Reinforcement Learning: An Introduction

Zamboni Riccardo

e-Novia, Italy, ricc.zamboni@gmail.com

Keywords: RL, Q-learning, Actor-Critic, Policy Gradient

Abstract

Reinforcement Learning have become a word used by cool kids these days. Here it is a peek in its main ideas, implementations and algorithms, pros and cons and caos.

1 Introduction

Reinforcement Learning (RL) might sound like a mysterious word, as much as its applications have become renowned in even non-technical fields. Its framework is actually pretty simple and general, yet its simplicity stands for its power and curse. We will have a short look inside the main formalisms and concepts and deep dive into some implementations.

2 Reinforcement Learning in a nutshell

2.1 Overview

Let's imagine to be something able to perform actions, hereafter we will call ourselves agent. Let's be clear: here the capacity of taking an action does not assume any specific cognitive capacity in a broader sense, and here comes one of the more fascinating points of RL thinking to me. Let's assume to find ourselves in an unknown environment as well and to be able to obtain some rewards by interacting with such environment. An agent then should take actions so as to maximize such rewards (hopefully), and these can be cumulated over the time. In reality, we could be a basketball-player robotic-arm aiming at getting a score out of a good shot. A positive reward would be given back from the environment once a basket is made, a negative one or nothing at all otherwise. Here, readers might notice that the generating process of the reward, and its design, could be a central part of the design of a RL algorithm, and other reward-based techniques such as LQR. They would be totally right. But unfortunately reward design will not be part of the discussion.

The goal of RL is to learn a good strategy for the agent from trials and relative simple feedback received from the environment. With the optimal strategy, the agent is capable to actively adapt to the environment to maximize future rewards.

In short words: The **agent** is acting in an **environment**. How the environment reacts to certain actions is defined by a **model**. The agent can stay in one of many **states** ($s \in \mathcal{S}$) of the environment, and choose to take one of many **actions** ($a \in \mathcal{A}$) to switch from one state to another, these variables coincides with the s_t and u_t of the LQR framework. Which state the agent will arrive in is decided by **transition probabilities** between states (P). Once an action is taken, the environment gives back a **reward** ($r \in \mathcal{R}$) as feedback.

The agent's **policy** $\pi(s)$ provides the guideline on what is the action to take in a certain state with the goal to maximize the total rewards. Such policy is then a **closed-loop policy** from a control stand-point, similar to an LQR policy. Each state is associated with a **value** function $V(s)$ predicting the expected amount of future rewards we are able to receive in such state by acting with the corresponding policy. In other words, the value function quantifies how potentially good a state is.

Both policy and some sort of value functions are what we try to learn in RL.

The interaction between the agent and the environment involves a sequence of actions and observed rewards in time, $t = 1, 2, \ldots, T$. During the process, the agent accumulates the knowledge about the environment, learns the optimal policy (hopefully), and makes decisions on which action to take next so as to efficiently learn or perform the best policy. Let's label the state, action, and reward at time step t as S_t, A_t, and R_t, respectively. Thus the interaction sequence is fully described by one episode and the sequence might end at the terminal state S_T. This defines the episodic or continuing scenario, based on the existence of a terminal states. As for Model Predictive Control formulations, which can be seen as a RL approach, the existence of a closed solution might be based on whether the task is episodic or not, or on the capacity of the algorithm to deal with non-terminating tasks. An **episode** is then defined as:

$$S_1, A_1, R_2, S_2, A_2, \ldots, S_T$$

2.2 The Environment

The **Environment** defines the reward function and transition probabilities. We may or may not know how the environment can be modeled and then how it works and this gives birth to two circumstances:

- **We know the model**: we will be acting with perfect information, which is often called model-based RL. When we fully know the environment (and theorically we can find the optimal solution by Dynamic Programming).
- **We do not know the model**: We will be acting with incomplete information. Consequently we will be doing model-free RL or try to learn the model explicitly as part

of the algorithm, in order to move back to the model-based framework.

