Reinforcement Learning

An Introductory Note

Jingye Wang

wangjy5@shanghaitech.edu.cn

Spring 2020

Contents

1 Introduction

Course Prerequisite:

- Linear Algebra
- Probability
- Machine Learning relevant course (data mining, pattern recognition, *etc*)
- PyTorch, Python

What is Reinforcement Learning and why we care:

a computational approach to learning whereby *an agent* tries to *maximize* the total amount of *reward* it receives while interacting with a complex and uncertain *environment*.[1]

Difference between Reinforcement Learning and Supervised Learning:

- Sequential data as input (*not i.i.d*);
- The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them;
- *Trial-and-error* exploration (balance between exploration and exploitation);
- There is *no supervisor*, only a reward signal, which is also *delayed*

Big deal: Able to Achieve Superhuman Performance

- Upper bound for Supervised Learning is human-performance.
- Upper bound for Reinforcement Learning ?

Why Reinforcement Learning works now?

- Computation power: many GPUs to do trial-and-error rollout;
- Acquire the high degree of proficiency in domains governed by simple, known rules;
- End-to-end training, features and policy are jointly optimized toward the end goal.

Sequential Decision Making:

- Agent and Environment: the agent learns to interact with the environment;
- Rewards: a scalar feedback signal that indicates how well agent is doing;
- Policy: a map function from state/observation to action models the agent’ s behavior;
- Value function: expected discounted sum of future rewards under a particular policy;
- Objective of the agent: selects a series of actions to maximize total future rewards;
- History: a sequence of observations, actions, rewards;
- Full observability: agent directly observes the environment state, formally as Markov decision process (MDP);

- Partial observability: agent indirectly observes the environment, formally as partially observable Markov decision process (POMDP)

All goals of the agent can be described by the maximization of expected cumulative reward.

Types of Reinforcement Learning Agents based on What the Agent Learns

- Value-based agent:
  - Explicit: Value function;
  - Implicit: Policy (can derive a policy from value function);
- Policy-based agent:
  - Explicit: policy;
  - No value function;
- Actor-Critic agent:
  - Explicit: policy and value function.

Types of Reinforcement Learning Agents on if there is model

- Model-based:
  - Explicit: model;
  - May or may not have policy and/or value function;
- Model-free:
  - Explicit: value function and/or policy function;
  - No model.

2 Review of Basic Probability

For more details of this part one can refer to *Introduction to Probability* [2] and *Monte Carlo Statistical Methods* [3].

2.1 Interpretation of Probability

The Frequentist view: *Probability represents a long-run frequency over a large number of repetitions of an experiment.*

The Bayesian view: *Probability represents a degree of belief about the event in question.*

Many machine learning techniques are derived from these two views. As the computing power and algorithms develop, however, Bayesian is becoming dominant.

2.2 Transformations

Change of variables: *Let $\boldsymbol{X} = (X_1, ..., X_n)$ be a continuous random vector with joint PDF $\boldsymbol{X}$, and let $\boldsymbol{Y} = g(\boldsymbol{X})$ where g is an invertible function from $\mathbb{R}^n$ to $\mathbb{R}^n$. Then the joint PDF of $\boldsymbol{Y}$ is*

$$f_{\boldsymbol{Y}}(\boldsymbol{y}) = f_{\boldsymbol{X}}(\boldsymbol{x}) \left| \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}} \right|$$

where the vertical bars say "take the absolute value of the determinant of $\partial \boldsymbol{x}/\partial \boldsymbol{y}$" and $\partial \boldsymbol{x}/\partial \boldsymbol{y}$ is a **Jacobian matrix**

$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} & \cdots & \frac{\partial x_1}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial x_n}{\partial y_1} & \frac{\partial x_n}{\partial y_2} & \cdots & \frac{\partial x_n}{\partial y_n} \end{pmatrix}.$$

It assumes that the determinant of the Jacobian matrix is never 0. It also supposes all the partial derivatives $\frac{\partial x_i}{\partial y_j}$ exist and are continuous.

2.3 Limit Theorem

Strong Law of Large Numbers (SLLN): *The sample mean $\bar{X_n}$ converges to the true mean μ point-wise as $n \to \infty$, w.p.1(i.e. with probability 1). In other words, the event $\bar{X_n} \to \mu$ has probability 1.*

Weak Law of Large Numbers (WLLN): *For all $\varepsilon > 0, P(|\bar{X_n} - \mu| > \varepsilon) \to 0$ as $n \to \infty$. (This form of convergence is called* **convergence in probability** *).*

The Weak Law of Large Numbers can be proved by using **Chebyshev's inequality**.

Central Limit Theorem (CLT): *As $n \to \infty$, $\sqrt{n}\left(\frac{\bar{X_n} - \mu}{\sigma}\right) \to \mathcal{N}(0, 1)$ in distribution. In words, the CDF of the left-hand side approaches the CDF of the standard Normal distribution.*

2.4 Sampling & Monte Carlo Methods

Inverse Transform Method: *Let F be a CDF which is a continuous function and strictly increasing on the support of the distribution. This ensures that the inverse function F^{-1} exists, as a function from $(0,1)$ to* R. *We then have the following results.*

1. *Let $U \sim$ Unif$(0,1)$ and $X = F^{-1}(U)$. Then X is an r.v. with CDF F.*
2. *Let X be an r.v. with CDF F. Then $F(X) \sim$ Unif$(0,1)$.*

Proof:

1. Let $U \sim$ Unif$(0,1)$ and $X = F^{-1}(U)$. Then we have $P(U \leq u) = u$ for $u \in (0,1)$. For all real x,

$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x),$$

so the CDF of X is F, as claimed.

2. Let X have CDF F, and find the CDF of $Y = F(X)$. Since Y takes values in $(0,1)$, $P(Y \leq y)$ equals 0 for $y \leq 0$ and equals 1 for $y \geq 1$. For $y \in (0,1)$,

$$P(Y \leq y) = P(F(X) \leq y) = P(X \leq F^{-1}(y)) = F(F^{-1}(y)) = y.$$

Thus Y has the CDF of Unif$(0,1)$. □

Box-Muller: *Let $U \sim$ Unif$(0, 2\pi)$, and let $T \sim$ Expo(1) be independent of U. Define $X = \sqrt{2T}\cos U$ and $Y = \sqrt{2T}\sin U$. Then X and Y are independent and the joint PDF of $(X,Y)is$*

$$f_{X,Y}(x,y) = \frac{1}{2\pi} e^{\frac{1}{2}(x^2+y^2)}$$

Proof:

The joint PDF of U and T is

$$f_{U,T}(u,t) = \frac{1}{2\pi} e^{-t},$$

for $u \in (0, 2\pi)$ and $t > 0$. And we have the Jacobian matrix

$$\frac{\partial(x,y)}{\partial(u,t)} = \begin{pmatrix} -\sqrt{2t}\sin u & \frac{1}{\sqrt{2t}}\cos u \\ \sqrt{2t}\cos u & \frac{1}{\sqrt{2t}}\sin u \end{pmatrix}.$$

Then we have

$$\begin{aligned} f_{X,Y}(x,y) &= f_{U,T}(u,t) \cdot \left| \frac{\partial(u,t)}{\partial(x,y)} \right| \\ &= \frac{1}{2\pi} e^{-t} \cdot 1 \\ &= \frac{1}{2\pi} e^{-\frac{1}{2}(x^2+y^2)} \\ &= \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \cdot \frac{1}{\sqrt{2\pi}} e^{-y^2/2} \end{aligned}$$

for all real x and y. The joint PDF $f_{X,Y}$ factors into a function x times a function of y, so X and Y are independent. Furthermore, we can find that X and Y are i.i.d. $\mathcal{N}(0,1)$. That shows how the *Box-Muller*

method works for generating Normal r.v.s. □

Monte Carlo Integration: *Given a function* $\Phi : \mathbb{R}^n \to \mathbb{R}$. *If* $p(\cdot)$ *denotes a valid PDF with the support over* $\mathbb{R}^n$, *then we have*

$$\begin{aligned}\int_{\mathbb{R}^n} \Phi(x)dx &= \int_{\mathbb{R}^n} \frac{\Phi(x)}{p(x)} \cdot p(x)dx \\ &= \mathbb{E}_p\left[\frac{\Phi(x)}{p(x)}\right] \\ &\approx \frac{1}{N}\sum_{k=1}^{N} \frac{\Phi(x_k)}{p(x_k)},\end{aligned}$$

where $x_k \sim p$.

By the law of large numbers, the estimator converges to the true value of the integral with probability 1 as $n \to \infty$. Therefore we can use random samples to obtain approximations of definite integrals when exact integration methods are unavailable. Such approach is often referred to as the **Monte Carlo method**.

Importance Sampling: *Let* $p(x)$ *denote the target distribution and* $\mathbb{E}_p[c(x)]$ *is what we want to estimate. With a PDF* $q(x)$ *which subject to that* $\frac{p(x)}{q(x)}$ *is finite for all* $x \in A$, *we have*

$$\begin{aligned}\mathbb{E}_p[c(x)] &= \int_A c(x)p(x)dx \\ &= \int_A c(x) \cdot \frac{p(x)}{q(x)} \cdot q(x)dx \\ &= \mathbb{E}_q\left[c(x) \cdot \frac{p(x)}{q(x)}\right] \\ &\approx \frac{1}{N}\sum_{k=1}^{N} c(x_k)\frac{p(x_k)}{q(x_k)}\end{aligned}$$

where $x_k \sim q$ *and* q *is called* **importance distribution**.

The estimator converges to the true value for the same reason the Monte Carlo method converges. Furthermore, the estimator with importance sampling has the less variance than that of the standard Monte Carlo method.

Acceptance-Rejection Method: *Let* $X \sim p$ *and* $Y \sim q$ *from which we can relatively easily generate samples. Then for a constant* c *such that* $c \geq \sup_\zeta \frac{p(\zeta)}{q(\zeta)}$, *we can simulate* $X \sim p$ *with three steps:*

Step 1: Generate $y \sim q$.

Step 2: Generate $u \sim \mathrm{U}(0, 1)$.

Step 3: If $u \leq \frac{p(y)}{cq(y)}$, *set* $x = y$. *Otherwise go back to Step 1.*

Proof:

This method can be easily proved. From the description, we have

$$\begin{aligned} P(X \leq \zeta) &= P\left(Y \leq \zeta | U \leq \frac{p(y)}{cq(y)}\right) \\ &= \frac{P(Y \leq \zeta, U \leq \frac{p(y)}{cq(y)})}{P(U \leq \frac{p(y)}{cq(y)})} \\ &= \frac{\int_0^{\zeta} \int_0^{\frac{p(y)}{cq(y)}} 1du \cdot q(y)dy}{\int_{-\infty}^{+\infty} \int_0^{\frac{p(y)}{cq(y)}} 1du \cdot q(y)dy} \\ &= \frac{\int_{-\infty}^{\zeta} p(y)dy}{\int_{-\infty}^{+\infty} p(y)dy} \end{aligned}$$

If p is a valid PDF, the denominator will equal 1. Thus we have

$$P(X \leq \zeta) = \int_{-\infty}^{\zeta} p(x)dx,$$

which shows that $X \sim p$. □

2.5 Basic Inequalities

Cauchy-Schwarz Inequality: *For any r.v.s X and Y with finite variances,*

$$|\mathbb{E}[XY]| \leq \sqrt{\mathbb{E}[X^2]\mathbb{E}[Y^2]}.$$

Proof:

For any t,

$$\begin{aligned} \mathbb{E}[(Y - tX)^2] &\geq 0 \\ \mathbb{E}[(Y^2 - 2tXY + t^2X^2)] &\geq 0, \end{aligned}$$

where the left-hand side is a quadratic function with respect to t. To satisfy the inequality, the discriminant of the quadratic must be less than 0, which means

$$\begin{aligned} [2\mathbb{E}[XY]]^2 - 4 \cdot \mathbb{E}[X^2]\mathbb{E}[Y^2] &\leq 0 \\ [\mathbb{E}[XY]]^2 &\leq \mathbb{E}[X^2]\mathbb{E}[Y^2] \\ |\mathbb{E}[XY]| &\leq \sqrt{\mathbb{E}[X^2]\mathbb{E}[Y^2]}. \end{aligned}$$

Therefore we have the Cauchy-Schwarz inequality. □

Jensen's Inequality: *Let X be a random variable. If g is a convex function, then $\mathbb{E}[g(X)] \geq g(\mathbb{E}[X])$. If g is a concave function, then $\mathbb{E}[g(X)] \leq g(\mathbb{E}[X])$. In both cases, the only way that equality can hold is if there are constants a and b such that $g(X) = a + bX$ with probability 1.*

Proof:

If g is convex, then all lines that are tangent to g lie below g. Denoting the tangent line of g by $a + bx$, we have $g(x) \geq a + bx$ for all x by convexity, so $g(X) \geq a + bX$. Taking the expectation of both sides,

$$\begin{aligned}\mathbb{E}[g(X)] &\geq \mathbb{E}[a + bX] \\ &\geq a + b\mathbb{E}[X] \\ &\geq g(\mathbb{E}[X]).\end{aligned}$$

If g is concave, then $h = -g$ is convex, so we can apply the proof to h to see that the inequality for g is reversed from the convex case.

