it better to go to s_1 or s_3? Our intuition says s_1, because it is closer to the terminal state s_0 which yields a big reward of 100. The agent, which does not have intuition, can also come to this conclusion by inspecting its values. And indeed, for the random agent, $V_\pi(s_1) = 85.04$ is larger than $V_\pi(s_3) = 72.43$, so the agent would rather be in s_1.

The problem in model-free RL is that the agent does not know how to get from s_2 to s_1, because it does not know the transition function. In fact, it does not even know that s_2 is a neighbor of s_1. To make the point clear, the agent has no clue as to which environment below it is operating in:



In all the environments, the agent will learn that the value of s_1 is higher than s_3. But since it does not know in which environment it is interacting (it just sees numbered states), it cannot determine which action is required to go from s_2 to s_1 (it is different in each environment above). In a sense, we cheated a bit when visualizing the values in a 2×4 grid, because the agent does not have this information. It should rather store its values in a one-dimensional array.

In fact, the agent does not even know that it is walking around in a grid world; it may even be playing chess, Tetris or an Atari game! It just receives numbered states, and returns numbered actions.

One solution to this problem is to learn not only the values of states, but to also learn the transition function between them. This is called "model learning", and is topic that has been studied within the RL community. But in the next section, we look at another solution, based on learning "Q-values". The "magical" aspect of RL is that we can still learn to act optimally, even if we know nothing about the transition function in the environment.

3.4 Summary

- The value $V^\pi(s)$ is the expected return of a state, given a certain policy
- We can estimate the value $\overline{V}^\pi(s)$ with two Monte-Carlo variants
  - Batch: $\overline{V}^\pi(s) = \frac{\sum R^1, R^2, R^3, \dots R^N}{N}$
  - Incremental: $\overline{V}^\pi(s) = \overline{V}^\pi(s) + \alpha[R - \overline{V}(s_t)]$
- The value function can only be used to change the policy if the agent knows the transition function. In model-free RL this is not the case.

4 State-Action Values (Q)

The value function $V_\pi(s)$ determines the value of *a certain state*. Another approach to the RL problem is to learn the value of *performing a certain action in a certain state*. Such state/action values are known as Q-Values, and the Q-value function is denoted $Q_\pi(s, a)$. Because $Q_\pi(s, a)$ depends on both s and a, we must now make a 2D table of size $|S| \times |A|$, as follows. The values in this table should be read as "$Q_\pi(s_0, a_{\text{LEFT}})$ is the value of executing action a_{LEFT} in state s_0, given the policy π".

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a_{LEFT}	T	100.0	89.0	79.1	89.4	89.0	79.1	70.2
a_{UP}	T	89.1	79.1	70.4	100.0	89.0	79.1	70.2
a_{RIGHT}	T	79.1	70.2	70.3	79.1	70.2	62.1	62.1
a_{LEFT}	T	79.1	70.2	62.2	89.3	79.3	70.2	62.2
		↓	↓	↓	↓	↓	↓	↓
		a_{LEFT}	a_{LEFT}	a_{LEFT}	a_{UP}	a_{LEFT}	a_{LEFT}	a_{LEFT}

We can use Monte-Carlo method to estimate the Q-values also. The rules are:

$$\overline{Q}^\pi(s, a) = \frac{\sum R^1, R^2, R^3, \ldots R^N}{N} \quad \text{Batch} \quad (4)$$

$$\overline{Q}^\pi(s, a) = \overline{Q}^\pi(s, a) + \alpha[R - \overline{Q}(s, a)] \quad \text{Incremental} \quad (5)$$

To update $\overline{Q}^\pi(s_i, a_j)$, we should only use returns R of episodes that started in s_i, and for which the first action was a_j.

How do Q-values solve the problem of knowing what to do to get to s_1 from s_2? Since this table contains the actual Q-values for all possible actions in all possible states, the agent can choose, in each state, the action that has the highest Q-value. For instance, the best action that the agent can choose in s_2 is a_{LEFT}, because it has the highest Q-value (89.0). This will take the agent to s_1, which is closer to the terminal state with the big reward s_0. It is important to realize that the agent does *not know* that a_{LEFT} will take it to s_1 from s_2; it just known that a_{LEFT} is the best thing it can do in s_1, given the Q-values.

The bottom row in the table above shows the action with the highest Q-value: $a = \pi(s_i) = \text{argmax}_a\ Q(s_i, a)$; these actions have been arranged in a 2×4 grid below.

$\pi(s_i) = \text{argmax}_a\ Q(s_i, a)$				
	T	a_{LEFT}	a_{LEFT}	a_{LEFT}
	a_{UP}	a_{LEFT}	a_{LEFT}	a_{LEFT}

As we can see, this is the optimal policy, as each action brings the agent closer to the terminal state where it will receive the reward of 100. Now comes the key insight: by exploring the world randomly, we have been able to learn the Q-values for the random policy. The policy that chooses the action with highest Q-value is then the optimal policy, even though these Q-values were acquired using a random policy. Thus, the agent can learn optimal behavior from random behavior, even if it knows nothing about the environment!