The main role of the model is to describe the environment, thus it has two major parts (for the RL purposes, you may argue): transition probability function P and reward function R. Let's say when we are in state s, we decide to take action a to arrive in the next state s' and obtain reward r. This is known as one transition step, represented by a tuple (s, a, s', r) and the transition function P records the probability of this transition. Denoting with $\mathbb{P}$ the "probability".

$$P(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a] \quad (1)$$

And the **transition probability function** can be defined as a function of $\mathbb{P}(s', r|s, a)$:

$$P^a_{ss'} = \mathbb{P}(s'|s, a) = \sum_{r \in \mathcal{R}} P(s', r|s, a) \quad (2)$$

The **reward function** R predicts the next reward triggered by one action:

$$R(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} P(s', r|s, a) \quad (3)$$

2.3 The Agent

Finally, the **Agent** acts in order to properly define a **policy**, which tells us which action to take in state s. Such policy is a mapping from state s to action a and can be either deterministic or stochastic:

- Deterministic: $\pi(s) = a$
- Stochastic: $\pi(a|s) = \mathbb{P}_\pi[A = a|S = s]$.

The future reward, also known as **return**, is a total sum of discounted rewards going forward. It is in the agent's interest to define a proper return, which can be usefull to learn an optimal policy. Let's compute the return G_t starting from time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (4)$$

The discounting factor $\gamma \in [0, 1]$ penalizes the rewards in the future, because some scenarios might happen:

- The future rewards may have higher uncertainty.
- The future rewards do not provide immediate benefits.
- Discounting provides mathematical convenience; i.e., we don't need to track future steps forever to compute return.
- We don't need to worry about the infinite loops in the state transition graph.

The **state-value** of a state s is the expected return if we are in this state at time t, $S_t = s$:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

Similarly, we define the **action-value** ("Q-value") of a state-action pair as:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

Additionally, when we follow a policy π (called **target policy**), we can make use of the probability distribution over possible actions and the Q-values to recover the state-value:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s, a)\pi(a|s)$$

The difference between action-value and state-value is the action **advantage** function ("A-value"):

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

2.4 Optimal Value and Optimal Policy

Some trivial results and definitions. The optimal value function produces the maximum return:

$$V_*(s) = \max_\pi V_\pi(s), \quad Q_*(s, a) = \max_\pi Q_\pi(s, a) \quad (5)$$

The optimal policy achieves optimal value functions:

$$\pi_* = \arg\max_\pi V_\pi(s), \quad \pi_* = \arg\max_\pi Q_\pi(s, a) \quad (6)$$

And of course, we have $V_{\pi_*}(s) = V_*(s)$ and $Q_{\pi_*}(s, a) = Q_*(s, a)$.

2.5 Markov Decision Processes

Formally, almost all the RL problems can be framed as **Markov Decision Processes** (MDPs). All states in MDPs has "Markov" property, referring to the fact that the future only depends on the current state and not the history:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$$

Or in other words, the current state encapsulates all the information we need to decide the future.

A Markov decision process then consists of a tuple of five elements $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$ well aligned with RL problem settings:

- $\mathcal{S}$ - a set of states;

- $\mathcal{A}$ - a set of actions;
- P - transition probability function;
- R - reward function;
- γ - discounting factor for future rewards.

In an unknown environment, we do not have perfect knowledge about P and R. In such cases these are said to be non-observable variables (as in control theory), or hopefully partially observable portions.

2.6 Bellman Equations

Bellman equations refer to a set of equations used on the value function in order to decompose it in simpler terms.They split the value function into the immediate reward plus the discounted future values:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \end{aligned} \tag{7}$$

And similarly for Q-value,

$$\begin{aligned} Q(s,a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a\sim\pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a] \end{aligned} \tag{8}$$

2.7 Bellman Equations in Expectation

The recursive update process can be further decomposed to be equations built on both state-value and action-value functions. As we go further in future action steps, we extend V and Q alternatively by following the policy π.

$$\begin{aligned} V_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s) Q_\pi(s,a) \\ Q_\pi(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_\pi(s') \\ V_\pi(s) &= \sum_{a\in\mathcal{A}} \pi(a|s)\big(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_\pi(s')\big) \\ Q_\pi(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} \sum_{a'\in\mathcal{A}} \pi(a'|s') Q_\pi(s',a') \end{aligned} \tag{9}$$

2.8 Bellman Optimality Equations

If we are only interested in the optimal values, we actually do not care at all about the expectation, and we could jump right into the maximum returns during the alternative updates without using a policy.

$$\begin{aligned} V_*(s) &= \max_{a\in\mathcal{A}} Q_*(s,a) \\ Q_*(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_*(s') \\ V_*(s) &= \max_{a\in\mathcal{A}} \big(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} V_*(s')\big) \\ Q_*(s,a) &= R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P^a_{ss'} \max_{a'\in\mathcal{A}} Q_*(s',a') \end{aligned} \tag{10}$$

Unsurprisingly they look very similar to Bellman equations in expectation.