Lastly, assume that $g(X) = a + bX$ holds in the convex case. Let $Y = g(X) - a - bX$. Then Y must be a nonnegative r.v. with $\mathbb{E}[Y] = 0$, so $P(Y = 0) = 1$. So equality holds if and only if $P(g(X) = a + bX) = 1$. For the concave case, we can use the similar argument with $Y = a + bX - g(X)$. □

Norm Inequality: *For a random variable X whose moment of order $r > 0$ is finite, we define the following norm*

$$\|X\|_r = (\mathbb{E}[|X|^r])^{\frac{1}{r}}.$$

With this definition, we have the following inequalities.

- *Holder Inequality*: *Let $\frac{1}{p} + \frac{1}{q} = 1$. If $\mathbb{E}[|X|^p], \mathbb{E}[|X|^q] < \infty$, then $|\mathbb{E}[XY]| \leq \mathbb{E}[|XY|] \leq \|X\|_p \cdot \|Y\|_q$.*
- *Lyapunov Inequality*: *For $0 < r \leq p$, $\|X\|_r \leq \|X\|_p$.*
- *Minkowski Inequality*: *Let $p \geq 1$. If $\mathbb{E}[|X|^p], \mathbb{E}[|Y|^p] < \infty$, then $\|X + Y\|_p \leq \|X\|_p + \|Y\|_p$.*

Markov's Inequality: *For any r.v. X and constant $a > 0$,*

$$P(|X| \geq a) \leq \frac{\mathbb{E}[|X|]}{a}.$$

Proof:

Let $Y = \frac{|X|}{a}$. We need to show that $P(Y \geq 1) \leq \mathbb{E}[Y]$. Note that

$$I(Y \geq 1) \leq Y,$$

taking the expectation of both sides, then we have Markov's Inequality. □

Markov's inequality is a very crude bound because it requires absolutely no assumptions about X. The right-hand side of the inequality could be greater than 1 sometimes, or even infinite.

Chebyshev's Inequality: *Let X have mean μ and variance σ^2. Then for any $a > 0$,*

$$P(|X - \mu| \geq a) \leq \frac{\sigma^2}{a^2}.$$

Proof:

By Markov's inequality,

$$\begin{aligned} P(|X-\mu| \geq a) &= P((X-\mu)^2 \geq a^2) \\ &\leq \frac{\mathbb{E}[(X-\mu)^2]}{a^2} \\ &= \frac{\sigma^2}{a^2}. \end{aligned}$$

□

Chernoff's Inequality: *For any r.v. X and constants $a > 0$ and $t > 0$,*

$$P(X \geq a) \leq \frac{\mathbb{E}[e^{tX}]}{e^{ta}}.$$

Proof:

The transformation g with $g(x) = e^{tx}$ is invertible and strictly increasing. So by Markov's inequality, we have

$$\begin{aligned} P(X \geq a) &= P(e^{tX} \geq e^{ta}) \\ &\leq \frac{\mathbb{E}[e^{tX}]}{e^{ta}}. \end{aligned}$$

□

2.6 Concentration Inequalities

Hoeffding Lemma: *Let X be a r.v. with $\mathbb{E}[X] = 0$, taking values in a bounded interval $[a, b]$, where a and b are constants. Then for any $\lambda > 0$,*

$$\mathbb{E}[e^{\lambda X}] \leq e^{\frac{1}{8}\lambda^2(b-a)^2}.$$

Proof:

For the case $a = b = 0$, we have $P(X = 0) = 1$. The equality holds since both sides of the inequality are 1. Then we consider the case that $a < 0$ and $b > 0$, which is the general case.

Let $f(x) = e^{\lambda x}$ where $x \in [a, b]$. According to its convexity, for any $\alpha \in (0, 1)$, we have

$$f(\alpha a + (1-\alpha)b) \leq \alpha f(a) + (1-\alpha)f(b) = \alpha e^{\lambda a} + (1-\alpha)e^{\lambda b}.$$

As $X \in [a, b]$, let $\alpha = \frac{b-X}{b-a}$, then we have $f(\alpha a + (1-\alpha)b) = f(X) = e^{\lambda X}$. Plugging the two equations into the previous inequality, we have

$$e^{\lambda X} \leq \frac{b-X}{b-a}e^{\lambda a} + \frac{X-a}{b-a}e^{\lambda b}.$$

Taking the expectation of both sides,

$$\mathbb{E}[e^{\lambda X}] \leq \frac{b}{b-a}e^{\lambda a} - \frac{a}{b-a}e^{\lambda b} = e^{\lambda a}\left[\frac{b}{b-a} - \frac{a}{b-a}e^{\lambda(b-a)}\right],$$

and defining a function $\Phi(t) = -\theta t + \ln(1 - \theta + \theta e^t)$ where $\theta = \frac{-a}{b-a} > 0$, we have

$$\mathbb{E}[e^{\lambda X}] \leq e^{\Phi(\lambda(b-a))},$$

as $e^{\Phi(\lambda(b-a))} = e^{\lambda a}\left[\frac{b}{b-a} - \frac{a}{b-a}e^{\lambda(b-a)}\right]$.

We now focus on $\Phi(t)$. According to Taylor expansion, for any $t > 0$, $\exists \tau \in [0, t)$ s.t.

$$\begin{aligned}\Phi(t) &= \Phi(0) + t\Phi'(0) + \frac{1}{2}t^2\Phi''(\tau) \\ &= \frac{1}{2}t^2 \cdot \frac{(1-\theta)\theta e^{\tau}}{(1-\theta+\theta e^{\tau})^2} \\ &\leq \frac{1}{8}t^2\end{aligned}$$

since

$$(1-\theta+\theta e^{\tau})^2 = (1-\theta-\theta e^{\tau})^2 + 4(1-\theta)\theta e^{\tau} \geq 4(1-\theta)\theta e^{\tau}.$$

Plugging in $t = \lambda(b-a)$, we have $\Phi(\lambda(b-a)) \leq \frac{1}{8}\lambda^2(b-a)^2$. It follows that

$$\mathbb{E}[e^{\lambda X}] \leq e^{\frac{1}{8}\lambda^2(b-a)^2}.$$

□

Hoeffding Bound: *Let $X_1, X_2, ..., X_n$ be independent r.v.s with $\mathbb{E}[X_i] = \mu, a \leq X_i \leq b$ for each $i = 1, 2, ..., n$, where a, b are constants. Then for any $\varepsilon \geq 0$,*

$$P\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| \geq \varepsilon\right) \leq 2e^{-\frac{2n\varepsilon^2}{(b-a)^2}}.$$

Proof:

Let $Z_i = X_i - \mu$ and $Z = \frac{1}{n}\sum_{i=1}^{n} Z_i$, then we have $\mathbb{E}[Z_i] = 0$ and $\mathbb{E}[Z] = 0$. For any $\lambda > 0$, we have

$$P(Z \geq \varepsilon) \leq \frac{\mathbb{E}[e^{\lambda Z}]}{e^{\lambda\varepsilon}}$$

by Chernoff's inequality. For $\mathbb{E}[e^{\lambda Z}]$, we have

$$\mathbb{E}[e^{\lambda Z}] = \mathbb{E}[e^{\lambda\frac{1}{n}\sum_{i=1}^{n} Z_i}] = \prod_{i=1}^{n}\mathbb{E}[e^{\frac{\lambda}{n}Z_i}].$$

As $Z_i = X_i - \mu \in [a-\mu, b-\mu]$, using Hoeffding Lemma, we have

$$\prod_{i=1}^{n}\mathbb{E}[e^{\frac{\lambda}{n}Z_i}] \leq e^{\frac{\lambda^2}{8n}(b-a)^2}.$$

Therefore we have

$$P(Z \geq \varepsilon) \leq e^{-\lambda\varepsilon + \frac{\lambda^2}{8n}(b-a)^2}.$$

Now we focus on the quadratic $-\lambda\varepsilon + \frac{\lambda^2}{8n}(b-a)^2$ w.r.t λ. It is easy to find the quadratic has the minimum at $\lambda = \frac{4n\varepsilon}{(b-a)^2}$. Plugging in $\lambda = \frac{4n\varepsilon}{(b-a)^2}$, we have

$$P(Z \geq \varepsilon) \leq e^{\frac{-2n\varepsilon^2}{(b-a)^2}}.$$

Therefore by the symmetry we have

$$P\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| \geq \varepsilon\right) \leq 2e^{\frac{-2n\varepsilon^2}{(b-a)^2}}.$$

□

2.7 Conditional Expectation

"Conditional probabilities are probabilities, and all probabilities are conditional."

For conditional expectation, the case is similar.

Taking out what's known: *For any function h,*

$$\mathbb{E}[h(X)Y|X] = h(X)\mathbb{E}[Y|X].$$

Intuitively, when we take expectations given X, we are treating X as if it has crystallized into a known constant. Then any function of X, say h(X), also acts like a known constant while we are conditioning on X.

Law of Total Expectation (LOTE): *Let $A_1, ..., A_n$ be a partition of a sample space, with $P(A_i) > 0$ for all i, and let Y be a random variable on this sample space. Then*

$$\mathbb{E}[Y] = \sum_{i=1}^{n} \mathbb{E}[Y|A_i]P(A_i).$$

Adam's Law: *For any r.v.s X and Y,*

$$\mathbb{E}[\mathbb{E}[Y|X]] = \mathbb{E}[Y].$$

Proof:

Without loss of generality, we consider the case where X and Y are both discrete. Let $\mathbb{E}[Y|X] = g(X)$. Expanding the definition of $g(x)$ by applying LOTUS, we have

$$\begin{aligned}
\mathbb{E}[\mathbb{E}[Y|X]] &= \mathbb{E}[g(X)] \\
&= \sum_x g(x)P(X = x) \\
&= \sum_x \mathbb{E}[Y|X = x]P(X = x) \qquad (1) \\
&= \sum_x \left(\sum_y yP(Y = y|X = x) \right) P(X = x) \\
&= \sum_y y \sum_x P(Y = y, X = x) \\
&= \sum_y yP(Y = y) \\
&= \mathbb{E}[Y] \qquad (2)
\end{aligned}$$

as desired. Also, as it shown in the proof, with (1) and (2) we can prove the LOTE. □

Adam's law with extra conditioning: *For any r.v.s X, Y, Z we have*

$$\mathbb{E}[\mathbb{E}[Y|X,Z]|Z] = \mathbb{E}[Y|Z].$$

Proof:

Define the expectation $\hat{\mathbb{E}}[\cdot] = \mathbb{E}[\cdot|Z]$. The key is that *"conditional expectation is expectation"*. We have

$$\begin{aligned}
\mathbb{E}[\mathbb{E}[Y|X,Z]|Z] &= \hat{\mathbb{E}}[\hat{\mathbb{E}}[Y|X]] \\
&= \hat{\mathbb{E}}[Y] && \text{(by Adam's Law)} \\
&= \mathbb{E}[Y|Z]
\end{aligned}$$

□

Eve's law: *For any r.v.s X and Y,*

$$Var(Y) = \mathbb{E}[Var(Y|X)] + Var(\mathbb{E}[Y|X]).$$

It is also known as the law of the variance or the variance decomposition formula.

