Reinforcement Learning

Summary
Fabian Damken
August 1, 2022

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this course we will look at lots of methods from the domain of *reinforcement learning (RL)*. RL is an approach for agent-oriented learning where the agent learns by repeatedly acting with the environment and from rewards. Also, it does not know how the world works in advance. RL is therefore close to how humans learn and tries to tackle the fundamental challenge of artificial intelligence (AI):

“The fundamental challenge in artificial intelligence and machine learning is learning to make good decisions under uncertainty.” (Emma Brunskill)

RL is so general that every AI problem can be phrased in its framework of learning by interacting. However, the typical setting is that at every time step, an agent perceives the state of the environment and chooses an action based on these perceptions. Subsequently, the agent gets a numerical reward and tries to maximize this reward by finding a suitable strategy. This procedure is illustrated in Figure 1.1.

1.1 Artificial Intelligence

The core question of AI is how to build “intelligent” machines, requiring that the machine is able to adapt to its environment and handle unstructured and unseen environments. Classically, AI was an “engine” producing answers to various queries based on rules designed by a human expert in the field. In (supervised) machine learning (ML), the rules are instead learned from a (big) data set and the “engine” produces answers based on the data. However, this approach (leaning from labeled data) is not sufficient for RL as demonstrations might be imperfect, the correspondence problem, and that we cannot demonstrate everything. We can break these issues down as follows: supervised learning does not allow “interventions” (trial-and-error) and evaluative feedback (reward).

The core idea leading to RL was to not program machines to simulate an adult brain, but to simulate a child’s brain that is still learning. RL formalizes this idea of intelligence to interpret rich sensory input and choosing complex actions. We know that this may be possible as us humans do it all the time. This lead to the RL view on AI depicted in Figure 1.1 and is based on the hypothesis that learning from a scalar reward is sufficient to yield intelligent behavior (Sutton and Barto, 2018).

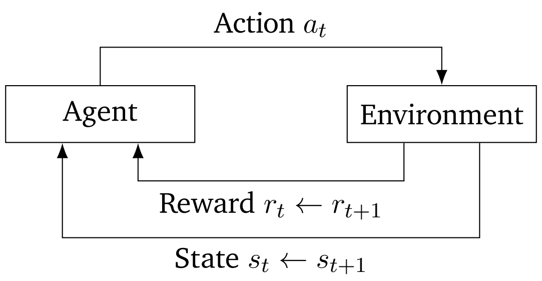

Action a_t
Agent
Environment
Reward $r_t \leftarrow r_{t+1}$
State $s_t \leftarrow s_{t+1}$

Figure 1.1: The Reinforcement Learning Cycle

	actions *do not* change the state of the world	actions change the state of the world
no model	(Multi-Armed) Bandits	Reinforcement Learning
known model	Decision Theory	Optimal Control, Planning

Table 1.1: Problem Classification

1.2 Reinforcement Learning Formulation

RL tries to *maximize the long-term reward* by finding a strategy/policy with the general assumption that it is easier to assess a behavior by specifying a cost than specifying the behavior directly. In general, we have the following things different to most (un)supervised settings:

- no supervision, but only a reward signal
- feedback (reward) is always delayed and not instantaneous
- time matters, the data is sequential and by no means i.i.d.
- the agent's actions influence the subsequent data, i.e., the agent generates its own data

In addition to this, RL is challenged by a numerous complicated factors and issues, e.g., dynamic state-dependent environments, stochastic and unknown dynamics and rewards, exploration vs. exploitation, delayed rewards (how to assign a temporal credit), and very complicated systems (large state spaces with unstructured dynamics). For designing an RL-application, we usually have to choose the state representation, decide how much prior knowledge we want to put into the agent, choose an algorithm for learning, design an objective function, and finally decide how we evaluate the resulting agent. By all these decisions, we want to reach a variety of goals, e.g., convergence, consistency, good generalization abilities, high learning speed (performance), safety, and stability. However, we are usually pretty restricted in terms of computation time, available data, restrictions in the way we act (e.g., safety constraints), and online vs. offline learning.

This sounds like a lot and, in fact, is! We therefore often limit ourselves onto specific (probably simpler) sub-problems and solve them efficiently under some assumptions. Some common flavors of the RL problem are, for instance:

- *Full:* no additional assumptions, the agent can only probe the environment through the state dynamics and its actions; the agent has to understand the environment
- *Filtered State and Sufficient Statistics:* assumption of a local Markov property (i.e., the next state only depends on the current state and action, and not on the past), decomposable rewards (into specific time steps); we can show that every problem is a (probably infinite) instance of this assumption, but how to filter the state to get such properties?
- *Markovian Observable State:* assume that we can observe the state fulfilling the Markov property directly
- *Further Simplifications:* contextual bandits (the dynamics do not depend on the action or the past and current state at all); bandits (only a single state)

We can summarize the different RL-like problems in a matrix, see Table 1.1.

1.2.1 Components

To solve an RL problem, we need three ingredients:

1. Model Learning
    - we want to approximate and learn the state transfer using methods from supervised learning
    - need to generate actions for model identification
    - estimation of the model or the model's parameters
2. Optimal Control/Planning
    - generation of optimal control inputs
3. Performance Evaluation

1.3 Wrap-Up

- why RL is crucial for AI and why all other approaches are ultimately doomed
- background and characteristics of RL
- classification of RL problems
- core components of RL algorithms
- Additional reading material:
    - Book: "Introduction to Reinforcement Learning" (Sutton and Barto, 2018), Chapter 1

2 Preliminaries

In this chapter we cover some preliminaries that are necessary for understanding the rest of the course. Note that most of this content is dense and should be used as a reference throughout this course as oppose to an actual introduction to the topic.

2.1 Functional Analysis

Definition 1 (Normed Vector Space)**.** A *normed vector space* is a vector space $\mathcal{X}$ over X equipped with a *norm* $\|\cdot\| : \mathcal{X} \to \mathbb{R}$ that has the following properties:

1. $\|x\| \geq 0$ for all $x \in \mathcal{X}$ and $\|x\| = 0$ iff $x = 0$ (non-negativity)
2. $\|\alpha x\| = |\alpha| \|x\|$ for all $\alpha \in X$ and $x \in \mathcal{X}$ (homogenity)
3. $\|x_1 + x_2\| \leq \|x_1\| + \|x_2\|$ for all $x_1, x_2 \in \mathcal{X}$ (triangle inequality)

For the rest of this course we usually use real finite-dimensional vectors spaces $\mathcal{X} = \mathbb{R}^d$, $d \in \mathbb{N}^+$, the L_∞-norm $\|\cdot\|_\infty$, and (weighted) L_2-norms $\|\cdot\|_{2,\rho}$.

Definition 2 (Complete Vector Space)**.** A vector space $\mathcal{X}$ is *complete* if every Cauchy sequence[1] in $\mathcal{X}$ has a limit in $\mathcal{X}$.

Definition 3 (Contraction Mapping)**.** Let $\mathcal{X}$ be a vector space equipped with a norm $\|\cdot\|$. An operator $T : \mathcal{X} \to \mathcal{X}$ is called an *α-contraction mapping* if $\exists \alpha \in [0, 1) : \forall x_1, x_2 \in \mathcal{X} : \|Tx_1 - Tx_2\| \leq \alpha \|x_1 - x_2\|$. If only $\exists \alpha \in [0, 1] : \forall x_1, x_2 \in \mathcal{X} : \|Tx_1 - Tx_2\| \leq \alpha \|x_1 - x_2\|$, T is called *non-expanding*.

Definition 4 (Lipschitz Continuity)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be vector spaces equipped with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$, respectively. A function $f : \mathcal{X} \to \mathcal{Y}$ is called *Lipschitz-continuous* if $\exists L \geq 0 : \forall x_1, x_2 \in \mathcal{Y} : \|f(x_1) - f(x_2)\|_Y \leq L \|x_1 - x_2\|_X$.

Remark 1. *Obviously, every contraction mapping is also Lipschitz-continuous with Lipschitz-constant $L \triangleq \alpha$ and is therefore continuous. Also, the product of two Lipschitz-continuous mappings is Lipschitz-continuous and therefore $T^n = T \circ \cdots \circ T$ is Lipschitz-continuous, too.*

Definition 5 (Fixed Point)**.** Let $\mathcal{X}$ be a vector space equipped and let $T : \mathcal{X} \to \mathcal{X}$ be an operator. Then $x \in \mathcal{X}$ is a *fixed point* of T if $Tx = x$.

Theorem 1 (Banach Fixed Point Theorem)**.** *Let $\mathcal{X}$ be a complete vector space with a norm $\|\cdot\|$ and let $T : \mathcal{X} \to \mathcal{X}$ be an α-contraction mapping. Then T has a unique fixed point $x^* \in \mathcal{X}$ and for all $x_0 \in \mathcal{X}$ the sequence $x_{n+1} = Tx_n$ converges to x^* geometrically, i.e., $\|x_n - x^*\| \leq \alpha^n \|x_0 - x^*\|$.*

[1]This section is already overflowing with mathematical rigor compared to the rest of the course, so we will skip the definition of a Cauchy sequence here.

2.2 Statistics

This section introduces some concepts of statistics, but you should

2.2.1 Monte-Carlo Estimation

Let X be a random variable with mean $\mu = \mathbb{E}[X]$ and variance $\sigma^2 = \mathrm{Var}[X]$ and let $\{x_i\}_{i=1}^n$ be i.i.d. realizations of X. We then have the *empirical mean* $\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^n x_i$ and we can show that $\mathbb{E}[\hat{\mu}_n] = \mu$ and $\mathrm{Var}[\hat{\mu}_n] = \sigma^2/n$. Also, if the sample size n goes to infinity, we have the *strong* and *weak law of large numbers,* respectively:

$$P\big(\lim_{n\to\infty} \hat{\mu}_n = \mu\big) = 1 \qquad\qquad \lim_{n\to\infty} P\big(|\hat{\mu}_n - \mu| > \epsilon\big) = 0$$

Also, we have the *central limit theorem:* no matter the distribution of P, its mean value converges to a normal distribution, $\sqrt{n}(\hat{\mu}_n - \mu) \overset{D}{\to} \mathcal{N}(0, \sigma^2)$.

2.2.2 Bias-Variance Trade-Off

When evaluating/training a ML model, the error is due to two factors (illustrated in Figure 2.1):

- *bias,* i.e., the distance to the expected prediction
- *variance,* i.e., the variability of a prediction for a given data point

In general, we want to minimize both, but we can only minimize one of them! This is known as the *bias-variance trade-off.*

2.2.3 Important Sampling

If we want to estimate the expectation of some function $f(x)$ for $x \sim p(x)$, but cannot sample from $p(x)$ (which is often the case for complicated models), we can instead use the following relation(s):

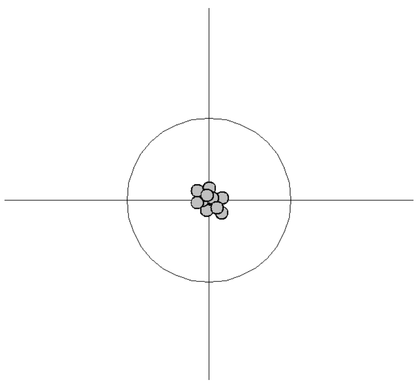
$$\mathbb{E}_{x\sim p}\big[f(x)\big] = \sum_x f(x)p(x) = \sum_x f(x)\frac{p(x)}{q(x)}q(x) = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$$
$$\mathbb{E}_{x\sim p}\big[f(x)\big] = \int f(x)p(x)\,\mathrm{d}x = \int f(x)\frac{p(x)}{q(x)}p(x)\,\mathrm{d}x = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$$

and sample from a surrogate distribution $q(x)$. This approach obviously has problems if q does not cover p sufficiently well along with other problems. See Bishop, 2006, Chapter 11 for details.