If we have complete information of the environment, this is just a planning problem, solvable by Dynamic Programming. Unfortunately, in most scenarios, we do not know $P^a_{ss'}$ or $R(s,a)$, so we cannot solve MDPs by directly applying Bellmen equations, yet these are very important for the RL framework and theoretical foundation of many algorithms.

3 Some Approaches

3.1 Dynamic Programming

When the model is fully known, following Bellman equations, we can use Dynamic Programming, but this is not our aim now so please give a look as [1].

3.1.1 Policy Iteration

This approach is based on two parts:

- Policy Evaluation stands for computing the state-value V_π for a given policy π:

$$\begin{aligned} V_{t+1}(s) &= \mathbb{E}_\pi[r + \gamma V_t(s')|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} P(s', r|s, a)(r + \gamma V_t(s')) \end{aligned} \tag{11}$$

- Based on the value functions, Policy Improvement generates a better policy $\pi' \geq \pi$ by acting greedily.

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= \sum_{s',r} P(s', r|s, a)(r + \gamma V_\pi(s')) \end{aligned} \tag{12}$$

The **Generalized Policy Iteration (GPI)** algorithm refers to an iterative procedure to improve the policy when combining policy evaluation and improvement.

$$\begin{aligned} &\pi_0 \xrightarrow{\text{evaluation}} V_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluation}} V_{\pi_1} \\ &\xrightarrow{\text{improve}} \pi_2 \xrightarrow{\text{evaluation}} \dots \xrightarrow{\text{improve}} \pi_* \xrightarrow{\text{evaluation}} V_* \end{aligned} \tag{13}$$

In GPI, the value function is approximated repeatedly to be closer to the true value of the current policy and meanwhile the policy is improved repeatedly to approach optimality. This policy iteration process works and always converges to the optimality, so that many algorithms try to reproduce such behavior, making use of the same operators (which formally lead to a contraction, but this is a long story and you can check on [2]).

Say, we have a policy π and then generate an i mproved version π' by greedily taking actions, $\pi'(s) = \arg\max_{a\in\mathcal{A}} Q_\pi(s,a)$. The value of this improved π' is guaranteed to be better because:

$$\begin{aligned} Q_\pi(s,\pi'(s)) &= Q_\pi(s, \arg\max_{a\in\mathcal{A}} Q_\pi(s,a)) \\ &= \max_{a\in\mathcal{A}} Q_\pi(s,a) \geq Q_\pi(s,\pi(s)) \\ &= V_\pi(s) \end{aligned} \tag{14}$$

3.2 Monte-Carlo Methods

Monte-Carlo (MC) methods uses a simple idea: It learns from episodes of raw experience without modeling the environmental dynamics and computes the observed mean return as an approximation of the expected return. To compute the empirical return G_t, MC methods need to learn from **complete** episodes $S_1, A_1, R_2, \ldots, S_T$ to compute $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$ and all the episodes must eventually terminate.

The empirical mean return for state s is:

$$V(s) = \frac{\sum_{t=1}^{T} \mathbb{I}[S_t = s] G_t}{\sum_{t=1}^{T} \mathbb{I}[S_t = s]}$$

where $\mathbb{I}[S_t = s]$ is a binary indicator function. We may count the visit of state s every time so that there could exist multiple visits of one state in one episode ("every-visit"), or only count it the first time we encounter a state in one episode ("first-visit"). This way of approximation can be easily extended to action-value functions by counting (s, a) pair.

$$Q(s,a) = \frac{\sum_{t=1}^{T} \mathbb{I}[S_t = s, A_t = a] G_t}{\sum_{t=1}^{T} \mathbb{I}[S_t = s, A_t = a]}$$

To learn the optimal policy by MC, we iterate it by following a similar idea to GPI.

1. Improve the policy greedily with respect to the current value function: $\pi(s) = \arg\max_{a\in\mathcal{A}} Q(s,a)$.

2. Generate a new episode with the new policy π (i.e. using algorithms like epsilon-greedy helps us balance between exploitation and exploration.)