Proof:

Let $g(X) = \mathbb{E}[Y|X]$. By Adam's law, we have $\mathbb{E}[g(X)] = \mathbb{E}[\mathbb{E}[Y|X]] = \mathbb{E}[Y]$. According to the variance of the expectation that

$$Var(Y|X) = \mathbb{E}[Y^2|X] - (\mathbb{E}[Y|X])^2,$$

which can be shown by the way we prove Adam's law, we have

$$\begin{aligned}
\mathbb{E}[Var(Y|X)] &= \mathbb{E}[\mathbb{E}[Y^2|X] - (g(X))^2] \\
&= \mathbb{E}[\mathbb{E}[Y^2|X]] - \mathbb{E}[(g(X))^2] \\
&= \mathbb{E}[Y^2] - \mathbb{E}[(g(X))^2], && (1) \\
Var(\mathbb{E}[Y|X]) &= \mathbb{E}[(g(X))^2] - (\mathbb{E}[\mathbb{E}[Y|X]])^2 \\
&= \mathbb{E}[g(X)^2] - (\mathbb{E}[Y])^2. && (2)
\end{aligned}$$

Then the Eve's law can be shown by $(1) + (2)$. □

3 Bandit Algorithms

The large part of this section was done with references [4, 1, 5, 6, 7].

3.1 Bandit Models

As a special case of Reinforcement Learning, bandit algorithms share some important concepts of that.

Reward: *A reward R_t is a scalar feedback signal which indicates how well agent is doing at step t.*

Reinforcement learning is based on the reward hypothesis that *all goals can be described by the maximization of expected cumulative reward*. Furthermore, Depends on how well we know the reward, there are three types of feedback.

Bandit Feedback: *the agent only knows the reward for the chosen arm;*

Full Feedback: *the agent knows the rewards for all arms that could have been chosen;*

Partial Feedback: *apart from the chosen arm, the agent knows rewards for some arms.*

Besides the feedback, the rewards model also varies, such as *IID rewards, adversarial rewards, constrained adversarial rewards, and stochastic rewards*.

3.2 Stochastic Bandits

The basic settings of stochastic bandits are *bandit feedback* and *IID reward*. Also, we assume that per-round rewards are bounded. A multi-armed bandit $< \mathcal{A}, \mathcal{R} >$ are defined as follows:

- $\mathcal{A}$: a known set of m actions;
- $\mathcal{R}^a(r) = \mathbb{P}(r|a)$: an *unknown* probability distribution over rewards r;
- a_t: the action selected by the agent at time t and $a_t \in \mathcal{A}$;
- r_t: the reward generated by the environment at time t and $r_t \sim \mathcal{R}^{a_t}$;
- $\mathbb{E}\left[\sum_{\tau=1}^{t} r_\tau\right]$: the goal we want to maximize.

For convenience, we define

- $Q(a) = \mathbb{E}[r_t|a_t = a]$: the mean reward for the action a;
- $V^* = \max_{a\in\mathcal{A}} Q(a)$: the optimal reward for the action a;
- $L_\tau = \mathbb{E}[V^* - Q(a_\tau)]$: the regret for the round τ, indicating the opportunity loss.

With those definitions, the goal of the multi-armed bandit $< \mathcal{A}, \mathcal{R} >$ is equivalent to minimize

$$L_t = \mathbb{E}\left[\sum_{\tau=1}^{t}(V^* - Q(a_\tau))\right].$$

That is to say, minimize the total regret we might have by not selecting the optimal action up to the time step t. Another way to formulate the regret is about counting:

$N_t(a) = \sum_{\tau=1}^{t} I_{a_\tau=a}$: the number of selections for the action a after the end of the round t;

$\Delta_a = V^* - Q(a)$: the difference in the value between the action a and the optimal action a^*;

Then the regret follows that

$$L_t = \sum_{a \in \mathcal{A}} \mathbb{E}[N_t(a)]\Delta_a,$$

which is called *Regret Decomposition Lemma*. Depending on how the regret converges, we have

Linear regret: *The regret L_t over t rounds is s.t.*

$$\lim_{t \to \infty} \frac{L_t}{t} = 1.$$

Sublinear regret: *The regret L_t over t rounds is s.t.*

$$\lim_{t \to \infty} \frac{L_t}{t} = 0.$$

Logarithmic asymptotic regret: *The performance of any bandit algorithm is determined by similarity between optimal arm and other arms. Asymptotic total regret is at least logarithmic in number of steps*:

$$\lim_{t \to \infty} L_t \geq \log t \sum_{a | \Delta_a > 0} \frac{\Delta_a}{KL(\mathcal{R}^{\dashv} || \mathcal{R}^*)}.$$

The lower bound is also known as *Lai-Robbins* lower bound. Such lower bound can be obtained with advance knowledge of the gap.

The Exploration-Exploitation Dilemma: *As the action-values are unknown, we must both try actions to learn the action-values (explore), and prefer those that appear best (exploit).*

3.3 Greedy Algorithms

Greedy algorithm tends to take the best action most of the time, while doing exploration occasionally.

Greedy Action: *Define the greedy action at time t as*

$$a_t^{\varepsilon} = \arg\max_{a \in \mathcal{A}} Q(a).$$

If $a_t = a_t^{\varepsilon}$, we are making exploitation, otherwise, we are making exploration, i.e. selecting an arm arbitrarily.

Usually $Q(a)$ is estimated by Monte-Carlo evaluation:

$$\hat{Q}_t(a) = \frac{1}{N_{t-1}(a)} \sum_{\tau=1}^{t-1} r_\tau I(a_\tau = a).$$

ε-greedy algorithm: *Rather than making exploitation every round, we make exploitation with probability $1 - \varepsilon$, or make exploration with probability ε, which usually is a small number.*

Algorithm 1 $\varepsilon-$greedy

1: initialize $\hat{Q}(a) \leftarrow 0,\ N(a) \leftarrow 0,\ \forall a \in \mathcal{A}$;
2: **for** each time slot **do**:
3: $a' \leftarrow \begin{cases} \arg\max_a \hat{Q}(a) & \text{with probability } 1-\varepsilon; \\ \text{a random action} & \text{with probability } \varepsilon; \end{cases}$
4: $r \leftarrow bandit(a')$;
5: $N(a') \leftarrow N(a') + 1$;
6: $\hat{Q}(a') \leftarrow \hat{Q}(a') + \frac{1}{N(a')}(r - \hat{Q}(a'))$;
7: **end for**

In practice, it is useful to initialize $\hat{Q}(a)$ with high values. Such trick is called *optimistic initialization*.

Decaying ε_t-greedy algorithm: *The greedy degree ε in decaying version is not a constant. Rather, we have a decay schedule for ε_t as*

$$\varepsilon_t = \min\left\{1, \frac{c|\mathcal{A}|}{d^2 t}\right\},$$

where $c > 0$ and $d = \min_{a|\Delta a>0} \Delta_i$.

For decaying ε-greedy algorithm, it usually has a better performance than the vanilla version, however, the schedule for ε_t requires advance knowledge of gaps Δ_a.

3.4 UCB Algorithms

In ε-greedy algorithm, we make exploration purely randomly, which may waste the opportunity to try out other options. To avoid such inefficient exploration, UCB algorithm allows us to pick the arm with the highest upper bound with high probability.

UCB algorithm: *Estimate an upper confidence bound $\hat{U}_t(a)$ for each $Q(a)$, i.e.*

$$Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a).$$

Then select actions with max upper confidence bound, i.e.

$$a_t = \arg\max_{a \in \mathcal{A}} \{\hat{Q}_t(a) + \hat{U}_t(a)\}.$$

In other words, UCB algorithm favors exploration of actions with a strong potential to have an optimal value. When we don't have any prior knowledge, the bound can be determined by *Hoeffding's Inequality*. It follows that

$$P(Q(a) > \hat{Q}_t(a) + \hat{U}_t(a)) \leq e^{-2N_t(a)U_t^2(a)}.$$

Since we want to pick a bound so that with high chances the true mean $Q(a)$ is blow the sample mean $\hat{Q}_t(a)$ + the upper confidence bound $\hat{U}_t(a)$, the right-hand side of the inequality should be a small probability, for example, a tiny threshold p, therefore solving for $\hat{U}_t(a)$ we have

$$\hat{U}_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}.$$

The upper bound $\hat{U}_t(a)$ is a function of $N_t(a)$ which means a larger number of trials $N_t(a)$ should give us a smaller bound $\hat{U}_t(a)$. Further, we want the small bound to have a high probability, which we can achieve by designing the p.

UCB1: *As we want to make more confident bound estimation with more rewards observed, the threshold p should be reduced in time. A reasonable idea is setting $p = t^{-4}$ and then we get UCB1 algorithm*

$$\hat{U}_t(a) = \sqrt{\frac{2\log t}{N_t(a)}},$$

$$a_t^{UCB1} = \arg\max_{a \in \mathcal{A}} \left\{ \hat{Q}(a) + \sqrt{\frac{2\log t}{N_t(a)}} \right\}.$$

The regret bound of UCB1 is

$$\lim_{t\to\infty} L_t \leq 8\log t \sum_{a|\Delta_a > 0} \Delta_a.$$

Algorithm 2 UCB1

initialize $\hat{Q}(a) \leftarrow 0,\ N(a) \leftarrow 0,\ \forall a \in \mathcal{A}$;
for each $a \in \mathcal{A}$ **do**:
    $\hat{Q}(a) \leftarrow bandit(a)$;
    $N(a) \leftarrow 1$;
end for
for each time slot **do**:
    $a' \leftarrow \arg\max_a \left(\hat{Q}(a) + c \cdot \sqrt{\frac{2\log t}{N(a)}}\right)$;
    $r \leftarrow bandit(a')$;
    $N(a') \leftarrow N(a') + 1$;
    $\hat{Q}(a') \leftarrow \hat{Q}(a') + \frac{1}{N(a')}(r - \hat{Q}(a'))$;
end for

An intuitive explanation of why UCB works better than greedy algorithm lies in how it handle exploitation *v.s* exploration: when $\hat{Q}_t(a)$ is large, specifically, $\hat{Q}_t(a) \gg \hat{U}_t(a)$, it indicates a high expected reward and leads to exploitation; when $N_t(a)$ is small, concretely, $\hat{Q}_t(a) \ll \hat{U}_t(a)$, it indicates the action may have a great potential and leads to exploration.

So far we have made no assumptions about the reward distribution $\mathcal{R}$, and therefore we have to rely on the Hoeffding's Inequality for a very generalize estimation. If we are able to know the distribution upfront, we would be able to make better bound estimation.

3.5 Thompson Sampling Algorithms

Thompson Sampling algorithm makes use of the posterior to guide exploration. To be specific, at each time step, we want to select action a according to the probability that a is the optimal action:

$$\pi(a|h_t) = P(Q(a) > Q(a'),\ \forall a' \neq a|h_t),$$

where $h_t = a_1, r_1, ..., a_{t-1}, r_{t-1}$ is the history.

Thompson sampling: *Thompson sampling use Bayes' law to compute posterior distribution* $P(\mathcal{R}|h_t)$ *and compute the action-value function* $\hat{Q}(a) = f(\mathcal{R}_a)$*, then it selects the action*

$$a_t^{TS} = \arg\max_{a \in \mathcal{A}} \hat{Q}(a).$$

A naive example for Thompson sampling is for Beta-Bernoulli Bandit. It assumes that the action k being played produces a reward satisfying *Bern*(θ_k), and θ_k follows a *Beta*(α_k, β_k), where θ_k is known but we have α_k, β_k. Let a_t denote the action selected at time t and $r_t \in \{0, 1\}$ denote the corresponding result of action a_t. Then we can update the parameters of the *Beta* distribution

$$(\alpha_k, \beta_k) \leftarrow \begin{cases} (\alpha_k, \beta_k) & \text{if } a_t \neq k, \\ (\alpha_k, \beta_k) + (r_t, 1 - r_t) & \text{if } a_t = k. \end{cases}$$

Algorithm 3 Thompson Sampling

initialize $\alpha_a = |\mathcal{A}|, \beta_a = |\mathcal{A}|, N(a) \leftarrow 0, \ \forall a \in \mathcal{A}$;
for each time slot **do**:
    for each $a \in \mathcal{A}$ **do**:
        Sample $\hat{Q}(a)$ from Beta(α_a, β_a);
        $N(a) \leftarrow 1$;
    end for
    $a' \leftarrow \arg\max_a \left(\hat{Q}(a) + c \cdot \sqrt{\frac{2 \log t}{N(a)}}\right)$;
    $r \leftarrow bandit(a')$;
    $N(a') \leftarrow N(a') + 1$;
    $\hat{Q}(a') \leftarrow \hat{Q}(a') + \frac{1}{N(a')}(r - \hat{Q}(a'))$;
end for

For the details of how it works one can refer to the idea of *Beta Bernoulli Conjugate*. It exploits the power of Bayesian inference to compute the posterior and finally achieves a probability matching. However, for many practical and complex problems, it can be computationally intractable to estimate the posterior distributions with observed true rewards using Bayesian inference. In that case, we may need to approximate the posterior distributions using methods like Gibbs sampling and Laplace approximate.