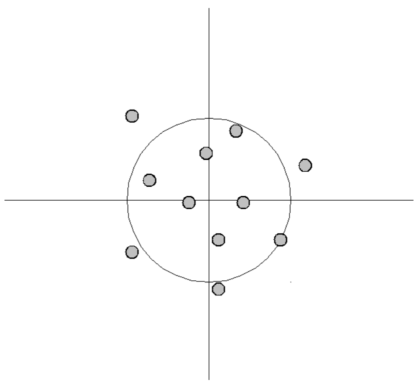
2.2.4 Linear Function Approximation

A basic approximator we will need often is the linear function approximator $f(\boldsymbol{x}) = \boldsymbol{w}^\top\boldsymbol{\phi}(\boldsymbol{x})$ with weights $\boldsymbol{w}$ and features $\boldsymbol{\phi}(\boldsymbol{x})$. As the weights are optimized and the features are designed, we have lots of variability here. Actually, constructing useful features is the influential step on the approximation quality. Most importantly, features are the only point where we can introduce interactions between different dimensions. A good representations therefore captures all dimensions and all (possibly complex) interaction.

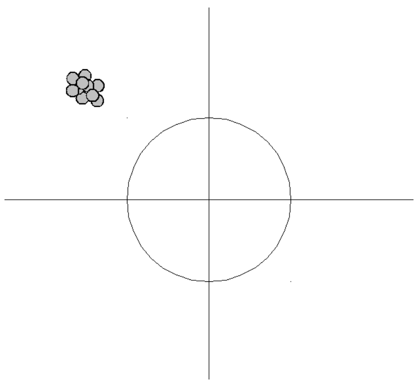
We will now go over some frequently used features.



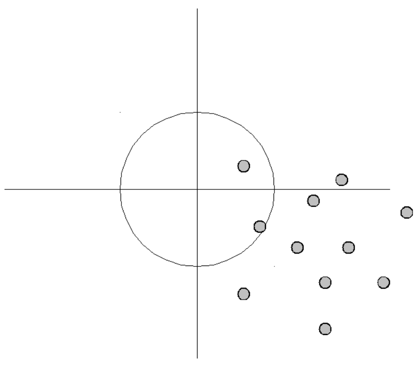
(a) Low Bias, Low Variance



(b) Low Bias, High Variance

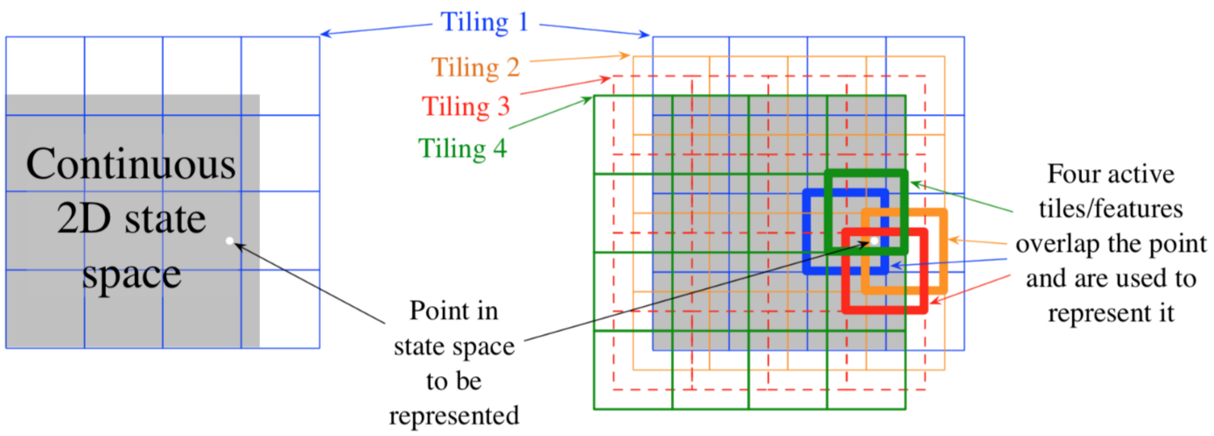


(c) High Bias, Low Variance



(d) High Bias, High Variance

Figure 2.1: Bias-Variance Trade-Off; Source: Bernhard Thiery (CC BY-SA 3.0)



Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 2.2: Tile Coding; Source: `https://towardsdatascience.com/reinforcement-learning-tile-coding-implementation-7974b600762b`

Polynomial Features *Polynomial features* are particularly simple and capture the interaction between dimensions by multiplication. For instance, the first- and second-order polynomial features of a two-dimensional state $\boldsymbol{x} = (x_1, x_2)^\top$ are:

$$\boldsymbol{\phi}_{P1}(\boldsymbol{x}) = (1, x_1, x_2, x_1x_2)^\top \qquad \boldsymbol{\phi}_{P2}(\boldsymbol{x}) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1x_2^2, x_1^2x_2, x_1^2, x_2^2)$$

However, the number of features grows *exponentially* with the dimension!

Fourier Basis Fourier series can be used to approximate periodic functions by adding sine and cosine waves with different frequencies and amplitudes. Similarly, we can use them for general function approximation of functions with bounded domain. As it is possible to approximate any even function with just cosine waves and we are only interested in bounded domains, we can set this domain to positive numbers only and can therefore approximate any function. For one dimension, the n-th order *Fourier (cosine) basis* is

$$\phi_m(x) = \cos(\pi m \tilde{x}), \quad m = 0, 1, \dots, n.$$

and $\tilde{x}$ is a normalized version of x, i.e., $\tilde{x} = (x - x_{\max})/(x_{\max} - x_{\min})$.

Coarse Coding *Coarse coding* divides the space into M different regions and produced M-dimensional coding features for which the j-th entry is 1 iff the data point lies withing the respective region; all values the data point does not lie in are 0. Features with this codomain are also called *sparse*.

Tile Coding *Tile coding* is a computationally efficient form of coarse coding which use square *tilings* of space. It uses N tilings, each composed of M tiles. The features "vector" is then an $N \times M$ matrix where a single value is 1 iff x lies inside the tile and 0 otherwise. Figure 2.2 shows an illustration of this coding.

Radial Basis Functions *Radial basis functions (RBFs)* are a generalization of coarse coding where the features are in the interval $(0, 1]$. A typical RBF is the Gaussian

$$\phi_j(\boldsymbol{x}) = \exp\left\{ -\frac{\|\boldsymbol{x} - \boldsymbol{c}_j\|_2^2}{2\sigma_j^2} \right\}$$

with center $\boldsymbol{c}_j$ and bandwidth σ_j^2.

Neural Networks A very powerful alternative to hand-crafting features are *neural networks (NNs)*. By stacking multiple layers of learned features, they are very powerful prediction machines.

2.2.5 Likelihood-Ratio Trick

Suppose we need to differentiate the expectation of some function $f(x)$ w.r.t. θ where $x \sim p_\theta(\cdot)$. However, we cannot directly calculate $\mathbb{E}_{x\sim p_\theta}[f(x)]$ or "differentiate through sampling." Instead, we can use the identity

$$\frac{\mathrm{d}}{\mathrm{d}z}\log h(z) = \frac{h'(z)}{h(z)} \qquad \implies \qquad f'(z) = h(z)\,\frac{\mathrm{d}}{\mathrm{d}z}\log h(z)$$

to reformulate the derivative of the expectation as

$$\frac{\partial}{\partial\theta}\mathbb{E}_{x\sim p_\theta}[f(x)] = \int f(x)\frac{\partial}{\partial\theta}p_\theta(x)\,\mathrm{d}x = \int f(x)\left(\frac{\partial}{\partial\theta}p_\theta(x)\right)p_\theta(x)\,\mathrm{d}x = \mathbb{E}_{x\sim p_\theta}\left[f(x)\,\frac{\partial}{\partial\theta}p_\theta(x)\right].$$

While this is a very powerful approach, the gradient estimator exhibits high variance!

2.2.6 Reparametrization Trick

Suppose we need to differentiate the expectation of some function $f(x)$ w.r.t. θ where $x \sim p_\theta(\cdot)$. However, we cannot directly calculate $\mathbb{E}_{x\sim p_\theta}[f(x)]$ or "differentiate through sampling." Instead, we reformulate the expectation with a function $x = g_\theta(\varepsilon)$ that separates the random components ε from the deterministic ones θ such that we can reparameterize the expectation as

$$\mathbb{E}_{x\sim p_\theta}[f(x)] = \mathbb{E}_\varepsilon\big[f\big(g_\theta(\varepsilon)\big)\big].$$

For instance, if $p_\theta(x) = \mathcal{N}(\mu_\theta, \sigma_\theta^2)$ is a Gaussian, $g_\theta(\varepsilon) = \mu_\theta + \sigma_\theta\varepsilon$ with $\varepsilon \sim \mathcal{N}(0,1)$. We can now simply use the chain rule to take the derivative w.r.t. θ. Compared to the likelihood-ratio trick, this estimator has less variance!

2.3 Miscellaneous

Finally, this section contains all the stuff that does not fit into the categories before.

2.3.1 Useful Integrals

The following hold for a distribution $p_\theta(x)$:

$$\int \frac{\partial}{\partial\theta}p_\theta(x)\,\mathrm{d}x = 0 \qquad\qquad \int \frac{\partial}{\partial\theta}\log p_\theta(x)\,\mathrm{d}x = \int \frac{\frac{\partial}{\partial\theta}p_\theta(x)}{p_\theta(x)}\,\mathrm{d}x = 0 \tag{2.1}$$

The first identity can be shown by swapping the integral and derivative and using the normalization condition of probability densities. For the second we use integration by parts with $f' = \frac{\partial}{\partial\theta}p_\theta(x)$, for which $f = 0$ due to the first integral. Hence, the second follows.

3 Markov Decision Processes and Policies

In this chapter we will develop the groundwork for all upcoming chapters and define some important mathematical concepts.

3.1 Markov Decision Processes

A *Markov decision process (MDP)* describes the environment for RL *formally* for the case where we can fully observe the environment, i.e., we directly "see" the state. Also, the current state fully characterized the system and future states are independent from the past *(Markov property)*. This mathematical framework allows precision and rigorous reasoning on, for instance, optimal solutions and convergence (note, however, that we will only touch the tip of the iceberg in theoretical analysis and we will be less rigorous than some mathematician may wish). The nice this of MDPs is there wide applicability: we can frame almost all RL problems as MDPs. Most of the remaining chapter here focuses on fully observable and finite MDPs, i.e., the number of states and actions is finite. Table 3.1 shows an overview over different Markov models.

We now went over some mathematical definitions for building up the "Markovian framework."