3. Estimate Q using the new episode: $q_\pi(s,a) = \frac{\sum_{t=1}^{T} \left(\mathbb{I}[S_t=s, A_t=a] \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \right)}{\sum_{t=1}^{T} \mathbb{I}[S_t=s, A_t=a]}$

3.3 Temporal-Difference Learning

Similar to Monte-Carlo methods, Temporal-Difference (TD) Learning is model-free and learns from episodes of experience. However, TD learning can learn from **incomplete** episodes and hence we don't need to track the episode up to termination, you can find a long peak at [3].

3.3.1 Bootstrapping

TD learning methods update targets with regard to existing estimates rather than exclusively relying on actual rewards and complete returns as in MC methods. This approach is known as **bootstrapping**.

3.3.2 Value Estimation

The key idea in TD learning is to update the value function $V(S_t)$ towards an estimated return $R_{t+1} + \gamma V(S_{t+1})$ (known as "**TD target**"). To what extent we want to update the value function is controlled by the learning rate hyperparameter α:

$$\begin{aligned} V(S_t) &\leftarrow (1-\alpha)V(S_t) + \alpha G_t \\ V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \\ V(S_t) &\leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \end{aligned} \tag{15}$$

Similarly, for action-value estimation:

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

Next, let's dig into the fun part on how to learn optimal policy in TD learning (aka "TD control"). Be prepared, you are gonna see many famous names of classic algorithms in this section.

3.3.3 SARSA: On-Policy TD control

The name "SARSA" comes from the fact that Q-values are updated according to the following sequence of $\ldots, S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \ldots$. The idea follows the same route of policy iteration:

1. At time step t, we start from state S_t and pick action according to Q values, $A_t = \arg\max_{a\in\mathcal{A}} Q(S_t, a)$;(ϵ-greedy is commonly applied)

2. With action A_t, we observe reward R_{t+1} and get into the next state S_{t+1}.

3. The next actions are selected in the same way as in step 1.: $A_{t+1} = \arg\max_{a\in\mathcal{A}} Q(S_{t+1}, a)$.

4. The action-value function is updated to: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$.

5. t = t+1 and repeat from step 1.

In each update of SARSA, we need to choose actions for two steps by following the current policy twice.

3.3.4 Q-Learning: Off-policy TD control

Q-learning is a big revolution in the early days of RL, being the very first effective off-policy algorithm:

1. At time step t, we start from state S_t and pick action according to Q values, $A_t = \arg\max_{a \in \mathcal{A}} Q(S_t, a)$;
2. With action A_t, we observe reward R_{t+1} and get into the next state S_{t+1}. No next action selection is needed because:
3. The action-value function if updated to: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t))$.
4. t = t+1 and repeat from step 1.

The first two steps are same as in SARSA. In step 3., Q-learning does not follow the current policy to pick the second action but rather estimate Q_* out of the best Q values independently of the current policy.

3.3.5 Deep Q-Network

Theoretically, we can memorize $Q_*(.)$ for all state-action pairs in Q-learning. However, it quickly becomes computationally untractable Thus we make use of functions (i.e. linear regression models, machine learning models) to approximate Q values and this is called **function approximation**. For example, if we use a function with parameter θ to calculate Q values, we can label Q value function as $Q(s, a; \theta)$.

Unfortunately Q-learning may suffer from instability and divergence when combined with an nonlinear Q-value function approximation and bootstrapping, since it can the shown that the function-approximation step is not a contraction.

Deep Q-Network aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

- **Experience Replay**: All the episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = \{e_1, \ldots, e_t\}$. D_t has experience tuples over many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.
- **Periodically Updated Target**: Q is optimized towards target values that are only periodically updated. The Q network is cloned and kept frozen as the optimization target every C steps (C is a hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations.

The loss function looks like this:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \big)^2 \Big] \tag{16}$$

where $U(D)$ is a uniform distribution over the replay memory D; θ^- is the parameters of the frozen target Q-network.

3.4 Policy Gradient

Policy Gradient methods instead learn the policy directly with a parameterized function respect to θ, $\pi(a|s; \theta)$. Let's define the reward function (opposite of loss function) as **the expected return** and train the algorithm with the goal to maximize the reward function.

In discrete space:

$$\mathcal{J}(\theta) = V_{\pi_\theta}(S_1) = \mathbb{E}_{\pi_\theta}[V_1]$$

where S_1 is the initial starting state.