3.6 Gradient Bandit Algorithms

Previously our bandit algorithms are based on the estimation of action values. while the gradient bandit algorithm is based on a new criterion.

Gradient Bandit Algorithm: *Let* $H_t(a)$ *be a learned preference for taking action* a*, then we have the rules*

$$P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(a),$$

and

$$H_{t+1}(a) = H_t(a) + \alpha(r_t - \bar{r_t})[I(A_t = a) - \pi_t(a)],$$

where $\bar{r_t} = \frac{1}{t}\sum_{i=1}^{t} r_i$ *and the learning rate* $\alpha > 0$.

The algorithm is also known as *stochastic gradient ascent algorithm* since the term $(r_t - \bar{r_t})[I(A_t = a) - \pi_t(a)]$ is essentially the stochastic gradient of $\mathbb{E}[r_t]$ with respect to $H_t(a)$.

Algorithm 4 Gradient Bandit

initialize $R = 0, H(a) = 0,\ \forall a \in \mathcal{A}$;
for each time slot t **do**:
    for each $a \in \mathcal{A}$ **do**:
        $\pi(a) = \frac{H(a)}{\sum_{a'} H(a')}$;
    end for
    Sample action a' from π;
    $r \leftarrow bandit(a')$;
    for each $a \in \mathcal{A}$ **do**:
        $H(a) = H(a) + \alpha(r - R)[I(a' = a) - \pi(a)]$;
        $R \leftarrow \frac{1}{t}[(t-1)R + r]$;
    end for
end for

Actually, gradient bandit algorithm has a bad performance when compared with other algorithms. However, the introduction of the gradient and the function, inspire many policy gradient based algorithms in Reinforcement Learning.

4 Markov Chains

The large part of this section was done with references [4, 2, 3].

4.1 Markov Model

Stochastic Processes: *A stochastic process is a collection, or, a sequence of random variables* $\{X_t, t \in \mathcal{T}\}$. *The set* $\mathcal{T}$ *is the index set of the process. All the r.v.s are defined on a common state space* $\mathcal{S}$.

Markov Property: *Given a stochastic process* $X_0, X_1, X_2, ..., X_n$ *taking values in the state space* $\mathcal{S}$*, the future evolution of the process is independent of the past evolution of the process, i.e.,*

$$P(X_{n+1} = j | X_n = i_n, X_{n-1} = i_{n-1}, ..., X_0 = i_0) = P(X_{n+1} = j | X_n = i_n).$$

The above equation holds for the first order Markov property. For the second order Markov property we have $P(X_{n+1}|X_n, ..., X_0) = P(X_{n+1}|X_n, X_{n-1})$, *etc*. With Markov property, many calculation tasks can be simplified greatly.

Markov Chain/ Process: *A sequence of random variables* $X_0, X_1, X_2, ...$ *taking values in the state space* $\mathcal{S}$ *is called a Markov chain if it has Markov property. A Markov process is the continuous-time version of a Markov Chain.*

Transition Matrix: *For a Markov Chain, let* $q_{ij} = P(X_{n+1} = j | X_n = i)$ *be the transition probability from state* i *to state* j. *Then the matrix* Q *is called the transition matrix of the chain.*

Transition Matrix is a common way to express a Markov chain. Besides that, Markov chain can be represented in a graphical form.

4.2 Basic Computations

With a little abuse of notation, I would use Q to denote the transition matrix when we talk about Markov chain, and $q^n_{i,j}$ to denote the entry $(Q^n)_{i,j}$. Now we introduce some useful computations.

n**-step Transition Probability**: *For a Markov chain, the* n*-step transition probability from* i *to* j *is the probability of being at* j *exactly* n *steps after being at* i*, and*

$$P(X_{n+m} = j | X_m = i) = q^n_{i,j}.$$

Proof:

For a Markov chain, the states are time-homogeneous. Thus we have

$$P(X_{n+m} = j | X_m = i) = P(X_n = j | X_o = i).$$

Hence it follows that

$$\begin{aligned} P(X_n = j|X_0 = i) &= \sum_k P(X_n = j, X_{n-1} = k|X_0 = i) && \text{(by LOTP)} \\ &= \sum_k P(X_n = j|X_{n-1} = k, X_0 = i)P(X_{n-1} = k|X_0 = i) \\ &= \sum_k q_{k,j} P(X_{n-1} = k|X_0 = i), && \text{(by Markov Property)} \end{aligned}$$

then by *induction* from 2 to $n-1$, we have

$$P(X_{n+m} = j|X_m = i) = q_{i,j}^n.$$

□

With n-step transition probability, we have the *Chapman-Kolmogorov Equation*.

Chapman-Kolmogorov Equation: *For $m, n \geq 0$, we have*

$$P(X_{m+n} = j|X_0 = i) = \sum_k P(X_m = k|X_0 = i)P(X_n = j|X_0 = k).$$

The equation can be proved by matrix identity that $q_{i,j}^{m+n} = \sum_k q_{i,k}^m q_{k,j}^n$. By the equation, for a Markov chain with transition matrix P, the Markov property can be generalized to

$$P(X_{n+1} = j|X_{n-m} = i, X_{n-m-1} = i_{n-m-1}, ..., X_0 = i_0) = P(X_{n+1} = j|X_{n-m} = i) = q_{i,j}^{m+1}$$

for $m < n$ and $m \geq 0$.

4.3 Classifications

Depending on whether they are visited over and over again in the long run or are eventually abandoned, the states of a Markov chain can be classified as recurrent or transient.

Recurrent and Transient states: *State i of a Markov chain is recurrent if starting from i, the chain can always return to i. Otherwise, the state is transient, which means that if the chain starts from i, there is a positive probability of never returning to i.*

Irreducible and Reducible Chain: *A Markov chain with transition matrix Q is irreducible if for any two sates i and j, it is possible to go from i to j in a finite number of steps (with positive probability). That is, for any states i, j there is some positive integer n such that the (i, j) entry of Q^n is positive. A Markov chain that is not irreducible is called reducible.*

In an irreducible Markov chain with a finite state space, all states are recurrent.

Period: *For a Markov chain with transition matrix Q, the period of state i, denoted $d(i)$, is the greatest common divisor of the set of possible return times to i. That is,*

$$d(i) = \text{gcd}\{n > 0 \mid q_{i,i}^n > 0\}.$$

If $d(i) = 1$, state i is said to be aperiodic. If the set of return times is empty, set $d(i) = +\infty$.

4.4 Stationary Distribution

Stationary Distribution: *A row vector $\mathbf{s} = (s_1, ..., s_M)$ such that $\sum_i s_i = 1$ is a stationary distribution for a Markov chain with transition matrix Q if*

$$\sum_i s_i q_{i,j} = s_j$$

for all j.

Any irreducible Markov chain has a unique stationary distribution.

Doubly Stochastic Matrix: *A nonnegative matrix such that the row sums and the column sums are all equal to 1 is called a doubly stochastic matrix.*

If the transition matrix Q of a Markov chain is a doubly stochastic matrix, then the uniform distribution over all states, $(1/M, 1/M, ..., 1/M)$, $M = |\mathcal{S}|$, is a stationary distribution of the chain.

Convergence to Stationary Distribution: *Let $X_0, X_1, ...$ be a Markov chain with stationary distribution* **s** *and transition matrix Q, such that some power Q^m is positive in all entries. (These assumptions are equivalent to assuming that the chain is irreducible and aperiodic.) Then $P(X_n = i)$ converges to s_i as $n \to \infty$. In terms of the transition matrix, Q^n converges to a matrix in which each row is* **s**.

Ergodic Markov chain: *A Markov chain is called ergodic if it is irreducible, aperiodic, and all states have finite expected return times (positive recurrent).*

For an ergodic Markov chain $X_0, X_1, ...$, there exists a unique stationary distribution π, which is the limiting distribution of the chain. That is

$$\pi_j = \lim_{n \to \infty} q^n_{i,j}, \ \forall i, j.$$

4.5 Reversibility

Reversibility: *Let Q be the transition matrix of a Markov chain. Suppose there is $\mathbf{s} = (s_1, ..., s_M)$ with $s_i \geq 0, \sum_i s_i = 1$, such that*

$$s_i q_{i,j} = s_j q_{j,i}$$

for all pairs of states i and j.

This equation is called the reversibility or detailed balance condition. We say that the chain is reversible with respect to **s** if it holds, and such **s** is a stationary distribution of the chain.

Detailed Balance Equation: *If for an irreducible Markov chain with transition matrix Q, there exists a probability solution π to the detailed balance equations*

$$\pi_i q_{i,j} = \pi_j q_{j,i}$$

for all pairs of states i and j, then this Markov chain is positive recurrent, time-reversible and the solution π is the unique stationary distribution.

4.6 Markov Chain Monte Carlo

Monte Carlo method is a simulation approach where we generate random values to approximate a quantity. A basic form of such method is directly generating *i.i.d.* draws $X_1, X_2, ..., X_n$ from a given distribution, then by the law of large numbers we can make a desired approximate if n is large. However, staring at a density function does not immediately suggest how to get a random variable with that density.

Fortunately, for this limitation, we have *Markov chain Monte Carlo* (MCMC), a powerful collection of algorithms, to enable us to simulate from complicated distributions using Markov chains. The basic idea is to *build your own Markov chain* so that the distribution of interest is the stationary distribution of the chain.

Convergence to stationary distribution: *Let $X_0, X_1, ...$ be a Markov chain with stationary distribution s and transition matrix Q, such that the chain is irreducible and aperiodic. Then $P(X_n = i)$ converges to s_i as $n \to \infty$.*

With the above theorem, which actually has been mentioned in *ergodic Markov chain*, we can approach the desired s by running our chain for a long time.

Metropolis-Hastings: Metropolis-Hastings allows us to start with any *irreducible* Markov chain on the state space of interest and then modify it into a new Markov chain that has the desired stationary distribution.

Recall: In an irreducible Markov chain, for any two states i and j it is possible to go from i to j in a finite number of steps.

Metropolis-Hastings Algorithm: *Let $s = (s_1, ..., s_M)$ be a desired stationary distribution on state space. Suppose that $Q = q_{ij}$ is the transition matrix for any irreducible Markov chain on state space $\{1, ..., M\}$. Then we can use a chain with transition matrix Q to construct a collection of states sample $X_0, X_1, ...$ with stationary distribution s.*

Algorithm 5 Metropolis-Hastings

1: **input** the desired distribution $s = (s_1, ..., s_M)$; the chain with transition matrix Q; the initial state X_0;
2: **for** $n = 0, 1, ...$ **do**: *# assume that* $X_n = i$;
3: Sample the next state j according to Q;
4: Calculate the acceptance probability $a_{ij} = \min\left(\frac{s_j q_{ji}}{s_i q_{ij}}, 1\right)$;
5: $X_{n+1} \leftarrow \begin{cases} j & \text{with probability } a_{ij}; \\ i & \text{with probability } 1 - a_{ij}; \end{cases}$
6: **end for**

In practice, a useful trick is *Burn-in*, which discards the initial iterations and retains $X_m, X_{m+1}, ...$ for some m. The key of the algorithm is that the moves are proposed according to the original chain, but the proposal may or may not be accepted. By the *reversibility condition*, it can be showed that the sequence $X_0, X_1, ...$ constructed by the Metropolis-Hastings algorithm is a *reversible* Markov chain with stationary distribution s.

Gibbs Sampler: Gibbs sampling is an MCMC algorithm for obtaining approximate draws from a joint distribution, based on sampling from conditional distributions one at a time: at each stage, one variable is updated (keeping all the other variables fixed) by drawing from the conditional distribution of that variable given all the other variables.

Gibbs sampler: *Let X and Y be discrete r.v.s with joint PMF $p_{X,Y}(x, y) = P(X = x, Y = y)$. We wish to construct a two-dimensional Markov chain (X_n, Y_n) whose stationary distribution is $p_{X,Y}$. The systematic scan Gibbs sampler proceeds by updating the X-component and the Y-component in alternation. If the current state is $(X_n, Y_n) = (x_n, y_n)$, then we update the X-component while holding the Y-component fixed, and then update the Y -component while holding the X-component fixed.*

Algorithm 6 Gibbs Sampling

1: **input** the desired joint distribution $P(X, Y)$; the initial state X_0, Y_0;
2: **for** $n = 0, 1, ...$ **do**:
3: Sample the next state X_{n+1} from $P(X, Y = Y_n)$;
4: Sample the next state Y_{n+1} from $P(X = X_{n+1}, Y)$;
5: **end for**

The algorithm can be generalized to high dimensional easily in the light of line 3 and 4.