Definition 6 (Markov Property)**.** A stochastic process X_t is *Markovian* or *fulfills the Markov property* if $P_t(S_{t+1} = s' \mid S_t = s, S_{t-1} = k_{t-1}, \dots, S_0 = k_0) = P_t(S_{t+1} = s' \mid S_t = s)$ for all t.

Definition 7 (Stationary Transition Probabilities)**.** If $P_t(S_{t+1} = s' \mid S_t = s)$ is time invariant, $p_{ss'} \coloneqq P_t(S_{t+1} = s' \mid S_t = s)$ are the *stationary transition probabilities.*

Definition 8 (State Transition Matrix)**.** With the transition probabilities $p_{ss'}$, let $\boldsymbol{P}_{ss'} \coloneqq p_{ss'}$ for all s, s' be the *transition matrix.*

Definition 9 (Markov Chain)**.** A *Markov chain* is a tuple $\langle \mathcal{S}, \boldsymbol{P}, \iota \rangle$ with the (finite) set of discrete-time states $S_t \in \mathcal{S}$, $n \coloneqq |\mathcal{S}|$, transition matrix $\boldsymbol{P} \in [0,1]^{n \times n}$, and the initial state distribution $\iota_i = P(S_0 = i)$.

Definition 10 (Probability Row Vector)**.** The vector $\boldsymbol{p}_t \coloneqq \sum_{i=1}^{n} P(S_t = i)\boldsymbol{e}_i^\top$ with the i-th unit vector $\boldsymbol{e}_i$ and includes the probability of being in the i-th state at time step t.

Theorem 2 (Chapman-Kolmogorov for Finite Markov Chains)**.** *The probability row vector $\boldsymbol{p}_{t+k}$ at time step $t+k$ starting from $\boldsymbol{p}_t$ at time step t is given by $\boldsymbol{p}_{t+k} = \boldsymbol{p}_t \boldsymbol{P}^k$.*

	All states observable?	
Actions?	Yes	No
Yes	Markov Decision Process	Partially Observable MDP
No	Markov Chain	Hidden Markov Model

Table 3.1: Types of Markov Models

Proof. Assume w.l.o.g. $t = 0$ We proof this by induction. For the base case, let $k = 1$. Let $\boldsymbol{p}_0 = (p_{0,1}, p_{0,2}, \dots, p_{0,n})$ be an arbitrary probability row vector. By linearity, we have

$$\boldsymbol{p}_0 \boldsymbol{P}_{ss'} = \sum_{i=1}^{n} p_{0,1} \boldsymbol{e}_i^\top \boldsymbol{P} = \sum_{i=1}^{n} p_{0,1} \boldsymbol{P}_i$$

where $\boldsymbol{P}_i$ is the i-th row of $\boldsymbol{P}$. Rewriting this equation in terms of explicit transition probabilities, we have

$$\begin{aligned} &= \sum_{i=1}^{n} P(S_0 = i) \sum_{j=1}^{n} \boldsymbol{e}_j^\top P(S_1 = j \mid S_0 = i) = \sum_{j=1}^{n} \boldsymbol{e}_j^\top \sum_{i=1}^{n} P(S_0 = i) P(S_1 = j \mid S_0 = i) \\ &= \sum_{j=1}^{n} \boldsymbol{e}_j^\top \sum_{i=1}^{n} P(S_1 = j, S_0 = i) = \sum_{j=1}^{n} \boldsymbol{e}_j^\top P(S_1 = j) = \sum_{j=1}^{n} p_{1,j} \boldsymbol{e}_j^\top = \boldsymbol{p}_1. \end{aligned}$$

The first equality is due to the definition of $\boldsymbol{P}_i$, the third is due to the definition of conditional probabilities, the fourth is due to marginalizing out S_0, and the final is just another application of the definit ion of the probability row vector. For the induction step $k \to k+1$, assume that $\boldsymbol{p}_k = \boldsymbol{p}_t \boldsymbol{P}^k$ holds for some k. We then have $\boldsymbol{p}_{k+1} = \boldsymbol{p}_k \boldsymbol{P} = \boldsymbol{p}_0 \boldsymbol{P}^k \boldsymbol{P} = \boldsymbol{p}_0 \boldsymbol{P}^{k+1}$ where the first equality is due to the base case and the second is due to the induction hypothesis. □

Definition 11 (Steady State)**.** A probability row vector $\boldsymbol{p}$ is called a *steady* state if an application of the transition matrix does not change it, i.e., $\boldsymbol{p} = \boldsymbol{p}\boldsymbol{P}$.

Remark 2. *While the steady state is in general not independent of the initial state (consider, for instance, $\boldsymbol{P} = \boldsymbol{I}$), it gives insights in which states of the Markov chain are visited in the long run.*

Definition 12 (Absorbing, Ergodic, and Regular Markov Processes)**.** A Markov process is called …

- …*absorbing* if it has at least one *absorbing state* (i.e., a state that can never be left) and if that state can be reached from every other state (not necessarily in one step).
- …*ergodic* if all states are *recurrent* (i.e., visited an infinite number of times) and *aperiodic* (i.e., visited without a systematic period).
- …*regular* if some power of the transition matrix has only positive (non-zero) elements.

3.1.1 Example

3.2 Markov Reward Processes

Definition 13. Markov Reward Process A *Markov reward process* is a tuple $\langle \mathcal{S}, \boldsymbol{P}, R, \gamma, \iota \rangle$ with the (finite) set of discrete-time states $S_t \in \mathcal{S}$, $n := |\mathcal{S}|$, transition matrix $\boldsymbol{P}_{ss'} = P(s' \mid s)$, reward function $R : \mathcal{S} \to \mathbb{R} : s \mapsto R(s)$, discount factor $\gamma \in [0, 1]$, and the initial state distribution $\iota_i = P(S_0 = i)$. We call $r_t = R(s_t)$ the immediate reward at time step t.

3.2.1 Time Horizon, Return, and Discount

Note that in 13 we did not clearly specify how the reward is computed. Especially we did not define how much time steps the reward "looks" into the future. For this we generally have three options: finite, indefinite, and infinite. The first computes the reward for a fixed and finite number of steps, the second until some stopping criteria is met, and the third infinitely.

Definition 14 (Cumulative Reward)**.** The *cumulative reward* summarizes the reward signals of a Markov reward process (MRP). We define the following:

$$J_t^{\text{total}} := \sum_{k=1}^{T} r_{t+k} \qquad J_t^{\text{average}} := \frac{1}{T}\sum_{k=1}^{T} r_{t+k} \qquad J_t \equiv J_t^{\text{discounted}} := \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k,$$

For an infinite horizon, we take the limit of these as $T \to \infty$.

Theorem 3. *The cumulative discounted reward fulfills the recursive relation* $J_t = r_{t+1} + \gamma J_{t+1}$.

Proof. $J_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k = r_{t+1} + \sum_{k=t+2}^{T} \gamma^{k-t-1} r_k = r_{t+1} + \gamma \sum_{k=t+2}^{T} \gamma^{k-t-2} r_k = r_{t+1} + \gamma J_{t+1}$ □

Definition 15 (Return)**.** The *return* $J(\tau)$ of a trajectory $\tau = (s_t)_{t=1}^{T}$ is the discounted reward $J(\tau) := J_0(\tau)$.

Remark 3. *The infinite horizon discounted cumulative reward for* $r_t = 1$ *(for all t) is a geometric series and we have* $J_t = \lim_{T\to\infty} \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k = \sum_{k=0}^{\infty} \gamma^k = 1/(1-\gamma)$ *for* $\gamma < 1$. *If the reward is lower/upper bounded by* $r_{\min}/r_{\max}$, *we have* $J_t \in \left[r_{\min}/(1-\gamma), r_{\max}/(1-\gamma)\right]$. *Similarly, the return is lower/upper-bounded.*

We can interpret the discount factor γ as a "measure" how important future rewards are to the current state (how delayed vs. immediate the reward is). For instance, $\gamma \approx 0$ yields myopic evaluation and $\gamma \approx 1$ yields far-sighted evaluation. An alternative interpretation is that the discount factor is the probability that the process continues (such that the discounted return is the expected return w.r.t. the discount factor). Despite the obvious advantage that including a discount factor prevents the return from diverging, we also have a couple of other reasons why it makes sense to weigh future rewards less:

- we might be *uncertain* about the future (e.g., with imperfect) models
- if the reward is *financial,* immediate rewards earn more interest than delayed rewards
- *animal and human behavior* also shows preference for immediate rewards—and why try to mimic biology in the end

However, sometimes we still use *undiscounted* MRPs (i.e., $\gamma = 1$), for instance if all sequences are guaranteed to terminate.

3.2.2 Value Function

Definition 16 (Value Function for MRP)**.** The *state value function* for a MRP is $V(s) := \mathbb{E}_{\boldsymbol{P}}[J_t \mid s_t = s]$ for any t.That is, the *expected* return starting from state s where the expectation is w.r.t. the state dynamics.

Theorem 4 (Bellman Equation)**.** *For all states* $s \in \mathcal{S}$, *we have* $V(s) = R(s) + \gamma \mathbb{E}\left[V(s_{t+1}) \,\middle|\, s_t = s\right]$.

Proof. $V(s) = \mathbb{E}\left[J_t \,\middle|\, s_t = s\right] = R(s) + \gamma \mathbb{E}\left[J_{t+1} \,\middle|\, s_t = s\right] = R(s) + \gamma \mathbb{E}\left[V(s_{t+1}) \,\middle|\, s_t = s\right]$ □

The Bellman equation allows us to decompose the value of any state into its immediate reward and the value of the subsequent states (in expectation). As we only consider discrete MRPs, we can also express the Bellman equation in matrix form,

$$\boldsymbol{V} = \boldsymbol{R} + \gamma \boldsymbol{P}\boldsymbol{V} \qquad \Longleftrightarrow \qquad \boldsymbol{V} = (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1}\boldsymbol{R}, \tag{3.1}$$

where $\boldsymbol{V}$ and $\boldsymbol{R}$ are columns vectors with the values and rewards, respectively, and $\boldsymbol{P}$ is the transition matrix. We can therefore directly solve this linear equation and get the values of the states! However, for n states the complexity is $\mathcal{O}(n^3)$ and hence this is only possible for small MRPs. For large MRPs, a variety of efficient iterative methods exist. In the following chapters, we will cover *dynamic programming* (chapter 4) *Monte-Carlo evaluation* (chapter 5) and *temporal difference learning* (chapter 6).

3.2.3 Example

3.3 Markov Decision Processes

So far, we only considered processes *without* actions, i.e., we were not able to interact with the process. The next natural extension is from MRPs to MDPs:

Definition 17 (Markov Decision Process)**.** A *Markov decision process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, R, \gamma, \iota \rangle$ with the (finite) set of discrete-time states $S_t \in \mathcal{S}$, $n \coloneqq |\mathcal{S}|$, (finite) set of actions $A_t \in \mathcal{A}$, $m \coloneqq |\mathcal{A}|$, transition matrix $\boldsymbol{P}_{ss'}^{a} = P(s' \,|\, s, a)$, reward function $R : \mathcal{S} \times \mathcal{A} : (s, a) \mapsto R(s, a)$, discount factor $\gamma \in [0, 1]$, and the initial state distribution $\iota_i = P(S_0 = i)$. We call $r_t = R(s_t)$ the immediate reward at time step t.