Or in continuous space:

$$\mathcal{J}(\theta) = \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) V_{\pi_\theta}(s) = \sum_{s \in \mathcal{S}} \Big(d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi(a|s, \theta) Q_\pi(s, a) \Big) \tag{17}$$

where $d_{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ.

Using **gradient ascent** we can find the best θ that produces the highest return. It is natural to expect policy-based methods are more useful in continuous space, because there is an infinite number of actions and/or states to estimate the values for in continuous space and hence value-based approaches are computationally much more expensive.

3.4.1 Policy Gradient Theorem

Computing the gradient **numerically** can be done by perturbing θ by a small amount ϵ in the k-th dimension. It works even when $J(\theta)$ is not differentiable (nice!), but unsurprisingly very slow.

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta_k} \approx \frac{\mathcal{J}(\theta + \epsilon u_k) - \mathcal{J}(\theta)}{\epsilon}$$

Or **analytically**,

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi(a|s; \theta) R(s, a)$$

Actually we have nice theoretical support for (replacing $d(.)$ with $d_\pi(.)$):

$$\begin{aligned}\mathcal{J}(\theta) &= \sum_{s\in\mathcal{S}} d_{\pi_\theta}(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta) \\ Q_\pi(s,a) &\propto \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta) Q_\pi(s,a)\end{aligned} \tag{18}$$

Then,

$$\begin{aligned}\mathcal{J}(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta) Q_\pi(s,a) \\ \nabla\mathcal{J}(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \nabla\pi(a|s;\theta) Q_\pi(s,a) \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta) \frac{\nabla\pi(a|s;\theta)}{\pi(a|s;\theta)} Q_\pi(s,a) \\ &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi(a|s;\theta) \nabla \ln\pi(a|s;\theta) Q_\pi(s,a) \\ &= \mathbb{E}_{\pi_\theta}[\nabla \ln\pi(a|s;\theta) Q_\pi(s,a)]\end{aligned} \tag{19}$$

This result is named **Policy Gradient Theorem** which lays the theoretical foundation for various policy gradient algorithms:

$$\nabla\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \ln\pi(a|s,\theta) Q_\pi(s,a)]$$

3.4.2 REINFORCE

REINFORCE, also known as Monte-Carlo policy gradient, relies on $Q_\pi(s,a)$, an estimated return by MC methods using episode samples, to update the policy parameter θ.

A commonly used variation of REINFORCE is to subtract a baseline value from the return G_t to reduce the variance of gradient estimation while keeping the bias unchanged. For example, a common baseline is state-value, and if applied, we would use $A(s,a) = Q(s,a) - V(s)$ in the gradient ascent update.

1. Initialize θ at random
2. Generate one episode $S_1, A_1, R_2, S_2, A_2, \ldots, S_T$
3. For $r_t = 1, 2, ..., T$:
   - Estimate the the return G_t since the time step t.
   - $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla \ln\pi(A_t|S_t,\theta)$.

3.5 Actor-Critic

If the value function is learned in addition to the policy, we would get Actor-Critic algorithm.

- **Critic**: updates value function parameters w and depending on the algorithm it could be action-value $Q(a|s;w)$ or state-value $V(s;w)$.
- **Actor**: updates policy parameters θ, in the direction suggested by the critic, $\pi(a|s;\theta)$.

Let's see how it works in an action-value actor-critic algorithm.

1. Initialize s, θ, w at random; sample $a \sim \pi(a|s;\theta)$.
2. For t = 1… T:
3. Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$.
4. Then sample the next action $a' \sim \pi(s',a';\theta)$.
5. Update policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q(s,a;w)\nabla_\theta \ln\pi(a|s;\theta)$.
6. Compute the correction for action-value at time t: $G_{t:t+1} = r_t + \gamma Q(s',a';w) - Q(s,a;w)$ and use it to update value function parameters: $w \leftarrow w + \alpha_w G_{t:t+1}\nabla_w Q(s,a;w)$.
7. Update $a \leftarrow a' and s \leftarrow s'$.

α_θ and α_w are two learning rates for policy and value function parameter updates, respectively.

References

[1] "Dynamic programming." `https://en.wikipedia.org/wiki/Dynamic_programming`.

[2] `https://web.stanford.edu/class/cme241/lecture_slides/BellmanOperators.pdf`.

[3] `http://incompleteideas.net/book/the-book.html`.