5 Markov Decision Process

The large part of this section was done with references [4, 8, 9, 1].

5.1 Markov Reward Process

Markov Reward Process (MRP): *A Markov Reward Process is a tuple* $\langle \mathcal{S}, \mathcal{P}, R, \gamma \rangle$ *where*

- $\mathcal{S}$: the (finite) set of states;
- $\mathcal{P}$: the state transition probability matrix, where $\mathcal{P}_{ss'} = P(S_{t+1} = s' | S_t = s)$ is the probability of arriving s' from s;
- R: the random variable or function of rewards, and we use R_t to represent the reward at time t;
- γ : the discount factor.

An MRP chain is a Markov chain with values, and we can define *return* as follows.

Return: *Suppose the reward at time* t *is* R_t*, then the return* G_t *is the total discounted reward from time-step* t*:*

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $\gamma \in (0, 1)$ *is the discount factor.*

There are many reasons to consider discounted rewards. One reason is that uncertainty about the future may not be fully represented. And for the case that the reward is financial, immediate rewards may earn more interest than delayed rewards. Besides, animal and human behavior shows preference for immediate reward.

Value Function: *The state value function* $v(s)$ *of an MRP is the expected return starting from state* s

$$v(s) = \mathbb{E}[G_t | S_t = s].$$

Bellman Equation for MRPs: *The value function* $v(s_t)$ *can be decomposed into two parts: the immediate reward* R_{t+1} *and the discounted value of successor state, which is*

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s].$$

Proof:

Notice that S is a random variable of states, and s is an instance of states. According to the Adam's Law with extra conditioning, it follows that

$$\mathbb{E}[\mathbb{E}[G_{t+1} | S_{t+1}, S_t] | S_t] = \mathbb{E}[G_{t+1} | S_t]. \tag{1}$$

By Markov property, the term $(G_{t+1}|S_{t+1}, S_t)$ equals $(G_{t+1}|S_{t+1})$. Therefore we arrive at

$$\begin{aligned}\mathbb{E}[\mathbb{E}[G_{t+1}|S_{t+1}, S_t]|S_t] &= \mathbb{E}[\mathbb{E}[G_{t+1}|S_{t+1}]|S_t] \\ &= \mathbb{E}[v(S_{t+1})|S_t]. \end{aligned} \tag{2}$$

With (1) and (2), we can make a conclusion that

$$\mathbb{E}[G_{t+1}|S_t] = \mathbb{E}[v(S_{t+1})|S_t]. \tag{3}$$

Then we have

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma\mathbb{E}[G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma\mathbb{E}[v(S_{t+1})|S_t = s] && \text{(by (3))} \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \end{aligned}$$

□

We can also derive another form of Bellman equation by the law of total expectation:

$$\begin{aligned} v(s) &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma\mathbb{E}[v(S_{t+1})|S_t = s] \\ &= R_s + \gamma \sum_{s' \in S} \mathbb{E}[v(S_{t+1})|S_t = s, S_{t+1} = s']\mathcal{P}_{ss'} && \text{(by LOTE)} \\ &= R_s + \gamma \sum_{s' \in S} v(s')\mathcal{P}_{ss'}, \end{aligned}$$

Though Bellman equation is elegant, to solve it directly is only possible for small MRPs.

5.2 Markov Decision Process

Markov Decision Process (MDP): *A Markov Decision Process is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma \rangle$*, where*

- $\mathcal{S}$: the (finite) set of states;
- $\mathcal{A}$: the (finite) set of actions;
- $\mathcal{P}$: the state transition probability matrix, where $\mathcal{P}^a_{ss'} = P(S_{t+1} = s'|S_t = s, A_t = a)$ is the probability of arriving s' by taking action a from s;
- R: the random variable or function of rewards, besides the notation we defined in MRP, we also use R^a_s to denote the reward we get by taking action a at state s;
- γ : the discount factor.

An MDP chain is an MRP chain with decisions (actions). It is an environment in which all states are Markovian.

Policy: *A policy* π *is a distribution over actions given states,*

$$\pi(a|s) = P(A_t = a|S_t = s).$$

A policy can fully define the behavior of an agent, and it depends only on the current state. Similar to MRP, we have the following definitions.

State Value Function: *The state value function $v^{\pi}(s)$ of an MDP is the expected return starting from state s, and following policy π*

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

State-action Value Function: *The state-action value function $q^{\pi}(s, a)$ is the expected return starting from state s, taking action a, and then following policy π, which is*

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

According to the definition, we have

$$v_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q^{\pi}(s, a),$$

$$q^{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v^{\pi}(s').$$

Then the relation between $v^{\pi}(s)$ and $q^{\pi}(s, a)$ follows that

$$v^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v^{\pi}\left(s'\right) \right),$$

$$q^{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi\left(a'|s'\right) q^{\pi}\left(s', a'\right).$$

And we can rewrite the above equations to get *Bellman expectation equation*.

Bellman Expectation Equation: *The value function can be decomposed into immediate reward plus discounted value of the successor state, that is to say,*

$$v^{\pi}(s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma v^{\pi}(S_{t+1})|S_t = s],$$

$$q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_{t+1} + \gamma q^{\pi}(S_{t+1}, A_{t+1})|S_t = t, A_t = a].$$

Optimal State Value Function: *The optimal state value function $v_*(s)$ is the maximum value function over all polices:*

$$v^*(s) = \max_{\pi} v^{\pi}(s).$$

Optimal State-action Value Function: *The optimal state-action value function $q_*(s, a)$ is the maximum value function over all polices:*

$$q^*(s, a) = \max_{\pi} q^{\pi}(s, a).$$

Optimal Policy: *For any Markov Decision Process, there exists an optimal policy π_* that is better than or equal to all other policies. Further, all optimal policies achieve the optimal value function, $v^{\pi_*}(s) = v_*(s)$ and the optimal state-action value function, $q^{\pi_*}(s,a) = q_*(s,a)$.*

We say that an MDP is *solved* when we know the optimal value function. Since the optimal policy can be derived from the optimal value function.

Prediction: *Given an MDP and a policy π, find the value function v^π and q^π.*

Control: *Given an MDP, find the optimal policy π^*.*

Actually, the term *prediction* can be replaced with *evaluation* in many cases. Hope it would not make you confused. For an unknown MDP, we have no idea about its transition matrix and its state, action spaces, thus the methods we mentioned in previous sections fail to work. What we can do is interacting with the environment and collecting episodes.

5.3 Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into subproblems. Specifically, the problems should be characterized by the two properties:

- Optimal substructure, which allows that optimal solution can be decomposed into subproblems;
- Overlapping subproblems, which entails that the solutions for subproblems can be cached and reused.

Obviously, MDPs satisfy both properties. Evaluation and control in MDP can be solved by dynamic programming.

5.3.1 Policy Evaluation

Bellman expectation backup: *Given a policy π, at each iteration $k+1$, for all states $s \in \mathcal{S}$, update $v_{k+1}(s)$ from $v_k(s')$ by*

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right).$$

Clearly, $v_k = v^\pi$ is a fixed point for this update rule because the Bellman equation for v^π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v^π as $k \to \infty$ under the same conditions that guarantee the existence of v^π. This algorithm is called iterative policy evaluation.

Algorithm 7 Iterative Policy Evaluation

initialize $v(s)$ arbitrarily for $s \in \mathcal{S}$ except that $v(terminal) = 0$;
input the policy π to be evaluated; the threshold $\theta > 0$ determining the accuracy of the evaluation;
for true **do**:
    $\Delta \leftarrow 0$;
    for each $s \in \mathcal{S}$ **do**:
        $old \leftarrow v(s)$;
        $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'}[R^a_s + \gamma v(s')]$;
        $\Delta \leftarrow \max(\Delta, |old - v(s)|)$;
    end for
    if $\Delta < \theta$ **then**:
        break;
    end if
end for

Formally, iterative policy evaluation converges only in the limit, but in practice it is better to be halted short of this.

5.3.2 Policy Iteration

For a deterministic policy $a = \pi(s)$, we can iteratively improve it through the two steps:

- Evaluate the policy π so that get the value function;
- Improve the policy by acting greedily for all $s \in \mathcal{S}$ with respect to the value function:

$$\pi'(s) = \arg\max_a q^\pi(s, a).$$

Proof:

Now we show that such improvement does give us a better policy.

$$\begin{aligned} v^\pi(s) &\leq \max_a q^\pi(s, a) \\ &= q^{\pi'}(s, \pi'(s)) \\ &= E_{\pi'}[R_{t+1} + \gamma v^\pi(S_{t+1})|S_t = t] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q^{\pi'}(S_{t+1}, \pi'(S_{t+1}))|S_t = t] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q^{\pi'}(S_{t+2}, \pi'(S_{t+2}))|S_t = s] \quad (1) \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + ...|S_t = s] \\ &= v^{\pi'}(s). \end{aligned}$$

The step (1) can be derived by Adam's law with extra conditioning and Markov theory. □

When the improvement converges, we have

$$q^{\pi}(s, \pi'(s)) = \max_{a} q^{\pi}(s, a) = q^{\pi}(s, \pi(s)) = v^{\pi}(s),$$

which gives us the optimal policy.

Algorithm 8 Policy Iteration

initialize a policy $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$;
for true **do**:
    $v \leftarrow$Policy Evaluation for π;
    $stable \leftarrow$ true;
    for each $s \in \mathcal{S}$ **do**:
        $old \leftarrow \pi(s)$;
        $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[R^a_s + \gamma v(s')]$;
        if $old \neq \pi(s)$ **then**:
            $stable \leftarrow$false;
        end if
    end for
    if $stable$ **then**:
        Return π
    end if
end for

It should be noticed that here we are talking about deterministic policy. For stochastic policy, expectation should be introduced.

Generalized Policy Iteration(GPI): *The combination of policy evaluation and policy improvement is called generalized policy iteration.*

In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.

5.3.3 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. After the value function converges, the change of the policy will lead to a new round evaluation.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps.

Algorithm 9 Value Iteration

initialize $v(s)$ arbitrarily for $s \in \mathcal{S}$ except that $v(terminal) = 0$;
input the threshold $\theta > 0$ determining the accuracy of the evaluation;
for true **do**:
    $\Delta \leftarrow 0$;
    for each $s \in \mathcal{S}$ **do**:
        $old \leftarrow v(s)$;
        $v(s) \leftarrow \max_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'}[R^a_s + \gamma v(s')]$;
        $\Delta \leftarrow \max(\Delta, |old - v(s)|)$;
    end for
    if $\Delta < \theta$ **then**:
        break;
    end if
end for
return $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[R^a_s + \gamma v(s')]$;

We can tell that value iteration is different from policy evaluation in how we update $v(s)$. The update rule in value iteration follows *Bellman optimality equation*.

Bellman Optimality Equation: *The optimal value functions are reached by the Bellman optimality equations*:

$$v^*(s) = \max_a R^a_s + \gamma \sum_{s' \in S} P\left(s'|s,a\right) v^*\left(s'\right).$$

The derivation of the equation is simple. According to the optimal policy, we know

$$v^*(s) = \max_a q^*(s,a). \tag{1}$$

Then from the definition of $q(s,a)$,

$$q^*(s,a) = R^a_s + \gamma \sum_{s' \in S} P\left(s'|s,a\right) v^*\left(s'\right). \tag{2}$$

Therefore we can get the Bellman optimality equation by plugging (2) into (1).

5.3.4 Comparison

The comparison of the two methods are shown as follows.

Policy Iteration: policy evaluation + policy improvement

- starts from an arbitrarily policy and then estimates the *state value* given the policy;
- generates a new policy given the estimated state value;
- converges faster in terms of the number of iterations since it doing a lot more work in each iteration.

Value Iteration: optimal value function + one policy extraction

- updates the value *greedily* (does not care the policy) at each iteration and then, *after* finding the optimal value function, determines a new policy given the optimal value function;
- convergences fast per iteration, but may be far from the true value function.

The summary for evaluation and control in MDP by dynamic programming is shown in Table 1.