The remaining question is when the agent should switch from random behavior (exploration) to greedy behavior (exploitation). This is discussed in the next section.

4.1 Exploration/Exploitation Trade-off

The exploration/exploitation trade-off is the trade-off between continuing the search for better options, or sticking to the best option you have found so far. Should I go to my favorite restaurant this evening (I know it will be good), or rather try a new one (it may be even better!).

In RL, the trade-off is between acquiring better estimates of the value function, or exploiting the value function to get a higher expected return. Let's make a distinction between two policies, that choose

- a stochastic policy, that always returns a random action (i.e. try a new restaurant)
- a greedy, deterministic policy, that always returns the action with the highest Q-value: $\pi(s) = \text{argmax}_a\ Q(s, a)$ (i.e. go to favorite restaurant)

Using the greedy, deterministic policy at the beginning is not a good idea, because without having yet had any observed returns, it is clear that the estimates of the Q-values are bad estimates. If you have not visited any restaurants yet, you cannot have a favorite one. On the other hand, using only the random strategy all the time is also a bad idea; once you have good estimates of the Q-values, it's better to use them to choose better actions. If you have found a very good restaurant, it's nice to go there more often.

Thus, a smart policy combines the two by starting off being completely random in the beginning (to explore the environment to learn Q-values), and become more greedy over time (to exploit the estimated Q-values). A common strategy for implementing this is the ϵ-*greedy exploration strategy*. It is implemented as follows:

- start with ϵ=1
- in the policy, return a random action with probability ϵ, and a greedy action otherwise
- decrease ϵ over time by setting $\epsilon \leftarrow d\epsilon$ after each return, where $0 < d \leq 1$ is a decay rate. This will make ϵ decay from 1 (random exploration only) to 0 (greedy exploitation only).

If we set $d = 0.5$, random exploration will decay very quickly. If we set it to $d = 0.99$, it will decay very slowly, which is preferable if we expect the agent to require a long time to learn accurate estimates of the Q-values.

4.2 Summary

- Q-values are values for state-action pairs: $Q^\pi(s, a)$
- Estimating the Q-value function can be done using the same methods as learning state values V (batch and incremental Monte-Carlo methods).
- Choosing the action with the highest Q-value allows the agent to improve its policy over time.
- ϵ-greedy exploration combines a random and a greedy policy
- Decaying ϵ allow the agent to go smoothly from random exploration (to learn Q-values) to greedy exploitation (to exploit learned Q-values)

Monte-Carlo Q-value estimation with ϵ-greedy exploration will allow the agent to explore the world, learn values, and improve its policy to maximize future expected returns. We now have a RL agent that can solve RL problems! In the next section, we'll make the agent learn more efficient.

5 Temporal Differencing

All the methods presented so far are Monte-Carlo methods, because they can update values (batch or incrementally) only *after* and episode is done. Temporal difference learning allows us to update values during an episode also, which leads to much quicker learning.

The rules for temporal differencing can be derived directly from the definition of the value in (6). First, we write out in full the return R as being the sum over the rewards r_t in (7), and then rather start the sum at $k = 1$ rather than $k = 0$ (8). We then recognize that the $E\left\{\sum_{k=1}^{T} r_{t+k+1}|s,\pi\right\}$ in (8) corresponds to the definition of the value for the next state s_{t+1}, and we replace $E\left\{\sum_{k=1}^{T} r_{t+k+1}|s,\pi\right\}$ with $V^{\pi}(s_{t+1})$ to acquire (9).

$$
\begin{aligned}
V^{\pi}(s_t) &= E\{R|s_t,\pi\} && (6)\\
&= E\left\{\sum_{k=0}^{T} r_{t+k+1}|s_t,\pi\right\} && (7)\\
&= E\left\{r_{t+1} + \sum_{k=1}^{T} r_{t+k+1}|s_t,\pi\right\} && (8)\\
&= E\{r_{t+1} + V^{\pi}(s_{t+1})|s_t,\pi\} && (9)
\end{aligned}
$$

The last equation is known as the recursive Bellman equation for V^{π}. It is recursive, because $V^{\pi}(s)$ are computed recursively from subsequent values $V^{\pi}(s_{t+1})$. It provides a definition of the value for a certain state. As for Monte-Carlo methods, estimating this value can again be done using an incremental update rule:

$$\overline{V}^{\pi}(s_t) = \overline{V}^{\pi}(s_t) + \alpha[r_{t+1} + \overline{V}(s_{t+1}) - \overline{V}(s_t)] \quad \text{TD Rule} \quad (10)$$