An interesting—yet philosophical—question is, whether a scalar reward is adequate to formulate a goal? The big hypothesis underlying its usage is the *Sutton hypothesis* that wall we mean by goals can be formulated as the maximization of a sum of immediate rewards. While this hypothesis might be wrong, it turns out to be so simple and flexible that we just use it. Also, it forces us to simplify our goal and to actually formulate *what* we want instead of *why*. Hence, the goal must be outside of the agent's direct control, i.e., it must not be a component of the agent. However, the agent must be able to measure successes explicitly and frequently.

In order to reason about an agent and what it might do, we first have to introduce *policies*.

3.3.1 Policies

A *policy* defines, at any point in time, what action an agent takes, i.e., it fully defines the *behavior* if the agent. Policies are very flexible and can be Markovian or history-dependent, deterministic or stochastic, stationary or non-stationary, etc.

Definition 18 (Policy)**.** A *policy* π is a distribution over actions given the state s, i.e., $\pi(a \,|\, s) = P(a \,|\, s)$.

Note that we can reduce a deterministic policy $a = \pi(s)$ to a stochastic one using $\pi(a \,|\, s) = \mathbb{1}\big[a = \pi(s)\big]$ for discrete and $\pi(a \,|\, s) = \delta\big(a - \pi(s)\big)$ for continuous action spaces. Given an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, R, \gamma, \iota \rangle$ and a policy π, let $\mathcal{M}^\pi = \langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}^\pi, R^\pi, \gamma, \iota \rangle$ be the policy π's MRP with

$$\boldsymbol{P}_{ss'}^{\pi} = \mathbb{E}_{a \sim \pi(\cdot \,|\, s)}\big[\boldsymbol{P}_{ss'}^{a}\big] \qquad\qquad R^\pi(s) = \mathbb{E}_{a \sim \pi(\cdot \,|\, s)}\big[R(s, a)\big]. \tag{3.2}$$

This allows us to apply theory of MRPs and Markov chains to MDPs. However, it is often useful to exploit the action distribution instead of reducing it to the state dynamics.

Value Functions

Like for MRPs, we define the value function for MDPs:

Definition 19 (Value Function for MDP)**.** The *state value function* of a MDP is $V^\pi(s) \coloneqq \mathbb{E}_{\boldsymbol{P},\pi}[J_t \,|\, s_t = s]$ for any t. That is, the *expected* return starting from state s where the expectation is w.r.t. the state dynamics and policy.

However, as we seek to maximize the return and therefore want to steer towards the largest point of the value function, it is helpful to also define the *action* value function:

Definition 20 (Action Value Function)**.** The *action value function* of a MDP is $Q^\pi(s, a) \coloneqq \mathbb{E}_{\boldsymbol{P},\pi}[J_t \,|\, s_t = s, a_t = a]$ for any t. That is, the *expected* return starting from state s, taking action a in the first step and subsequently following policy π where the expectation is w.r.t. the state dynamics and policy.

Hence, if we know the action value function for some policy π, we can easily choose the action that steers the system to the largest return achievable following π by locally maximizing $Q(s,a)$ over a for a given state s: $\pi(s) = \arg\max_a Q(s,a)$.

Similar to MRPs, we can also decompose the state and action value function according to a Bellman equation.

Theorem 5 (Bellman Expectation Equation)**.** *For all states $s \in \mathcal{S}$, we have the following decompositions:*

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{a,s'}\left[R(s,a) + \gamma V^\pi(s') \,\middle|\, s\right] = \mathbb{E}_a\left[Q^\pi(s,a)\right] \\ Q^\pi(s,a) &= R(s,a) + \gamma \mathbb{E}_{s',a'}\left[Q^\pi(s',a') \,\middle|\, s,a\right] = R(s,a) + \gamma \mathbb{E}_{s'}\left[V^\pi(s') \,\middle|\, s,a\right] \end{aligned} \tag{3.3}$$

Note that the Q-function decomposition is very similar to the MRP-decomposition of the state value function.

Proof. Left as an exercise for the reader (hint: plug in the definitions of the individual components). □

Due to the reformulation (3.2), we can reformulate the Bellman equation analogous to (3.1) as $\boldsymbol{V}^\pi = \boldsymbol{R}^\pi + \gamma \boldsymbol{P}^\pi \boldsymbol{V}^\pi$ which we can solve in closed form. However, also analogous to the MRP-case, this is inefficient for high-dimensional state spaces.

Definition 21 (Bellman Operator)**.** The *Bellman operator* for V and Q is an operator T^π mapping from state and action value functions to state and action value functions. It is defined as follows:

$$\begin{aligned} (T^\pi V)(s) &= \mathbb{E}_{a,s'}\left[R(s,a) + \gamma V(s') \,\middle|\, s\right] \\ (T^\pi Q)(s,a) &= R(s,a) + \gamma \mathbb{E}_{s',a'}\left[Q(s',a') \,\middle|\, s,a\right] \end{aligned}$$

Theorem 6. *The Bellman operator is an α-contraction mapping w.r.t. $\|\cdot\|_\infty$ if $\gamma \in (0,1)$.*

Proof. □

Remark 4. *With these operators, we can compactly write the Bellman equation(s) as $T^\pi V^\pi = V^\pi$ and $T^\pi Q^\pi = Q^\pi$ and the policy's state and action value functions are the* unique *respective fixed points of T^π. With Theorem 6, repeated application of T^π to any vector converges towards this fixed point.*

Optimality

Definition 22 (Optimality)**.** The optimal state/action-value function is the maximum value over all policies:

$$V^*(s) := \max_\pi V^\pi(s) \qquad\qquad Q^*(s,a) := \max_\pi Q^\pi(s,a).$$

The optimal value function then specifies the *best* possible performance in the MDP and we call an MDP *solved* when we know the optimal value function.

Definition 23 (Policy Ordering)**.** For two policies π, π', we write $\pi \geq \pi'$ iff $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$.

Theorem 7. *For any Markov decision process there exists a optimal policy π^* with $\forall \pi : \pi^* \geq \pi$ and all policies achieve the unique optimal state- and action-value functions (22). There exists a deterministic optimal policy.*

Remark 5. *We can recover the optimal deterministic policy by maximizing $Q^*(s,a)$ over a:*

$$\pi^*(s \,|\, a) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

With these definitions at hand, we can take a look at Bellman's principle of optimality:

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." (Richard Bellman, 1957)

This principle is formalized in the *Bellman optimality equation.*

Theorem 8 (Bellman Optimality Equation). *For all states $s \in \mathcal{S}$, we have the following decompositions:*

$$\begin{aligned} V^*(s) &= \max_{a\in\mathcal{A}} Q^*(s,a) = \max_{a\in\mathcal{A}} \Big\{ R(s,a) + \gamma \mathbb{E}_{s'}\big[V^*(s') \,\big|\, s\big] \Big\} \\ Q^*(s,a) &= R(s,a) + \gamma \mathbb{E}_{s'}\big[V^*(s') \,\big|\, s\big] = R(s,a) + \gamma \mathbb{E}_{s'}\big[\max_{a'\in\mathcal{A}} Q^*(s',a') \,\big|\, s\big] \end{aligned} \tag{3.4}$$

Proof. Left as an exercise for the reader (hint: plug in the definitions of the individual components). □

Definition 24 (Bellman Optimality Operator). The *Bellman optimality operator* for V and Q is an operator T^* mapping from state- and action-value functions to state- and action-value functions. It is defined as follows:

$$\begin{aligned} (T^*V)(s) &= \max_{a\in\mathcal{A}} \Big\{ R(s,a) + \gamma \mathbb{E}_{s'}\big[V(s') \,\big|\, s\big] \Big\} \\ (T^*Q)(s,a) &= R(s,a) + \gamma \mathbb{E}_{s'}\big[\max_{a'\in\mathcal{A}} Q(s',a') \,\big|\, s\big] \end{aligned}$$

Theorem 9. *The Bellman optimality operator is an α-contraction mapping w.r.t. $\|\cdot\|_\infty$ if $\gamma \in (0,1)$.*

Proof. □

Remark 6. *With these operators, we can compactly write the Bellman equation(s) as $T^*V^* = V^*$ and $T^*Q^* = Q^*$ and the optimal state- and action-value functions are the* unique *fixed points of T^*. Also, repeated application of T^* to any state- or action-value function converges to the optimal state- or action-value function.*

While we had closed-form solutions for the MRP and policy value functions, it is not possible to solve the Bellman optimality equation in closed form as its a nonlinear equation system (due to the involved maximizations). In the following chapters we will look at a variety of methods for solving this problem iteratively, starting with *dynamic programming.*

3.3.2 Example

3.4 Wrap-Up

- definition of Markov reward processes and Markov decision processes
- definition of the two value functions and how to compute them
- definition of an optimal policy
- the Bellman equation
- the Bellman expectation and optimality equations
- Additional reading material:
  - Book: "Introduction to Reinforcement Learning" (Sutton and Barto, 2018), Chapter 3
  - Book: "Markov Decision Processes" (Puterman, 2005), Chapter 2

4 Dynamic Programming

In this chapter we will look into a very general approach of solving problem, *dynamic programming (DP),* and will apply it to MDPs. All these methods have in common that we have a perfect model of the world (e.g., a given MDP) and we want to find a policy that maximizes the MDP's reward. The general idea is to break the overall problem down into smaller sub-problems which we then solve optimally. By combining optimal sub-solutions, we get an optimal global solution, assuming that *Bellman's principle of optimality* applies and that the decomposition is possible. An additional assumption is that the sub-problems *overlap,* i.e., they may recur many times and we can cache and reuse their solutions.

Both of these assumptions are fulfilled by MDPs where the Bellman equation gives recursive decompositions and the value function stores and reuses solutions. For finite-horizon MDPs, DP is straightforward by starting from $k = T - 1$ and iterating backwards, $k = T - 1, T, \dots, 0$, reusing the sub-solutions. The value function and policy are then k-dependent and we get the optimal value function and policy for $k = 0$ (when the information was able to "flow" through the MDP). Compare to brute-force policy search, we get an exponential speedup! As DP for finite-horizon problems is straightforward, we will now stick to infinite-horizon problems.

Note that we have two central problems we can solve in an MDP: *prediction* and *control.* In prediction, we just want to predict the MDPs's behavior, i.e., measure its value function given a policy. In control, we want to find the optimal value function and policy. As these problems are closely related and we need the former for the latter, we will discuss them jointly.

4.1 Policy Iteration

policy iteration (PI) is an algorithm for solving infinite-horizon MDPs by repeating the following steps:

1. Policy Evaluation: estimate V^π
2. Policy Improvement: find a $\pi' \geq \pi$

The following sections describe these steps in detail and also discuss convergence.