Problem	Bellman Equation		Algorithm
Prediction	Expectation	$v^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v^\pi(S_{t+1})\|S_t = s]$	Policy
		$q^\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q^\pi(S_{t+1}, A_{t+1})\|S_t = t, A_t = a]$	Evaluation
Control	Expectation	$v^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v^\pi(S_{t+1})\|S_t = s]$	Policy
		$q^\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q^\pi(S_{t+1}, A_{t+1})\|S_t = t, A_t = a]$	Iteration
Control	Optimality	$v^*(s) = \max_a R_s^a + \gamma \sum_{s' \in \mathcal{S}} P(s'\|s,a) v^*(s')$	Value
		$q^*(s,a) = R_s^a + \gamma \sum_{s' \in \mathcal{S}} P(s'\|s,a) \max_{a'} q^*(s',a')$	Iteration

Table 1: Dynamic Programming algorithms for MDPs

6 Model-Free Prediction

To begin, we firstly give definition of the term *episode* then give the prediction algorithm based MC and TD methods.

Episodes: *An episode τ is a sequence of states and actions in the environment,*

$$\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T).$$

We call an episode is complete if it ends with the terminal state.

In model free method, we only utilize the episode itself without further exploitation.

6.1 Monte-Carlo Policy Evaluation

Monte-Carlo (MC) policy evaluation methods learn directly from the *complete* episodes, thus it needs no knowledge of MDP transitions or rewards. The limitation of MC policy method methods is that it requires MDPs are episodic.

Monte-Carlo Policy Evaluation: *Given some complete episodes under the policy π, we can approximate the value of s by the average returns observed after visiting to s.*

Depending on when average returns for state s in an episode, there are two different implementations.

First-Visit Monte-Carlo Policy Evaluation: *Only at the first time-step t that state s is visited in an episodes, we do the following procedure:*

- Increment counter $N(s) \leftarrow N(s) + 1$;
- Increment the total return $return(s) \leftarrow return(s) + G_t$;
- Update the value by the mean return $v(s) \leftarrow return(s)/N(s)$.

Every-Visit Monte-Carlo Policy Evaluation: *Every time-step t that state s is visited in an episode, we do the following procedure:*

- Increment counter $N(s) \leftarrow N(s) + 1$;
- Increment the total return $return(s) \leftarrow return(s) + G_t$;
- Update the value by the mean return $v(s) \leftarrow return(s)/N(s)$.

Algorithm 10 First-Visit Monte-Carlo Policy Evaluation

initialize $v(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
initialize $return(s) \leftarrow$ an empty list for all $s \in \mathcal{S}$;
input the policy π to be evaluated;
for true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
    Generate a complete episode $\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    $G \leftarrow 0$;
    for $t = T-1, T-2, ...0$ **do**:
        $G \leftarrow \gamma G + r_{t+1}$;
        if s_t appears in $(s_0, s_1, ..., s_{t-1})$ **then**:
            Append G to $return(s_t)$;
            $v(s_t) \leftarrow$average($return(s_t)$);
        end if
    end for
end for

Algorithm 11 Every-Visit Monte-Carlo Policy Evaluation

input the policy π to be evaluated;
initialize $v(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
initialize $return(s) \leftarrow$ an empty list for all $s \in \mathcal{S}$;
for true **do**:
    Generate a complete episode $\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    for $t = T-1, T-2, ..., 0$ **do**:
        $G \leftarrow \gamma G + r_{t+1}$;
        Append G to $return(s_t)$;
        $v(s_t) \leftarrow$average($return(s_t)$);
    end for
end for

By *law of large numbers*, both of two methods can achieve $v(s) \to v^\pi(s)$ as $N(s) \to \infty$. In practice, we can use a trick *incremental mean* to simplify the calculation.

Differences between DP and MC for policy evaluation:

- DP computes v_k by bootstrapping the rest of the expected return calculated with v_{k-1};
- DP iterates on Bellman expectation backup:

$$v_k(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_{k-1}(s') \right).$$

- MC updates the empirical mean return with a sampled episode:

$$v(s_t) \leftarrow v(s_t) + \alpha(G_{k,t} - v(s_t)).$$

Advantages of MC over DP:

- MC can work when the environment is unknown;
- Working with sampled episodes has a huge advantage. Even with the complete knowledge of the environment's dynamics, the complexity still could be a challenge;
- Cost of estimating a single state's value is independent of the total number of states. So one can sample episodes starting from the states of interest then average returns.

6.2 Temporal-Difference Learning

Unlike MC methods, temporal-difference (TD) does not require the episodes are complete. TD methods can learn from incomplete episodes by bootstrapping.

Temporal-Difference Learning: *Given some incomplete episodes under the policy π, we update value $v(s_t)$ toward estimated return $r_{t+1} + \gamma v(s_{t+1})$:*

$$v(s_t) \leftarrow v(s_t) + \alpha(r_{t+1} + \gamma v(s_{t+1}) - v(s_t)),$$

where $r_{t+1} + \gamma v(s_{t+1})$ is called the TD target, α is the step-size, and we call

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$$

the TD error.

Algorithm 12 TD(0) Evaluation

initialize $v(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$ except that $v(terminal) = 0$;
input the policy π to be evaluated; the step size $\alpha \in (0, 1]$;
for true **do**:
    Generate initial state s;
    while s is not terminal **do**:
        $a \leftarrow \pi(s)$;
        $r, s' \leftarrow environment(s, a)$;
        $v(s) \leftarrow v(s) + \alpha(r + \gamma v(s') - v(s))$;
        $s \leftarrow s'$;
    end while
end for

Differences between MC and TD for policy evaluation:

- TD can learn online after every step;
- MC must wait until end of episode before return is known;
- TD can learn from incomplete sequences;
- MC can only learn from complete sequences;
- TD works in continuing (non-terminating) environments;
- MC only works in episodic (terminating) environments;
- TD exploits Markov property, and it is efficient in Markov environments;
- MC does not exploit Markov property, thus it is relatively effective in non-Markov environments.

Bootstrapping: involves old values, it is something like in-place update.

Sampling: samples to get an expectation.

A summary of *bootstrapping* and *sampling* for DP, MC, and TD is shown in Table 2.

Algorithm	Bootstraps	Sampling
Dynamic Programming	$\surd$	
Monte-Carlo		$\surd$
Temporal-Difference	$\surd$	$\surd$

Table 2: *Bootstrapping* and *sampling* for DP, MC, and TD.

n-step TD methods: *Unlike the simplest TD method, in n-step TD methods, the updating rule of value $v(s_t)$ is*

$$v(s_t) \leftarrow v(s_t) + \alpha(G_t^{(n)} - v(S_t)),$$

where $G_t^{(n)}$ is the n-step return

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^n v(s_{t+n}).$$

Notice that with additional definition, we can generalize TD to MC when $n \to \infty$.

TD(λ) methods: *To make use of the information from all time-steps, we can use weight $(1-\lambda)\lambda^{n-1}$ to average n-step returns over different n as*

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)},$$

thus the updating rule for the value becomes

$$v(s_t) \leftarrow v(s_t) + \alpha(G_t^{\lambda} - v(s_t)).$$

7 Model-Free Control

In the last section, we introduce Monte-Carlo (MC) and Temporal Difference (TD) methods to evaluate the value function of a policy. With the idea of generalized policy iteration (GPI) we mentioned in section 5.3.2, we now consider how the value function can be used in control, that is, to find the optimal policy.

For an MDP with large state space, the way we sample episodes matters a lot. We define *behavior policy* that determines which action to take and *target policy* that determines the best policy we have so far, then based on the two concepts, we have two learning form:

- **On-policy** learning: the target policy and the behavior policy are the same, which means the action we will take follows the optimal policy we find;
- **Off-policy** learning: the target policy and the behavior policy are different, which means the action we will take is independent to the optimal policy we find.

On-policy may not ensure the enough exploration of the state space as when we update the policy greedily we may discard those states with great potential. Compared with on-policy learning, off-policy learning is more powerful and general, though it is often of greater variance and is slower to converge.

7.1 On Policy Monte-Carlo Control

Like MC evaluation we mentioned in section 6.1, MC control also requires fully episodes to improve the policy. It is natural to alternate between evaluation and improvement on an episode-by-episode basis. The following algorithms are based on the implementation of MC, and the differences lie in how they generate samples.

Exploring Starts: *To obtain diverse episodes, a naive ideal is initializing each episode randomly:*

Random starts usually are not common in reality. In the iteration part, we can leverage $\varepsilon-$*Greedy Exploration* to avoid the unlikely assumption of exploring starts.

$\varepsilon-$**Greedy Exploration**: *For a state with m actions, there is non-zero probability to try them:*

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon, & a = \arg\max_{a' \in \mathcal{A}} q(s, a') \\ \frac{\varepsilon}{m}, & \text{otherwise} \end{cases}$$

which means we will choose the greedy action with probability $1 - \varepsilon$ and choose an action at random with probability ε.

Policy improvement theorem: *For any ε-greedy policy π, the ε-greedy policy π' with respect to q^π is an improvement, i.e., $v'_\pi(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$.*

Algorithm 13 Monte-Carlo Control

initialize $\pi(s) \in \mathcal{A}(s)$(arbitrarily), for all $s \in \mathcal{S}$; initialize $return(s) \leftarrow$ an empty list for all $s \in \mathcal{S}$;
for true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
    Generate a complete episode $\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    $G \leftarrow 0$;
    for $t = T-1, T-2, ...0$ **do**:
        $G \leftarrow \gamma G + r_{t+1}$;
        if s_t, a_t appears in $(s_0, a_0, s_1, a_1, ..., s_{t-1}, a_{t-1})$ **then**:
            Append G to $return(s_t, a_t)$;
            $q(s_t, a_t) \leftarrow$average($return(s_t, a_t)$);
            $\pi(s_t) \leftarrow \arg\max_{a'} q(s_t, a')$;
        end if
    end for
end for

Proof:

$$
\begin{aligned}
q^{\pi}\left(s, \pi'(s)\right) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q^{\pi}(s,a) \\
&= \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}} q^{\pi}(s,a) + (1-\varepsilon) \max_{a} q^{\pi}(s,a) \\
&\geq \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}} q^{\pi}(s,a) + (1-\varepsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\varepsilon}{m}}{1-\varepsilon} q^{\pi}(s,a) \qquad (1) \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) q^{\pi}(s,a) \\
&= v^{\pi}(s).
\end{aligned}
$$

□

(The above proof is given in the Chapter 5 of the book [1]. After expanding the sum, however, it can be shown the inequality should be equality. Hmm...)

For ε-greedy, when we explore, we choose actions at random without regard to their estimated values, which may waste a chance on the action that we can sure it has a bad performance. To focus more on those states with higher value, we can refer to *Boltzmann exploration*.

Boltzmann Exploration: *Boltzmann exploration also known as Gibbs sampling and soft-max, it chooses an action based on its estimated value:*

$$\pi(a|s) = \frac{e^{\beta \hat{q}(s,a)}}{\sum_{a'} e^{\beta \hat{q}(s,a')}},$$

where $\hat{q}(s,a)$ is the estimation of the value of being in state s and taking action a, β is a tunable parameter.

7.2 On Policy Temporal-Difference Control: Sarsa

As we mentioned before, Temporal-Difference(TD) learning has several advantages over Monte-Carlo (MC) such as lower variance, online, incomplete sequences. The difference between TD control and TD learning is similar to that between MC control and MC learning. What we need to do is applying TD to $Q(S, A)$ and updating the policy every time-step.

Sarsa: *The name Sarsa is inspired by the exploitation on the quintuple $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$. Consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs:*

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)].$$

Algorithm 14 TD Sarsa

initialize $\pi(s) \in \mathcal{A}(s)$(arbitrarily), for all $s \in \mathcal{S}$; initialize $q(s, a) \in \mathcal{R}$(arbitrarily) for all possible pairs in $\mathcal{S} \times \mathcal{A}$; initialize $q(terminal, \cdot) = 0$;
for true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
    Generate a start state s;
    Choose action $a \leftarrow \pi(s)$;
    while s is not terminal **do**:
        $r, s' \leftarrow environment(s, a)$;
        $a' \leftarrow \pi(s')$
        $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma q(s', a') - q(s, a)]$;
        Update π according to $q(s, a)$;
        $s \leftarrow s'$;
        $a \leftarrow a'$;
    end while
end for

There are also n-step Sarsa and Sarsa(λ) which can be derived from TD methods.

n-**step Sarsa**: *Define the n-step Q-return as*

$$q_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n q(s_{t+n}, a_{t+n}),$$

then n-step Sarsa updates $q(s, a)$ towards the n-step Q-return

$$q(s_t, a_t) \leftarrow q(a_t, t_t) + \alpha(q_t^{(n)} - q(s_t, a_t)).$$

Like what we mentioned in n-step TD learning, when $n \to \infty$, Sarsa$\to$MC.