It is important to note that this temporal differencing (TD) rule can be applied *as soon as* a reward has been received. There is no need to wait for the end of the episode to compute the return R as in Monte Carlo, the associated incremental update rule for which we repeat below:

$$\overline{V}^{\pi}(s) = \overline{V}^{\pi}(s) + \alpha[R - \overline{V}(s_t)] \text{ Monte-Carlo} \quad (11)$$

A similar derivation of the Bellman equations may be applied to Q-values. The associated update rule is:

$$
\begin{aligned}
\overline{Q}^{\pi}(s_t,a_t) = \overline{Q}^{\pi}(s_t,a_t) &+\\
\alpha[r_{t+1} + \text{max}_a\overline{Q}(s_{t+1},a) - \overline{Q}(s_t,a_t)] & \quad (12)
\end{aligned}
$$

5.1 Summary

- The definition of the value function enables us to derive the recursive Bellman equation
- Estimating these values can be done using temporal differencing, based on incremental update rules for V and Q
  - $V^{\pi}(s_t) = V^{\pi}(s_t) + \alpha[r_{t+1} + V(s_{t+1}) - V(s_t)]$
  - $Q^{\pi}(s_t,a_t) = Q^{\pi}(s_t,a_t) +$ $\alpha[r_{t+1} + \text{max}_a Q(s_{t+1},a) - Q(s_t,a_t)]$
- The main difference between Monte-Carlo and Temporal Differencing is that MC updates are only possible at the end of an episode (as it is based on the return), whereas TD updates can be applied at each time step (as it is based on individual rewards). In practice, this often leads to faster convergence.

6 Stochastic Environments

In this tutorial, we have assumed that environments are deterministic. This implies that the same action will always have the same effect in the same state. In contrast, many real-world environments are stochastic (non-deterministic). In Markov Decision Processes, this can be modelled by having the transition function map to probabilities, rather than states.

$$T : S \times A \mapsto S \qquad \text{Deterministic} \qquad (13)$$

$$T : S \times A \times S \mapsto [0, 1] \qquad \text{Stochastic} \qquad (14)$$

In the grid world, for instance, it may happen that going left actually sometimes (e.g. 10% of cases) leads the agent to go down. The corresponding transition function for state s_2 in our example would be

$$\begin{aligned} T :& S \times A \times S \mapsto [0, 1] = \{ \\ & \text{(high probability of actually going left)} \\ & T(s_2, a_{\text{LEFT}}, s_1) \mapsto 0.9, \\ & \text{(wanted to go left, but went down...)} \\ & T(s_2, a_{\text{LEFT}}, s_6) \mapsto 0.1, \\ & \text{(this never happens)} \\ & T(s_2, a_{\text{LEFT}}, s_{\{0,2,3,4,5,7\}}) \mapsto 0.0 \\ & \text{etc.} \\ \} & \end{aligned}$$

It is important to note that in stochastic environments, we will get different returns for the same state, even if we use a deterministic policy. Sometimes a_{LEFT} will move you downwards, which will lead to a different route through the grid, and thus a different return. Thus, the expected return $E\{R|s_t, \pi\}$ is an expectation not only over potential stochasticity in the policy, but also over stochasticity in the environment.

Does the difference between having a deterministic or stochastic environment change any of our update rules for the values? Not at all, because we are using model-free RL, which means the agent makes no assumptions at all about the transition function of the environment.

7 Further Reading

This tutorial has focussed on episodic model-free RL with discrete state/action spaces. Some important topics in RL that are beyond the scope of this tutorial include:

- Optimal value functions: the value function for the optimal policy
  http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node35.html
- Bellman optimality principle: in an optimal policy, you must only consider doing the optimal action at the current time step, because all subsequent actions will be optimal also by definition
  http://en.wikipedia.org/wiki/Bellman_equation#Bellman.27s_Principle_of_Optimality
- Curse of dimensionality: the number of discrete state/action pairs grows exponentially with the number of dimensions in the state and action space. Since RL is based on exploring these states, learning may be very slow when high dimensions are involved.
  http://en.wikipedia.org/wiki/Curse_of_dimensionality
- Dynamic Programming: computing values based not on experience, but on models of the environment
  http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node40.html
- Eligibility traces: which form a bridge between Monte Carlo methods and TD(0) as presented in this tutorial.
  http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node72.html
- Direct policy search: searching in the space of policies, without computing a value function
  http://en.wikipedia.org/wiki/Reinforcement_learning#Direct_policy_search
- Continuous state/action spaces: which require function approximation
  http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node85.html

References

[1] M. Harmon. Reinforcement learning: a tutorial, 1996.
http://www.nbu.bg/cogs/events/2000/Readings/Petrov/rltutorial.pdf

[2] R. Sutton and A. Barto. *Reinforcement Learning: an Introduction*. MIT Press, 1998.
http://webdocs.cs.ualberta.ca/~sutton/book/ebook/

[3] V. Mnih, et al. *Human-level control through deep RL*. Nature, 2015. 518, 529–533.