4.1.1 Policy Evaluation

In *policy evaluation,* we want to compute the value function V^π for a given policy π (i.e., perform prediction). For this we would either directly solve the Bellman expectation equation using (3.1), but this has complexity $\mathcal{O}(n^3)$. Instead, we start with some approximation $V_{k=0}$ of the value function (which can be arbitrarily bad) and repeatedly apply the Bellman operator (21) until convergence:

$$V^{(k+1)}(s) \leftarrow (T^\pi V^{(k)})(s) = \sum_{a \in \mathcal{A}} \pi(s \mid a) \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{(k)}(s') \right).$$

The sequence $\big(V^{(k)}\big)_{k=0}^{\infty}$ converges, by Banach's fixed point theorem and Theorem 6, to the fixed point V^π. In matrix form, we can also write this update as $\boldsymbol{V}^{(k+1)} = \boldsymbol{R}^\pi + \gamma \boldsymbol{P}^\pi \boldsymbol{V}^{(k)}$.

However, in subsequent steps we need the action-value function Q^π. There are generally two options for this: either recover Q^π from V^π using (3.3) or directly estimate it by repeatedly applying its Bellman operator. As this is completely analogous, we will skip the explicit equations here.

4.1.2 Policy Improvement

In *policy improvement,* we want to use find a better policy $\pi' \geq \pi$ using Q^π. We do this by acting greedily, i.e., $\pi'(s) = \arg\max_{a\in\mathcal{A}} Q^\pi(s,a)$. With this definition, we have the following inequality:

$$Q^\pi\big(s,\pi'(s)\big) = \max_{a\in\mathcal{A}} Q^\pi(s,a) \geq Q^\pi\big(s,\pi(s)\big) = V^\pi(s).$$

Theorem 10 (Policy Improvement Theorem)**.** *Let π and π' be policies with $Q^\pi\big(s,\pi'(s)\big) \geq V^\pi(s)$. Then $\pi' \geq \pi$.*

Proof. By repeatedly applying the premise $(*)$, the Bellman expectation equation $(\dagger)$, and the definition of the value function $(\ddagger)$, we find the following inequality chain:

$$\begin{aligned} V^\pi(s) &\overset{(*)}{\leq} Q^\pi\big(s,\pi'(a)\big) \overset{(\dagger)}{=} \mathbb{E}_{\pi',\boldsymbol{P}}\big[r_{t+1} + \gamma V^\pi(s_{t+1}) \,\big|\, s_t = s\big] \overset{(*)}{\leq} \mathbb{E}_{\pi',\boldsymbol{P}}\big[r_{t+1} + \gamma Q^\pi\big(s_{t+1},\pi'(s_{t+1})\big) \,\big|\, s_t = s\big] \\ &\overset{(*\dagger)}{\leq} \mathbb{E}_{\pi',\boldsymbol{P}}\big[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^\pi\big(s_{t+2},\pi'(s_{t+2})\big) \,\big|\, s_t = s\big] \overset{(*\dagger)}{\leq} \mathbb{E}_{\pi',\boldsymbol{P}}\big[r_{t+1} + \gamma r_{t+2} + \dots \,\big|\, s_t = s\big] \overset{(\ddagger)}{=} V^{\pi'}(s) \end{aligned}$$

□

With this theorem of which the premise is fulfilled when acting greedily, we see that the policy is either improved or, if improvement stops and $V^{\pi'} = V^\pi$, we found the optimal policy through the Bellman optimality equation $Q^\pi\big(s,\pi(s)\big) = V^\pi(s)$. Hence, PI always converges to the optimal policy π^*!

4.1.3 Remarks

As both policy evaluation and improvement converge to a unique fixed point, policy iteration overall converges to the optimal policy! The algorithm is summarized in algorithm 1. Note that recovering Q^π from V^π requires a model of the MDP. While this is not a problem for policy iteration as the policy evaluation step requires a model anyway, later on we will estimate the value differently and may not have a model. If so, it makes more sense to directly estimate the action-value function.

4.1.4 Examples

4.2 Value Iteration

Value iteration (VI) follows a similar idea as PI. However, we do not explicitly compute a policy π but instead only find the optimal value function V^* from which we can recover the optimal policy (using (3.4) and greedy updates). To find the optimal value function, we repeatedly apply the Bellman optimality operator (24)

$$V^{(k+1)}(s) = \max_{a\in\mathcal{A}} \Big\{ R(s,a) + \gamma\mathbb{E}_{s'}\big[V^{(k)}(s') \,\big|\, s\big] \Big\}.$$

As this an α-contraction due to Theorem 9, multiple applications converge to the optimal state-value function V^* for an arbitrary initialization $V^{(0)}$. This procedure is summarized in algorithm 2. We also have the following theorem on the accuracy of the value function estimates.

Algorithm 1: Policy Iteration

```
   Input: Markov decision process $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, R, \gamma, \iota \rangle$
   Output: optimal $V^*$, $Q^*$, and $\pi^*$
1  initialize $\pi$ arbitrarily
2  repeat
       // Policy Evaluation
3      $k \leftarrow 0$
4      initialize $Q^{(k)}$ arbitrarily
5      repeat
6          $V^{(k+1)}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(s \mid a) R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) Q^{(k)}(s', a)$
7          $k \leftarrow k + 1$
8      until until convergence
       // Recover the action-value function.
9      $Q^{(\infty)}(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{(\infty)}(s')$
       // Policy Improvement
10     $\pi(a \mid s) \leftarrow \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^{(\infty)}(s, a) \\ 0 & \text{otherwise} \end{cases}$
11 until until convergence
12 return $V^{(\infty)}$, $Q^{(\infty)}$, $\pi$
```

Theorem 11. *Let $\|V\|_\infty = \max_{s \in \mathcal{S}} V(s)$ be the maximum-norm of V and let $\epsilon > 0$. If the following inequality holds two successive state-value functions V_{i+1} and V_i:*

$$\|V_{i+1} - V_i\|_\infty < \epsilon,$$

then the error w.r.t. the maximum norm of V_{i+1} is upper-bounded by

$$\|V_{i+1} - V^*\|_\infty < \frac{2\epsilon\gamma}{1 - \gamma}$$

where γ is the discount factor.

4.3 Remarks

In VI, we had to repeatedly apply max-operations which can be costly. In contrast, PI does not require these redundant maximizations, but needs to carry around an explicit policy. However, both algorithms are in the same complexity class: for m actions and n states, the state-value based methods is in $\mathcal{O}(mn^2)$ and the action-value based methods (where instead of recovering Q^π from V^π we estimate it directly) is in $\mathcal{O}(m^2n^2)$.

Note that this chapter only considered *synchronous* DP which is only applicable to problems with a relatively small[1] state space. *Asynchronous* DP, on the other hand, allows high parallelization and can be applied to larger problems.

[1]A few million, but for high-dimensional problems the state space increases exponentially (cure of dimensionality).

Algorithm 2: Value Iteration

```
  Input: Markov decision process $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \boldsymbol{P}, R, \gamma, \iota \rangle$
  Output: optimal $V^*$ and $\pi^*$
  // Find optimal state-value function.
1 $k \leftarrow 0$
2 initialize $V^{(k)}$ arbitrarily
3 repeat
4     $V^{(k+1)}(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V^{(k)}(s') \right\}$
5     $k \leftarrow k + 1$
6 until until convergence
  // Recover optimal policy.
7 $\pi(a \mid s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) V^{(k)}(s') \right\} \\ 0 & \text{otherwise} \end{cases}$
8 return $V^{(\infty)}$, $\pi$
```

Problem	Core Equation	Algorithm
Prediction	Bellman Exp. Eq.	Policy Evaluation
Control	Bellman Exp. Eq., Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Eq.	Value Iteration

Table 4.1: Synchronous Dynamic Programming

4.4 Wrap-Up

- dynamic programming
- computing optimal policies and value functions for environments with known dynamics
- Additional reading material:
  - Book: “Introduction to Reinforcement Learning” (Sutton and Barto, 2018), Chapter 4

5 Monte-Carlo Methods

While DP requires a model of the world (in terms of a fully specified MDP) and can perform planning inside an MDP, Monte-Carlo (MC) methods and algorithm are *model-free*. They are mostly based on PI which—when estimating the action-value functions directly—does not require a model of the world during policy improvement. To evaluate the policy, MC methods use repeated random sampling and just produce an estimate. However, they still assume finite MDPs. We can again identify two tasks:

- *Model-Free Prediction:* estimate the value function of an unknown MDP given a policy
- *Model-Free Control:* find the optimal value function and policy of an unknown MDP; achieved by combining policy improvement with MC prediction

The core idea is to use the mean return of multiple episodes as an estimation for the value of a state. Hence, MC methods can only be applied to episodic MDPs, i.e., ones which eventually terminate. Here we have two options:

- *First-Visit MC:* estimate the value of a state s by averaging the returns *only for the first time* s is visited in an episode; yields an *unbiased* estimator; see algorithm 3
- *Every-Visit MC:* estimate the value of a state s by averaging the returns *every time* s is visited in an episode; yields a *biased* but still *consistent* estimator; see algorithm 4

Algorithm 3: First-Visit Monte-Carlo Policy Evaluation

```
   Input: policy $\pi$
   Output: approximate $V^\pi$
 1 initialize $V^{(k)}$ arbitrarily
 2 initialize $Returns(s)$ as an empty list for all $s \in \mathcal{S}$
 3 repeat
 4     $(s_0; r_1, s_1; r_2, s_2; \ldots; r_{T-1}, s_{T-1}, r_T) \leftarrow$ generate episode
 5     foreach $t = 0, 1, \ldots, T$ do
 6         if $s_t$ is visited for the first time then
 7             $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$    // cumulative discounted reward
 8             append $J_t$ to $Returns(s_t)$
 9             $V(s_t) \leftarrow$ average$\big(Returns(s_t)\big)$    // update value estimate
10 until convergence
11 return $V$
```

Algorithm 4: Every-Visit Monte-Carlo Policy Evaluation

```
   Input: policy $\pi$
   Output: approximate $V^\pi$
1  initialize $V^{(k)}$ arbitrarily
2  initialize $Returns(s)$ as an empty list for all $s \in \mathcal{S}$
3  repeat
4      $(s_0; r_1, s_1; r_2, s_2; \ldots; r_{T-1}, s_{T-1}, r_T) \leftarrow$ generate episode
5      foreach $t = 0, 1, \ldots, T$ do
6          $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$    // cumulative discounted reward
7          append $J_t$ to $Returns(s_t)$
8          $V(s_t) \leftarrow$ average$\big(Returns(s_t)\big)$    // update value estimate
9  until convergence
10 return $V$
```

5.1 Example

5.2 Wrap-Up

- approximation of value functions when the dynamics are not available
- differences of DP and MC
- Additional reading material:
  - Book: “Introduction to Reinforcement Learning” (Sutton and Barto, 2018), Chapter 5
  - Book: “Monte-Carlo Simulation-Based Statistical Modeling” (Chen and Chen, 2017)

6 Temporal Difference Learning

We will now turn to methods that feel more like "reinforcement learning" rather than DP and control theory: *temporal difference (TD) learning.* Like MC methods, TD learning learns directly from experience and is also *model-free,* i.e., it has no knowledge of the MDP dynamics. However, TD methods can learn from incomplete episodes and therefore do not require episodic MDPs, making them readily applicable to all kinds of problems. The core idea is to update an estimate towards a *better estimate.* Consider *incremental* every-visit MC policy evaluation,

$$V(s_t) \leftarrow V(s_t) + \alpha\big(J_t - V(s_t)\big) = (1-\alpha)V(s_t) + \alpha J_t, \tag{6.1}$$

where J_t is the return following s_t and α is a trade-off between the two estimates. Note that for $\alpha = 1/(K+1)$ where K is the number of samples collected before this sample, this update reduces to plain every-visit MC. The idea of TD learning is to replace the return J_t which requires all future samples with an estimate of the return using the immediate reward r_{t+1} together with an estimate of the value function V_{t+1}. This process of using an estimate to update another estimate is called *bootstrapping.* This yields the simplest TD learning algorithm, TD(0), with the following update:

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big) = V(s_t) + \alpha\delta_t. \tag{6.2}$$

We call $r_{t+1} + \gamma V(s_t)$ the *TD target* and $\delta_t \coloneqq r_{t+1} + \gamma V(s_t) - V(s_t)$ the *TD error.* Again, α is a trade-off between the two estimates where $\alpha \approx 0$ just sticks to the previous estimate and $\alpha \approx 1$ directly jumps to the new (bootstrapped) estimate.