7.3 Off-Policy Temporal-Difference Control: Q-Learning

The development of Q-learning is a big breakout in the early days of Reinforcement Learning.

For the target policy, Q-learning updates it in a *greedy* way:

$$\pi(s_{t+1}) = \arg\max_{a'} q(s_{t+1}, a').$$

For the behavior policy, it will be updated in an ε-*greedy* way on $q(s, a)$.

The state-action value will be updated as

$$q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma \max_{a'} q(s', a') - q(s, a)].$$

Algorithm 15 Q-Learning

1: initialize $\pi(s) \in \mathcal{A}(s)$(arbitrarily), for all $s \in \mathcal{S}$; initialize $q(s, a) \in \mathcal{R}$(arbitrarily) for all possible pairs in $\mathcal{S} \times \mathcal{A}$; initialize $q(terminal, \cdot) = 0$;
2: **for** true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
3: Generate a start state s;
4: **while** s is not terminal **do**:
5: Choose action $a \leftarrow \pi(s)$;
6: $r, s' \leftarrow environment(s, a)$;
7: $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma \max_{a'} q(s', a') - q(s, a)]$;
8: Update π according to $q(s, a)$;
9: $s \leftarrow s'$;
10: **end while**
11: **end for**

One should notice that different with TD Sarsa in line 8, the action Q-learning uses to update $q(s, a)$ in line 7 is not the same as the one it truly takes.

8 Value Function Approximation

For some problems, the state spaces may huge or even infinity, *e.g.*:

- Backgammon: 10^{20} states;
- Chess: 10^{47} states;
- Came of Go: 10^{170} states;
- Robot Arm and Helicopter: continuous state space;

How can we scale up the model-free methods for these problems?

A solution for these is *function approximation*, which is an instance of *supervised learning*. Concretely, it follows that:

- $\hat{v}(s;\boldsymbol{w}) \approx v^\pi(s)$,
- $\hat{q}(s,a;\boldsymbol{w}) \approx q^\pi(s,a)$,
- $\hat{\pi}(s,a;\boldsymbol{w}) \approx \pi(a|s)$,

where $\boldsymbol{w}$ is the parameter to be learned. With such approximation, we can generalize from seen states to unseen states. Obviously, we have many available function approximator:

- Linear combinations of features,
- Neural networks,
- Decision trees,
- Nearest neighbors.

Among those, we will focus on the first two as they are differentiable and thus easy to be optimized. It is worth to notice that though such generalization makes the learning potentially more powerful (it may even applicable to partially observable problems), it is potentially more difficult to manage and understand.

8.1 Semi-gradient Method

Now we consider a naive case where we have an oracle for knowing the true value $v^\pi(s)$ for any given state s under policy π. Then the object is to find the estimator $\hat{v}(s,\boldsymbol{w})$ approximating $v^\pi(s)$. We can define the loss function by the mean squared error:

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(v^\pi(s) - \hat{v}(s,\boldsymbol{w}))^2],$$

and the gradient descend for the loss function is:

$$\Delta\boldsymbol{w} = -\frac{1}{2}\alpha\nabla_{\boldsymbol{w}}J(\boldsymbol{w}),$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \Delta\boldsymbol{w}.$$

Follow the gradient descend, we can find a local minimum. Further, if the value function is represented by a linear combination of features, then such method can converge to the global optimum.

8.1.1 Semi-gradient Method for Prediction

However, in practice, no access to the oracle of the true value $v^\pi(s)$. What we have is only the reward, or, the bootstrapping estimation. Thus we can substitute the target for $v^\pi(s)$:

- For MC, the target is the actual return G_t and hence

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{v}(s_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w});$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w})$ and hence

$$\Delta \boldsymbol{w} = \alpha(r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}) - \hat{v}(s_t, \boldsymbol{w}))\nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w})$$

For TD(0), the gradient is also called as semi-gradient, as the target $r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w})$ depends on the current value of the weight vector $\boldsymbol{w}$ and we just ignore such effect.

Algorithm 16 Gradient Monte Carlo Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
variants for this alg. can be start with $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$ randomly, $\varepsilon-$greedy, *etc.*
    Generate a complete episode $\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    for $t = 0, 1, ..., T-1$ **do**:
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G_t - \hat{v}(s_t, \boldsymbol{w})]\nabla \hat{v}(s_t, \boldsymbol{w})$;
    end for
end for

Algorithm 17 Semi-gradient TD(0) Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a \leftarrow \pi(s)$;
        $r, s' \leftarrow environment(s, a)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla \hat{v}(s, \boldsymbol{w})$;
        $s \leftarrow s'$;
    end while
end for

8.1.2 Semi-gradient Method for Control

The extension of the semi-gradient prediction methods to state-action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q^\pi$, that is represented as a parameterized functional

form with weight vector $\boldsymbol{w}$. The control version is derived from the *generalized policy iteration*. It quite like the way we make prediction:

- For MC, the target is return G_t

$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{q}(s_t, a_t, \boldsymbol{w}))\nabla \hat{q}(s_t, a_t, \boldsymbol{w});$$

- For Sarsa, the target is TD target $r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \boldsymbol{w})$:

$$\Delta \mathbf{w} = \alpha \left(r_{t+1} + \gamma \hat{q}\left(s_{t+1}, a_{t+1}, \mathbf{w}\right) - \hat{q}\left(s_t, a_t, \mathbf{w}\right)\right) \nabla_{\mathbf{w}} \hat{q}\left(s_t, a_t, \mathbf{w}\right);$$

- For Q-learning, the target is TD target $r_{t+1} + \gamma \max_a \hat{q}\left(s_{t+1}, a, \mathbf{w}\right)$:

$$\Delta \mathbf{w} = \alpha \left(r_{t+1} + \gamma \max_a \hat{q}\left(s_{t+1}, a, \mathbf{w}\right) - \hat{q}\left(s_t, a_t, \mathbf{w}\right)\right) \nabla_{\mathbf{w}} \hat{q}\left(s_t, a_t, \mathbf{w}\right).$$

One can get the corresponding control algorithm by modifying the update rule in the prediction version and then extracting the optimal policy according to $\hat{q}$, which is $\pi(s) = \arg\max_a \hat{q}(s, a, \boldsymbol{w})$.

Though TD control is powerful and practical in most case, there is no guarantees in the convergence of TD. Firstly, TD with VFA follows the *semi-gradient* rather than the true gradient of any objective function. Secondly, the updates involve doing an approximate Bellman backup (model-free) followed by fitting the underlying value function (approximation). Further, for off-policy, behavior policy and target policy are not identical, thus value function approximation may diverge.

The Deadly Triad for *Danger of Instability and Divergence*:

- Function approximation: A scalable way of generalizing from a state space much larger than the memory and computational resources;
- Bootstrapping: Update target that includes existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods);
- Off-policy learning: training on a distribution of transitions other than that produced by the target policy.

8.2 Deep Q-Learning

Deep Q-learning leverages nonlinear function approximator, deep neural networks, to avoid manual designing of features. Such method is also called DQN, and it is a well-known case of *deep reinforcement learning*.

Deep Reinforcement Learning:

- Frontier in machine learning and artificial intelligence;
- Deep neural networks can be used to represent value function, policy function, and even model;
- Optimize loss function by stochastic gradient descent.

Algorithm 18 Semi-gradient TD(0) Policy Evaluation

initialize the differentiable function $\hat{v}(s, \boldsymbol{w}) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$;
input the policy π to be evaluated; the step size α;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a \leftarrow \pi(s)$;
        $r, s' \leftarrow environment(s, a)$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla\hat{v}(s, \boldsymbol{w})$;
        $s \leftarrow s'$;
    end while
end for

The success of DQN lies in two innovative techniques that greatly improve and stabilize the training procedure.

Experience Replay

- To reduce the correlations among samples, DQN stores transition (s_t, a_t, r_t, s_{t+1}) in a replay memory $\mathcal{D}$;
- To perform experience replay, DQN repeats the following
  - samples an experience tuple from the dataset:

$$(s, a, r, s') \sim \mathcal{D};$$

  - computes the target value with the sampled tuple: $r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w})$;
  - uses stochastic gradient descent to update the network weights

$$\Delta\boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}) - q(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{q}(s, a, \boldsymbol{w}).$$

Fixed Target

- To help improve stability, DQN fixes the weights used in the target calculation for multiple updates;
- Let a different set of parameter $\boldsymbol{w}^-$ be the set of weights used in the target, and $\boldsymbol{w}$ be the weights to be updated in each step;
- To perform experience replay with fixed target, DQN repeats the following
  - samples an experience tuple from the dataset: $(s, a, r, s') \sim \mathcal{D}$;
  - computes the target value for the sampled tuple:

$$r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}^-);$$

  - uses stochastic gradient descent to update the network weights:

$$\Delta\boldsymbol{w} = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \boldsymbol{w}^-) - q(s, a, \boldsymbol{w}))\nabla_{\boldsymbol{w}}\hat{q}(s, a, \boldsymbol{w});$$

– updates $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$ periodically.

Algorithm 19 Deep-Q Network

initialize the replay memory $\mathcal{D}$ with capacity N; initialize the state-action value function q with random weight $\boldsymbol{w}$; initialize the target state-action value function $\hat{q}$ with random weight $\boldsymbol{w}^- = \boldsymbol{w}$;
input the period C;
for true **do**:
    Generate a start state s;
    while s is not terminal **do**:
        Choose action $a = \begin{cases} \arg\max_{a' \in \mathcal{A}} q(s, a'), & \text{with probability } \varepsilon; \\ \text{a random action}, & \text{otherwise}; \end{cases}$
        $r, s' \leftarrow environment(s, a)$;
        Store transition (s, a, r, s') in $\mathcal{D}$;
        Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$;
        Set $y_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is the terminal state}; \\ r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \boldsymbol{w}^-), & \text{otherwise}; \end{cases}$
        Perform a gradient descent step on $(y_j - q(s_j, a_j, \boldsymbol{w}))^2$ with respect to the parameter $\boldsymbol{w}$;
        Reset $\hat{q} = q$ every C steps;
    end while
end for

9 Policy Optimization

For value-based reinforcement learning, deterministic policy is generated directly from the value function using greedy $a = \arg\max_{a'} q(a', s)$. Now instead we can parameterize the policy function as $\pi_\theta(a|s)$ where θ is the learnable policy parameter and the output is a probability over the action set.

Value-based *v.s.* Policy-based

- Value-based methods: solve RL problems through dynamic programming
  - related to classic RL and control theory;
  - learn value function;
  - generate an implicit policy based on the value function;
  - learn a deterministic policy based on the estimated action values;
  - developed by Richard Sutton, David Silver, DeepMind;
- Policy-based methods: solve RL problems mainly through learning
  - related to machine learning and deep learning;
  - do not require value function for action selection;
  - learn a stochastic policy;
  - developed by Pieter Abbeel, Sergey Levine, OpenAI, Berkeley;

The two methods can also be combined together. A popular algorithm called *Actor-Critic* entails learning both policy and value function.

Pros and cons of Policy-based

- Advantages:
  - can converge on a local optimum (worst case) or global optimum (best case);
  - is effective in high-dimensional action space;
- Disadvantages:
  - typically converges to a local optimum;
  - the policy is of high variance;

9.1 Policy Optimization Theorem

Objective of Optimization Policy: *Given a policy approximator $\pi(s, a)$ with parameter θ, find the $\theta^\star$ that gives us the optimal policy.*

One thing we care is how do we measure the quality of a policy π_θ? Let τ be a trajectory sampled from the policy function π_θ, then we defined the value of policy π_θ as

$$J(\theta) = \mathbb{E}_\tau \left[\sum_t r(s_t, a_t^\tau) \right].$$

Thus we have the goal of policy-based methods as

$$\theta^* = \arg\max_\theta \mathbb{E}_\tau \left[\sum_t r(s_t, a_t^\tau) \right].$$

However, such $J(\theta)$ may not available or handy. Hence a trick is using approximation. For example,

- In the episodic environment with discrete space, we can use the value of the starting state s_0:

$$J(\theta) \approx v^{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}[v(s_0)],$$

- In the environment with continuous space, we can use the average reward:

$$J(\theta) \approx \sum_s d^{\pi_\theta}(s) v^{\pi_\theta}(s) = \sum_s d^{\pi_\theta} \sum_{a \in \mathcal{A}} \pi_\theta(a|s) q^{\pi_\theta}(s, a),$$

  where d^{π_θ} is the stationary distribution of Markov chain for π_θ.