Algorithm 5: TD(0)

```
  Input: environment
  Output: value function
1 repeat
2     initialize $s$
3     while $s$ is not terminal do
4         take action $a \sim \pi(\cdot \mid s)$, observe reward $r$ and next state $s'$
5         $\delta \leftarrow r + \gamma V(s') - V(s)$
6         $V(s) \leftarrow V(s) + \alpha\delta$
7         $s \leftarrow s'$
8 until convergence
9 return $V$
```

6.1 Temporal Differences vs. Monte-Carlo (vs. Dynamic Programming)

While both TD and MC methods are model-free, they still have some major differences. Firstly, TD can learn before and even *without* knowing the final outcome of an episode. This enables online learning (i.e.,

Method	Uses Bootstrapping	Uses Sampling
Dynamic Programming	Yes	No
Monte-Carlo	No	Yes
Temporal Differences	Yes	Yes

Table 6.1: Dynamic Programming vs. Monte-Carlo vs. Temporal Difference

learning while running) whereas MC has to wait for an episode to end and the final return is known before learning anything. This directly transfers to continuing (i.e., non-terminating) environments where TD can be employed and MC cannot.

As usual, we have a bias-variance trade-off between the methods, or more precisely between the two updates (6.1) and (6.2). While the return J_t is an unbiased estimate of $V^\pi(s_t)$ and therefore the (first-visit) MC update is unbiased, the TD target has much lower variance as it only depends on a single random action-transition-reward triple. However, the TD target is biased unless $V(s_{t+1}) = V^\pi(s_t)$, i.e., the true value function, is found.

While TD actively exploits the Markov property, MC does not at all. Of course, this makes MC more well-rounded and applicable to non-Markovian environments while TD's wrong assumptions can hurts its efficiency. However, for very complicated environments we might not be able to use tabular methods (**??**), but have to resort to function approximations (**??**) where MC excels. Also, TD is more efficient to the initial values as MC which makes sense as MC might not use them at all.

See Table 6.1 for a comparison of DP, MC, and TD with regard to bootstrapping and sampling.

6.1.1 Backup

6.2 TD(λ)

Looking at (6.2), one might ask why we only look *one* step into the future an not, for instance, three steps. In fact, this is possible and we can consider the *n-step return*

$$J_t^{(n)} := r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

with n-step TD learning,

$$V(s_t) \leftarrow V(s_t) + \alpha\big(J_t^{(n)} - V(s_t)\big).$$

Increasing n has the advantage of reducing the estimate's bias for the cost of increasing its variance (as with a larger n, more random factors come into play). With $n \to \infty$, this update approaches (6.1), the MC update.

6.2.1 Forward-View

The natural next idea is to average multiple n-step returns to combine information from different time steps. For instance, we could average the 2- and 4-step returns $\tilde{J} = J^{(2)}/2 + J^{(4)}/2$. But can we come up with a principled way of weighing different n-step returns? A fruitful idea is to use an exponential weighting, i.e., to weigh samples in the future exponentially less than close samples. This is the idea of the *λ-return*

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

which uses the weighting function $(1-\lambda)\lambda^{n-1}$, $\lambda \in [0,1)$ for the n-step return. The factor $(1-\lambda)$ is necessary for the sum to be convex, i.e.,

$$(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1} = (1-\lambda)\sum_{n=0}^{\infty}\lambda^{n} = (1-\lambda)\frac{1}{1-\lambda} = 1.$$

We call the resulting update $V(s_t) \leftarrow V(s_t) + \alpha\big(J_t^{\lambda} - V(s_t)\big)$ *forward-view TD(λ)*. Of course now we are back to the initial problem we tried to avoid using TD methods: the forward-view TD target can only be computed for complete episodes!

6.2.2 Backward-View and Eligibility Traces

We saw that forward-view TD(λ) provides a good approach to reduce the bias of the TD target. However, it had the disadvantage of only working in episodic settings, i.e., only with complete episodes. An alternative is *backward-view TD(λ)* following a similar idea, but looking backward in time, allowing updates from incomplete episodes. However, now we have the *credit assignment problem:* how to weigh past rewards and states? Two common heuristics are the *frequency* and *recency* heuristic, i.e., assigning credit to the most frequent or most recent states, respectively. Both of these ideas can be summarized into *eligibility traces*. For every time step t, we keep an eligibility value $z_t(s)$ for every state s describing the weight of state s at time step t. Starting with $z_{-1}(s) = 0$, we iteratively compute the next time step as

$$z_t(s) = \gamma\lambda z_{t-1}(s) + \mathbb{1}[s = s_t],$$

where γ is the discount factor and λ is the exponential decay for the eligibility. The intuition of this update is as follows: Whenever we take a step in the environment, we decrease the weight by multiplying with λ (implementing the recency heuristic); by adding one whenever we see a state, we implement the frequency heuristic (by adding one often for frequent states) and "give the recency heuristic something to work with," i.e., some value it can actually decrease. The update (applied to all states and not just the current state) then is

$$V(s) \leftarrow V(s) + \alpha\delta_t z_t(s)$$

where δ_t is the TD error. Note that for $\lambda = 0$, the update collapses exactly on the TD(0) update (6.2), hence the name. The algorithm is summarized in algorithm 6.

6.3 Example

6.4 Wrap-Up

- differences of DP, MC, and TD
- definition of eligibility traces
- computation of TD(λ)
- Additional reading material:
  - Book: "Introduction to Reinforcement Learning" (Sutton and Barto, 2018), Chapters 6, 7, and 12

Algorithm 6: Backward-View TD(λ)

```
Input: environment
Output: value function
repeat
    $z(s) \leftarrow 0$ for all $s \in \mathcal{S}$
    initialize $s$
    while $s$ is not terminal do
        take action $a \sim \pi(\cdot \mid s)$, observe reward $r$ and next state $s'$
        $\delta \leftarrow r + \gamma V(s') - V(s)$
        $z(s) \leftarrow z(s) + 1$
        foreach $\tilde{s} \in \mathcal{S}$ do
            $V(\tilde{s}) \leftarrow V(\tilde{s}) + \alpha \delta z(\tilde{s})$
            $z(\tilde{s}) \leftarrow \gamma \lambda z(\tilde{s})$
        $s \leftarrow s'$
until converged
return $V$
```

7 Tabular Reinforcement Learning

In this chapter we cover methods from *tabular* RL where the state-action-space is small enough such that we can represent the action-value function as a table. We distinguish two categories of methods: on- and off-policy learning. In the former the algorithm learns “on the job,” i.e., the experience is sampled from the policy that is being trained. In the latter the algorithms learns “by looking over someone’s shoulder,” i.e., the experience is samples from a different policy q that the one that is being trained.

7.1 On-Policy Methods

In this section we discuss tabular on-policy methods.

7.1.1 Monte-Carlo Methods and Exploration vs. Exploitation

In *generalized* PI, we do not evaluate the state-value function (for which a greedy policy improvement needs the transition dynamics), but instead evaluate the action-value. However, just using the deterministic policy found during policy improvement for MC policy evaluation means that we do not have exploration (no “new” actions are tried)! This brings us to the *exploration vs. exploitation dilemma.*

During decision-making, we have two options: *exploit* the current knowledge to make the best known decision or *explore* and gather more information. In other words, the best long-term options (which we want to find) might involve short-term sacrifices. This dilemma is a fundamental problem in RL and we do not have a satisfying solution yet. Two common approaches are *ϵ-greedy*,

$$a = \begin{cases} a^* & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases},$$

and *softmax.* A softmax policy biases the action selection towards exploitation. The most common softmax is the Boltzmann distribution

$$\pi(a \,|\, s) = \frac{\exp\{Q(s, a)/\tau\}}{\exp\{\sum_{a' \in \mathcal{A}} Q(s, a')/\tau\}}$$

with a *temperature* τ defining how uniform the distribution is. For $\tau \to \infty$, the distribution approaches a uniform distribution of a and for $\tau = 0$ it acts greedy.

ϵ-Greedy Exploration and Policy Improvement

The simplest exploration strategy, ϵ-greedy, can also be formulated as a distribution:

$$\pi(a \,|\, s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a = \arg\max_{a \in \mathcal{A}} Q(s, a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

Interestingly, the ϵ-greedy policy still causes monotonic improvements as for any ϵ-greedy policy π, the ϵ-greedy policy w.r.t. Q^π fulfills the premise of the policy improvement theorem (Theorem 10):

$$\begin{aligned}
Q^\pi\big(s, \pi'(s)\big) &= \sum_{a \in \mathcal{A}} \pi'(a \,|\, s) Q^\pi(s, a) = \sum_{a \in \mathcal{A}} Q^\pi(s, a) \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^\pi(s, a) \\ \epsilon/m & \text{otherwise} \end{cases} \\
&\overset{(*)}{=} \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s, a) + (1 - \epsilon) \max_{a \in \mathcal{A}} Q^\pi(s, a) \\
&\overset{(\dagger)}{\geq} \frac{\epsilon}{m} \sum_{a \in \mathcal{A}} Q^\pi(s, a) + (1 - \epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a \,|\, s) - \epsilon/m}{1 - \epsilon} Q^\pi(s, a) \overset{(\ddagger)}{=} \sum_{a \in \mathcal{A}} \pi(a \,|\, s) Q^\pi(s, a) = V^\pi(s)
\end{aligned}$$

In step $(*)$ we explicitly plugged in the policy π' using that its first case is applied exactly once for the maximum of $Q^\pi(s, a)$. Hence, we can pull it out of the sum. In step $(\dagger)$, we "inverse plugged in" an ϵ-greedy policy using the relations

$$\frac{\pi(a \,|\, s) - \epsilon/m}{1 - \epsilon} = \frac{\epsilon/m - \epsilon/m}{1 - \epsilon} = 0 \qquad\qquad \frac{\pi(a \,|\, s) - \epsilon/m}{1 - \epsilon} = \frac{\epsilon/m + 1 - \epsilon - \epsilon/m}{1 - \epsilon} = 1$$

for the "maximum" a and all others, respectively. Note that this maximum is *not* w.r.t. $Q^\pi(s, a)$ and hence the sum over all actions must be less than or equal to the maximum value of $Q^\pi(s, a)$. Finally, in step $(\ddagger)$, we used the definition of π and that $Q^\pi(s, a)$ is its action-value function.