Depends on the form of $J(\theta)$, we have different methods to maximize it

- If $J(\theta)$ is differentiable, we can use gradient-based methods:
  - Gradient Ascend;
  - Conjugate Gradient;
  - Quasi-newton.
- If $J(\theta)$ is non-differentiable or hard to compute the derivative, we can use some derivative-free black-box optimization methods:
  - Cross-entropy Method (CEM);
  - Hill Climbing;
  - Evolution Algorithm;
  - Approximate Gradients by Finite Difference.

In this note, we mainly focus on gradient-based methods.

Policy Gradient Theorem: *For a policy π_θ with parameter θ, we have the gradient as*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s;\theta) q^\pi(s, a)].$$

The proof is available in the Section 12.1 of *Reinforcement Learning: An Introduction* [1]. A detailed proof is also given in the blog [10]. The following *proof* is just for personal interest.

Proof:

For simplicity, we leave it implicit in all cases that π is a function of θ and all gradients are also implicitly

with respect to θ. Notice that the gradient of the state-value function:

$$\begin{aligned}
\nabla v^\pi(s) &= \nabla \left(\sum_a \pi(a|s) q^\pi(s,a) \right) \\
&= \sum_a \left(\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s) \nabla q^\pi(s,a) \right) && \text{(Product rule of calculus)} \\
&= \sum_a \left(\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s) \nabla \left(R_s^a + \sum_{s'} \mathcal{P}_{ss'}^a v^\pi(s') \right) \right) && \text{(Sec 5.2 MDP)} \\
&= \sum_a \left(\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s) \nabla \sum_{s'} \mathcal{P}_{ss'}^a v^\pi(s') \right) && (R_s^a \text{ is irrevalent to } \theta) \\
&= \sum_a \left(\nabla \pi(a|s) q^\pi(s,a) + \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a \nabla v^\pi(s') \right),
\end{aligned}$$

which gives us a nice recursive form of the gradient. Therefore the future state value function $v^\pi(s')$ can be repeated unrolled by following the same equation.

Before unrolling, we define the probability of transitioning from state s to state s' in k steps under policy π as $\rho^\pi(s \to s', k)$. Thus it follows that

- when $k = 0$: obviously, we have $\rho^\pi(s \to s, k = 0) = 1$;
- when $k = 1$, it is easy to get the probability from the transition matrix as $\rho^\pi(s \to s', k = 1) = \sum_a \pi(a|s) \mathcal{P}_{ss'}^a$;
- for other cases such as going from state s to s' in $k+1$ steps, we can consider a middle state x where it takes k steps from s to x, thus we have a recursively expression as $\rho^\pi(s \to s', k+1) = \sum_x \rho^\pi(s \to x, k) \rho^\pi(s \to s', 1)$.

We now consider unrolling the recursive representation of $\nabla v^\pi(s)$. For simplicity, let $\phi(s) = \sum_a \nabla \pi(a|s) q^\pi(s,a)$. Then it follows that

$$\begin{aligned}
\nabla v^\pi(s) &= \phi(s) + \sum_a \pi(a|s) \sum_{s'} \mathcal{P}_{ss'}^a \nabla v^\pi(s') \\
&= \phi(s) + \sum_{s'} \sum_a \pi(a|s) \mathcal{P}_{ss'}^a \nabla v^\pi(s') && (\pi(a|s) \text{ is irrevalent to } s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \nabla v^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \left(\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', 1) \nabla v^\pi(s'') \right) \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \nabla v^\pi(s'') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1) \phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \phi(s'') + \sum_{s'''} \rho^\pi(s \to s''', 3) \nabla v^\pi(s''') \\
&= \dots \\
&= \sum_{s' \in \mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to s', k) \phi(s'),
\end{aligned}$$

which finally arrives at

$$\nabla v^{\pi}(s) = \sum_{s} \sum_{k=0}^{\infty} \rho^{\pi}(s \to s', k) \sum_{a} \nabla \pi(a|s') q^{\pi}(s', a).$$

Thus for the $J(\theta)$ of the episodic environment, we have

$$\begin{aligned}
\nabla J(\theta) &= \nabla v^{\pi}(s_0) \\
&= \sum_{s} \sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k) \sum_{a} \nabla \pi(a|s) q^{\pi}(s, a) \\
&= \sum_{s} \eta(s) \sum_{a} \nabla \pi(a|s) q^{\pi}(s, a) \qquad (\eta(s) = \textstyle\sum_{k=0}^{\infty} \rho^{\pi}(s_0 \to s, k)) \\
&= \left(\sum_{s} \eta(s) \right) \sum_{s} \frac{\eta(s)}{\sum_{s} \eta(s)} \sum_{a} \nabla \pi(a|s) q^{\pi}(s, a) \\
&\propto \sum_{s} d^{\pi}(s) \sum_{a} \nabla \pi(a|s) q^{\pi}(s, a)
\end{aligned}$$

as $\sum_s \eta(s)$ is a constant and $d^{\pi}(s)$ is exactly the stationary distribution. Further, such deriving also shows the connection between the two different $J(\theta)$ we defined above. Now we rewrite the gradient as

$$\begin{aligned}
\nabla J(\theta) &\propto \sum_{s} d^{\pi}(s) \sum_{a} \nabla \pi(a|s) q^{\pi}(s, a) \\
&= \sum_{s} d^{\pi}(s) \sum_{a} q^{\pi}(s, a) \pi(a, s) \frac{\nabla \pi(a|s)}{\pi(a|s)} \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s; \theta) q^{\pi}(s, a)].
\end{aligned}$$

□

For other general forms of policy gradient methods, one can refer to the paper [11] and the note [12] (again, many thanks to lilianweng's blog [10]).

9.2 REINFORCE: Monte-Carlo Policy Gradient

We now consider the policy gradient for the case where we can get complete episodes. According to the *policy gradient theorem*, it follows that

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s; \theta) q^{\pi}(s, a)] \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s; \theta) G_t]
\end{aligned}$$

as $q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a]$. Thus it is similar to *Monte-Carlo policy evaluation*, we can measure G_t from real complete sample episodes and use that to update our policy gradient.

Monte-Carlo Policy Gradient: *Starting from the state s_0 sampled from a distribution $d(s)$, we denote one episode as*

$$\tau = (s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T) \sim (\pi_\theta, \mathcal{P}^{a}_{s_t s_{t+1}}).$$

Then we update the parameter in the rule:

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi(a|s;\theta).$$

Algorithm 20 REINFORCE: Monte-Carlo Policy-Gradient Control

initialize policy parameter $\theta \in \mathbb{R}^d$;
input a differentiable policy parameterization $\pi(a|s;\theta)$; the step size α;
for true **do**:
    Generate an episode $(s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T)$;
    for each step of the episode $t = 0, 1, ..., T-1$ **do**:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$;
        $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \ln \pi(a_t|s_t;\theta)$;
    end for
end for

The analysis for REINFORCE requires us to consider the episode level. We define the sum of rewards over the trajectory τ (for G_t baseline, the case is quite similar) as

$$R(\tau) = \sum_{t=1}^{T} r_t.$$

Let $\mathcal{D}(\tau;\theta) = d(s_0)\prod_{t=0}^{T-1} \pi(a_t|s_t;\theta)\mathcal{P}^{a_t}_{s_t s_{t+1}}$ denote the probability over trajectories when executing the policy π_θ. Then the policy gradient of $J(\theta)$ is equivalent to

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} r_t\right] \\
&= \sum_{\tau} \nabla_\theta \mathcal{D}(\tau;\theta) R(\tau) \\
&= \sum_{\tau} \mathcal{D}(\tau;\theta) R(\tau) \frac{\nabla_\theta \mathcal{D}(\tau;\theta)}{\mathcal{D}(\tau;\theta)} \\
&= \sum_{\tau} \mathcal{D}(\tau;\theta) R(\tau) \nabla_\theta \ln \mathcal{D}(\tau;\theta) \\
&\approx \frac{1}{m}\sum_{i=1}^{m} R(\tau_i) \nabla_\theta \ln \mathcal{D}(\tau;\theta),
\end{aligned}$$

where we suppose there are m episodes and refer to MC thought to approx the expectation. Such approximation is reasonable as long as $m \to \infty$.

We now show that such method does not need the dynamics of the model. Considering the term that

matters in the gradient we got above, it follows that

$$\begin{aligned}\nabla_\theta \ln \mathcal{D}(\tau;\theta) &= \nabla_\theta \ln \left[\mu(s_0)\prod_{t=0}^{T-1}\pi(a_t|s_t;\theta)\,\mathcal{P}^{a_t}_{s_t s_{t+1}}\right] \\ &= \nabla_\theta \left[\ln \mu(s_0) + \sum_{t=0}^{T-1}\ln \pi(a_t|s_t;\theta) + \ln \mathcal{P}^{a_t}_{s_t s_{t+1}}\right] \\ &= \sum_{t=0}^{T-1}\nabla_\theta \ln \pi(a_t|s_t;\theta)\,,\end{aligned}$$

and thus

$$\nabla_\theta J(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} R(\tau_i)\left(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi(a_t^i|s_t^i;\theta)\right).$$

It shows that the dynamics, *i.e.* the transition matrix, of the model is not needed, which means policy gradient is a model-free method.

9.3 Actor-Critic Policy Gradient

Recall that in section 8 we use $\hat{v}(s;\boldsymbol{w}) \approx v^\pi(s)$. Now we combine the value approximation with policy approximation, then we will get the Actor-Critic method.

Actor: *Actor is actually a policy parameterization $\pi(a|s;\theta)$ to generate actions; it updates parameter θ in direction suggested by critic.*

Critic: *Critic is actually a value approximation $\hat{v}(s;\boldsymbol{w})$ to evaluate the reward of a state under current actor (policy); it needs to update parameter $\boldsymbol{w}$ to make accurate evaluation.*

Algorithm 21 Actor-Critic

initialize policy parameter θ; initialize the state-value function parameter $\boldsymbol{w}$;
input a differentiable policy parameterization $\pi(a|s;\theta)$; a differentiable state-value function parameterization $\hat{v}(s;\boldsymbol{w})$, the step size $\alpha_{\boldsymbol{w}}, \alpha_\theta$;
for true **do**:
    Generate a start state s;
    $I \leftarrow 1$
    while s is not terminal **do**:
        Choose action $a \leftarrow \pi(a|s;\theta)$;
        $r, s' \leftarrow environment(s,a)$;
        $\delta \leftarrow r + \gamma\hat{v}(s';\boldsymbol{w}) - \hat{v}(s;\boldsymbol{w})$;
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}}\delta\nabla\hat{v}(s;\boldsymbol{w})$;
        $\theta \leftarrow \theta + \alpha_\theta I\delta\nabla \ln \pi(a|s;\theta)$;
        $I \leftarrow \gamma I$;
        $s \leftarrow s'$;
    end while
end for

As we can see, the critic is solving a familiar problem *policy evaluation*, while the actor is doing *policy improvement*.

9.4 Extension of Policy Gradient

Nowadays, State-of-the-art RL methods are almost all policy-based.

A2C, A3C: Asynchronous Methods for Deep Reinforcement Learning, ICML’16. Representative high-performance actor-critic algorithm.

TRPO: Trust region policy optimization: deep RL with natural policy gradient and adaptive step size.

PPO: Proximal policy optimization algorithms: deep RL with importance sampled policy gradient.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] J. K. Blitzstein and J. Hwang, *Introduction to Probability*. Chapman and Hall/CRC, 2014.

[3] C. Robert and G. Casella, *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[4] Z. Shao, "Si252 reinforcement learning," https://piazza.com/class/k5ug5osvzhp3z8.

[5] A. Slivkins *et al.*, "Introduction to multi-armed bandits," *Foundations and Trends® in Machine Learning*, vol. 12, no. 1-2, pp. 1–286, 2019.

[6] T. Lattimore and C. Szepesvári, "Bandit algorithms," *preprint*, p. 28, 2018.

[7] L. Weng, "The Multi-Armed Bandit Problem and Its Solutions," https://lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html.

[8] B. Zhou, "Intro to reinforcement learning," https://github.com/zhoubolei/introRL.

[9] D. Silver, "Reinforcement learning," https://www.davidsilver.uk/teaching/.

[10] L. Weng, "Policy Gradient Algorithms," https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html.

[11] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2015.

[12] D. Seita, "Notes on the Generalized Advantage Estimation Paper," https://danieltakeshi.github.io/2017/04/02/notes-on-the-generalized-advantage-estimation-paper/.