Using ϵ-greedy exploration therefore gives monotonic improvement while not shutting canceling exploration during policy evaluation!

GLIE Monte-Carlo

Definition 25 (Greedy in the Limit of Infinite Exploration (GLIE))**.** A policy π is *greedy in the limit of infinite exploration (GLIE)* if all state-action pairs are explore infinitely many times,

$$\lim_{k \to \infty} N^{(k)}(s, a) \to \infty,$$

and the policy converges to a greedy policy,

$$\lim_{k \to \infty} \pi_k(a \,|\, s) = \mathbb{1}\big[a = \arg\max_{a' \in \mathcal{A}} Q^{(k)}(s, a')\big].$$

Here k is the policy iteration index.

GLIE MC achieves this by choosing $\epsilon_k = 1/k$ and $\pi = \epsilon$-greedy. We then also have the following theorem.

Theorem 12 (Convergence of GLIE Monte-Carlo)**.** *GLIE MC converges to the* optimal *action-value function.*

7.1.2 TD-Learning: SARSA

We already saw in chapter 6 that TD learning has several advantages over MC, namely lower variance, online capabilities, and learning from incomplete sequences. So a natural idea is to replace MC with TD in the control loop, i.e., applying TD to the action-value function using ϵ-greedy policy improvement. In SARSA (for "state, action, reward, state, action"), we use the policy's proposed action during the TD update, i.e.,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big),$$

where $a_{t+1} \sim \pi^Q(\cdot \,|\, s_{t+1})$ is sampled from a policy derived from Q (indicated by the superscript). Hence, SARSA is an on-policy algorithm. The pseudocode is depicted in algorithm 7. If we choose the step sizes α wisely, we also have convergence guarantees!

Theorem 13 (Convergence of SARSA). *If the policies $\pi_t(s, a)$ constitute a GLIE sequence and the step sizes α_t constitute a Robbins-Monro sequence, i.e., $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$, SARSA converges to the optimal action-value function.*

Algorithm 7: SARSA

```
   Input: environment
   Output: (optimal) action-value function
1  initialize $Q$ arbitrarily except $Q(\mathrm{terminal}, \cdot) = 0$
2  repeat
3      initialize $s$
4      choose $a \sim \pi^Q(\cdot \mid s)$
5      while $s$ is not terminal do
6          take action $a$, observe reward $r$ and next state $s'$
7          choose $a' \sim \pi^Q(\cdot \mid s')$
8          $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
9          $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
10         $s \leftarrow s'$
11         $a \leftarrow a'$
12 until converged
13 return $Q$
```

Eligibility Traces and SARSA(λ)

Similar to TD(λ) (section 6.2), could use the n-step return $J_t^{(n)}$ in the SARSA update,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(J_t^{(n)} - Q(s_t, a_t)\big).$$

We can again generalize this to the λ-return J_t^{λ} with exponential decay. However, we have the same disadvantage with this *forward-view* as before: now we need complete sequences to calculate the TD target! Hence, we instead use the equivalent *backward-view* and eligibility traces for state-action pairs,

$$z_t(s, a) = \gamma\lambda z_{t-1}(s, a) + \mathbb{1}[s = s_t, a = a_t],$$

kicking the recursion off with $z_{-1}(s, a) = 0$.

Example

7.2 Off-Policy Methods

While on-policy methods are great and often exhibit convergence guarantees, they have a major flaw: exploration has to be built into the policy, for instance using ϵ-greed. *Off-policy* methods, on the other hand, learn about a target policy $\pi(a \mid s)$ while following a behavioral policy $q(a \mid s)$. We can directly transfer this approach to a variety of tasks such as learning directly from a human or other agents, re-use experience of old policies, explore while learning an optimal policy, or learning multiple policies while following a single.

Algorithm 8: SARSA(λ)

```
Input: environment
Output: (optimal) action-value function
1  initialize $Q$ arbitrarily except $Q(\text{terminal}, \cdot) = 0$
2  repeat
3      $z(s, a) \leftarrow 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
4      initialize $s$
5      choose $a \sim \pi^Q(\cdot \mid s)$
6      while $s$ is not terminal do
7          take action $a$, observe reward $r$ and next state $s'$
8          choose $a' \sim \pi^Q(\cdot \mid s')$
9          $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
10         $z(s, a) \leftarrow z(s, a) + 1$
11         foreach $\tilde{s} \in \mathcal{S}$ do
12             $Q(s, a) \leftarrow Q(s, a) + \alpha\delta z(s, a)$
13             $z(s, a) \leftarrow \gamma\lambda z(s, a)$
14         $s \leftarrow s'$
15         $a \leftarrow a'$
16 until converged
17 return $Q$
```

7.2.1 Monte-Carlo

The core concept for off-policy MC is *importance sampling* (subsection 2.2.3). Hence, we update the return J_t sampled from a behavioral policy q by the importance sampling corrections,

$$J_t^{\pi/q} = \frac{\pi(a_t \mid s_t)}{q(a_t \mid s_t)} \frac{\pi(a_{t+1} \mid s_{t+1})}{q(a_t \mid s_t)} \cdots \frac{\pi(a_T \mid s_T)}{q(a_T \mid s_T)} J_t.$$

Note that we only need to correct using the policy distributions even though the return distribution contains factors of the state dynamics. However, these are the same for both policies so they cancel in the importance sampling weights. We then update the action-value towards this corrected return:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(J_t^{\pi/q} - Q(s_t, a_t)\big).$$

While this allows off-policy MC, importance sampling can dramatically increase the variance even further! Also, it is not possible to sample actions for which q is zero but π is non-zero.

7.2.2 TD and Q-Learning

Similar to off-policy MC, we can also use importance sampling in TD learning by weighing the TD error accordingly:

$$\rho_t = \frac{\pi(a_t \mid s_t)}{q(s_t \mid a_t)} \qquad\qquad V(s_t) \leftarrow V(s_t) + \alpha\rho_t\delta_t.$$

Compared to MC, the TD has much lower variance and the policies only need to be similar over a single step, hence the risk of $q \approx 0$ is reduced. But we can do better and mitigate the use of importance sampling at all!

In *Q-learning,* we update the action-value towards the value of an alternative action,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_{t+1} + \gamma Q(s_t, a') - Q(s_t, a_t)\big), \tag{7.1}$$

where $a_t \sim q(\cdot \mid s_t)$ is sampled from the behavioral policy and $a' \sim \pi(\cdot \mid s_{t+1})$ is sampled from another policy. With a greedy π and, for instance, an ϵ-greedy q, the Q-learning target simplifies to

$$r_{t+1} + \gamma Q(s_{t+1}, a') = r_{t+1} + \gamma Q\big(s_{t+1}, \arg\max_{a' \in \mathcal{A}} Q(s_{t+1}, a')\big) = r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$$

which we can directly plug into (7.1) yielding the following update rule:

$$Q(s, a) \leftarrow Q(s, t) + \alpha\big(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\big) \tag{7.2}$$

The algorithm is summarized in algorithm 9. For convergence, we have again a nice guarantee:

Theorem 14 (Convergence of Q-Learning). *Q-learning converges to the optimal action value function.*

Algorithm 9: Q-Learning

```
   Input: environment
   Output: (optimal) action-value function
1  initialize $Q$ arbitrarily except $Q(\text{terminal}, \cdot) = 0$
2  repeat
3      initialize $s$
4      while $s$ is not terminal do
5          take action $a \sim q^Q(\cdot \mid s)$, observe reward $r$ and next state $s'$
6          $\delta \leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)$
7          $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
8          $s \leftarrow s'$
9  until converged
10 return $Q$
```

Example

7.3 Remarks

Both SARSA and Q-learning are great algorithms on their own, but of course have some differences. While SARSA is an on-policy TD algorithm, Q-learning is off-policy. Also, if $\epsilon \neq 0$, SARSA performs better online, but for $\epsilon \to 0$, both converge to the optimal solution. Note that for a true online application, constant exploration (and therefore $\epsilon \neq 0$) can be necessary! Table 7.1 shows the relationship between DP and TD.

7.4 Wrap-Up

- differences of on- and off-policy learning
- relationship between model-free control and generalized PI

Full Backup (Dynamic Programming)	**Sample Backup (Temporal Differences)**
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}_{a,s'}\left[R(s,a) + \gamma V(s') \mid s\right]$	TD Learning $V(s_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma V(s_{t+1})$
Q-Policy Iteration $Q(s,a) \leftarrow R(s,a) + \gamma \mathbb{E}_{s',a'}\left[Q(s',a') \mid s,a\right]$	SARSA $Q(s_t,a_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma Q(s_{t+1},a_{t+1})$
Q-Value Iteration $Q(s,a) \leftarrow R(s,a) + \gamma \mathbb{E}_{s'}\left[\max_{a' \in \mathcal{A}} Q(s',a') \mid s\right]$	Q-Learning $Q(s_t,a_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1},a')$

Table 7.1: Relationship Between Dynamic Programming and Temporal Difference Learning

- sufficient conditions for an effective exploration strategy
- how to use ϵ-greedy for exploration
- SARSA and its application in on-policy control
- incorporation of λ-returns in TD control
- off-policy learning with importance sampling
- off-policy control with Q-learning without importance sampling
- relationship of the Bellman equations and the TD target
- Additional reading material:
  - Book: “Introduction to Reinforcement Learning” (Sutton and Barto, 2018), Chapters 5, 6, 7, and 12

8 Function Approximation

So far, we considered only discrete (and finite) MDPs for which we were able to represent the value functions as tables. However, most RL problems are not discrete and the state-action-space is continuous and infinite! To handle these kinds of environments, we need to use *function approximation* for the value functions (and also policies). As before, we split this chapter into on- and off-policy methods.

8.1 On-Policy Methods

For large state-spaces, it is infeasible to store or update $V(s)$ and $Q(s, a)$ in a table. Hence, we use function approximations

$$\hat{V}(s; \boldsymbol{w}) \approx V(s) \qquad\qquad \hat{Q}(s, a; \boldsymbol{w}) \approx Q(s, a)$$

with weights $\boldsymbol{w} \in \mathbb{R}^d$. Common choices are linear regression, neural networks, regression trees, and Gaussian processes. As the weights are far fewer than the states (otherwise the approximation would not make sense), changing a weight affects the approximation of the value *for all states* and may even decrease the accuracy of some. We measure the accuracy using the *mean squared value error (MSVE)*

$$\overline{VE}(s; \boldsymbol{w}) \coloneqq \sum_{s \in \mathcal{S}} \mu(s) \big[V(s) - \hat{V}(s; \boldsymbol{w})\big]^2$$

where $\mu(s) \geq 0$, $\sum_{s \in \mathcal{S}} \mu(s) = 1$ weighs the importance of the states. For on-policy methods, $\mu(s)$ is the "fraction of time" spent in state s following π, i.e.,

$$\mu(s) = \frac{1}{T + 1} \sum_{t=0}^{T} \mathbb{1}[s_t = s].$$

8.1.1 Stochastic Gradient Descent

Assuming the real value function $V(s)$ is known, we can find suitable parameters $\boldsymbol{w}$ for a differentiable $V(s; \boldsymbol{w})$ by stochastic gradient descent (SGD). For each time step $t = 0, 1, \dots$, we can then locally update the weights using the update rule

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \frac{1}{2}\alpha \boldsymbol{\nabla}_{\boldsymbol{w}} \big[V(s_t) - \hat{V}(s; \boldsymbol{w})\big]^2 \Big|_{\boldsymbol{w} = \boldsymbol{w}_t} = \boldsymbol{w}_t + \alpha \big[V(s_t) - \hat{V}(s; \boldsymbol{w}_t)\big] \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{V}(s_t; \boldsymbol{w}) \big|_{\boldsymbol{w} = \boldsymbol{w}_t} \tag{8.1}$$

where $\alpha > 0$ is the *learning rate.* Note that we update the value function for each step in the trajectory, hence we use a "time" index here. For a proper decay of the learning rate, this is guaranteed to converge to a local optimum. However, we usually do not have access to the real value function—this is why we ultimately need learning!

8.1.2 Gradient Monte-Carlo

As we usually do not have access to the real value function, we replace its appearance in (8.1) with an arbitrary value estimate U_t:

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\big[U_t - \hat{V}(s; \boldsymbol{w}_t)\big]\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s_t; \boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t}$$

If U_t is an unbiased estimate of $V(s_t)$, i.e., $\mathbb{E}[U_t] = V(s_t)$, then convergence is guaranteed for a proper α-decay. One suitable estimate is the MC estimate $J_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$ for which $\mathbb{E}[J_t] = V(s_t)$ holds by definition of $V(s_t)$[1]. This approach is summarized in algorithm 10. When using a linear function approximator $\hat{V}(s, \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{\phi}(s)$ with features $\boldsymbol{\phi}(s)$, the gradient becomes especially simple: $\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s_t; \boldsymbol{w}) = \boldsymbol{\phi}(s)$.

Algorithm 10: Gradient Monte-Carlo

```
  Input: policy $\pi$, differentiable approximator $\hat{V}$ with parameters $\boldsymbol{w}$
  Output: estimated parameters $\boldsymbol{w}$
1 initialize $\boldsymbol{w}$ arbitrarily
2 repeat
3     $(s_0; r_1, s_1; r_2, s_2; \dots; r_{T-1}, s_{T-1}; r_T) \leftarrow$ generate episode
4     foreach $t = 0, 1, \dots, T$ do
5         $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
6         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big[V(s_t) - \hat{V}(s; \boldsymbol{w})\big]\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s_t; \boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}}$
7 until convergence
8 return $\boldsymbol{w}$
```

8.1.3 Semi-Gradient Methods

In chapter chapter 6, we saw that MC methods are not the only way for estimating V and indeed, we could also use DP or bootstrapped TD targets, allowing step-based updates and learning from incomplete episodes. The DP and n-step TD targets are:

$$U_t = \mathbb{E}_{a_{t+1}, s_{t+1}}\big[R(s_t, a) + \gamma\hat{V}(s_{t+1}; \boldsymbol{w})\big] \qquad\qquad U_t = \sum_{k=t+1}^{t+n} \gamma^{k-t-1} r_k + \gamma^n \hat{V}(s_{t+n}; \boldsymbol{w}).$$

Note, however, that the targets themselves depend on $\boldsymbol{w}$! As we just ignore this dependence, the result algorithms are called *semi-gradient* TD methods. Using a linear approximator $\hat{V}(s, \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{\phi}(s)$ with features

[1]Note that this expectation holds for a specific t, but $\mathbb{E}[J_t] = V(s)$ does *not* hold, hence every-visit MC prediction is biased but gradient MC is not.

$\phi(s)$, the update rule of semi-gradient TD(0) becomes

$$\begin{aligned}
\boldsymbol{w}_{t+1} &\leftarrow \boldsymbol{w}_t + \alpha\Big[r_{t+1} + \gamma\hat{V}(s_{t+1};\boldsymbol{w}_t) - \hat{V}(s_t;\boldsymbol{w}_t)\Big]\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t} \\
&= \boldsymbol{w}_t + \alpha\Big[r_{t+1} + \gamma\boldsymbol{w}_t^\top\boldsymbol{\phi}(s_{t+1}) - \boldsymbol{w}_t^\top\boldsymbol{\phi}(s_t)\Big]\boldsymbol{\phi}(s_t) \\
&= \boldsymbol{w}_t + \alpha\Big[r_{t+1}\boldsymbol{\phi}(s_t) + \gamma\boldsymbol{w}_t^\top\boldsymbol{\phi}(s_{t+1})\boldsymbol{\phi}(s_t) - \boldsymbol{w}_t^\top\boldsymbol{\phi}(s_t)\boldsymbol{\phi}(s_t)\Big] \\
&\overset{(*)}{=} \boldsymbol{w}_t + \alpha\Big[r_{t+1}\boldsymbol{\phi}(s_t) + \gamma\boldsymbol{\phi}(s_t)\boldsymbol{\phi}^\top(s_{t+1})\boldsymbol{w}_t - \boldsymbol{\phi}(s_t)\boldsymbol{\phi}^\top(s_t)\boldsymbol{w}_t\Big] \\
&= \boldsymbol{w}_t + \alpha\Big[r_{t+1}\boldsymbol{\phi}(s_t) + \boldsymbol{\phi}(s_t)\big(\gamma\boldsymbol{\phi}(s_{t+1}) - \boldsymbol{\phi}(s_t)\big)^\top\boldsymbol{w}_t\Big] \\
&= \boldsymbol{w}_t + \alpha\Big[r_{t+1}\boldsymbol{\phi}(s_t) - \boldsymbol{\phi}(s_t)\big(\boldsymbol{\phi}(s_t) - \gamma\boldsymbol{\phi}(s_{t+1})\big)^\top\boldsymbol{w}_t\Big]
\end{aligned}$$

where $(*)$ is due to $\underbrace{\boldsymbol{v}_1^\top\boldsymbol{v}_2}_{\text{scalar}}\boldsymbol{v}_3 = \boldsymbol{v}_3\boldsymbol{v}_1^\top\boldsymbol{v}_2$.

Algorithm 11: Semi-Gradient TD(0)

```
   Input: policy $\pi$, differentiable approximator $\hat{V}$ with parameters $\boldsymbol{w}$
   Output: estimated parameters $\boldsymbol{w}$
1  initialize $\boldsymbol{w}$ arbitrarily
2  repeat
3      initialize $s$
4      while $s$ is not terminal do
5          take action $a \sim \pi(\cdot \mid s)$, observe reward $r$ and next state $s'$
6          $\delta \leftarrow r + \gamma\hat{V}(s';\boldsymbol{w}) - \hat{V}(s;\boldsymbol{w})$
7          $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}}$
8          $s \leftarrow s'$
9  until convergence
10 return $w$
```

8.1.4 Semi-Gradient SARSA

Similar to semi-gradient TD(0), we can also approximate the action-value function instead of the state-value function using SARSA. The one-step SARSA update is then

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\big[r_{t+1} + \gamma\hat{Q}(s_{t+1}, a_t;\boldsymbol{w}_t) - \hat{Q}(s_t, a_t;\boldsymbol{w}_t)\big]\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{Q}(s_t, a_t;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t}.$$

The complete algorithm is summarized in algorithm 12

8.2 Off-Policy Methods

Of course, we can apply function approximation not only in on-, but also in off-policy methods. For instance, we can apply importance sampling to get off-policy TD(0):

$$\rho_t = \frac{\phi(a_t \mid s_t)}{q(a_t \mid s_t)} \qquad\qquad \boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha\rho_t\delta_t\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{Q}(s_t, a_t;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t}$$

Algorithm 12: Semi-Gradient SARSA

```
Input: policy $\pi$, differentiable approximator $\hat{Q}$ with parameters $\boldsymbol{w}$
Output: estimated parameters $\boldsymbol{w}$
1  initialize $\boldsymbol{w}$ arbitrarily
2  repeat
3      initialize $s$
4      choose $a \sim \pi^Q(\cdot \mid s)$
5      while true do
6          take action $a \sim \pi(\cdot \mid s)$, observe reward $r$ and next state $s'$
7          if $s'$ is terminal then
8              break
9          choose $a' \sim \pi^Q(\cdot \mid s')$ $\delta \leftarrow r + \gamma \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w})$
10         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \delta \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t}$
11         $s \leftarrow s'$
12         $a \leftarrow a'$
13 until convergence
14 return $w$
```

with, for instance, $\delta_t = r_{t+1} + \gamma \hat{V}(s_{t+1}; \boldsymbol{w}) - \hat{V}(s_t; \boldsymbol{w}_t)$.

While off-policy methods might allow better exploration, they have a major flaw: updating the value function *on-policy* is important for convergence. Hence, off-policy methods with approximations can *diverge!*

Example

8.3 The Deadly Triad

In fact, we have a lot of instability in methods that are based on the following elements: function approximation, bootstrapping, and off-policy training. However, we need everyone of these! Function approximation to scale, bootstrapping for data efficiency, and off-policy training for heterogeneous experience.

8.4 Offline Methods

So far, we only looked at *online* methods, i.e., methods that only use the current transitions and which can be applied "on the job." *Offline* or *batch* methods, on the other hand, can use data (transitions and trajectories) that was collected previously.

8.4.1 Least-Squares TD and Least-Squares PI

When using linear function approximation, semi-gradient methods are guaranteed to converge to a point near a local optimum. Let $\tilde{\boldsymbol{w}}$ be that solution, then it satisfies the fixed point equation

$$\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{w}} + \alpha(\boldsymbol{b} - \boldsymbol{A}\tilde{\boldsymbol{w}}) \tag{8.2}$$

with $\boldsymbol{b} := r_{t+1}\boldsymbol{\phi}(s_t)$ and $\boldsymbol{A} := \boldsymbol{\phi}(s_t)\big(\boldsymbol{\phi}(s_t) - \gamma\boldsymbol{\phi}(s_{t+1})\big)^\top$. Hence, the fixed point is $\tilde{\boldsymbol{w}} = \boldsymbol{A}^{-1}\boldsymbol{b}$ and the MSVE at the fixed point is bounded by

$$\overline{VE}(s, \tilde{\boldsymbol{w}}) \leq \frac{1}{1-\gamma} \min_{\boldsymbol{w}} \overline{VE}(s, \boldsymbol{w}).$$

If we directly finds the fixed point by solving (8.2), we arrive at *least-squares TD (LSTD)*, an *offline* variant of semi-gradient TD(0). Given a set of transitions, LSTD first compute the weight matrix and vector,

$$\hat{\boldsymbol{A}}_t = \sum_{k=0}^{t-1} \boldsymbol{\phi}(s_k)\big(\boldsymbol{\phi}(s_k) - \gamma\boldsymbol{\phi}(s_{k+1})\big)^\top \qquad \hat{\boldsymbol{b}}_t = \sum_{k=0}^{t-1} r_{k+1}\boldsymbol{\phi}(s_k),$$

and then solves then computes the fixed point using $\boldsymbol{w}_t = \big(\hat{\boldsymbol{A}}_t + \varepsilon\boldsymbol{I}\big)^{-1}\hat{b}_t$ where ϵ is a small regularization constant.

Least-squares policy iteration (LSPI) extends the idea of LSTD to learning the action-value function, forming LSTDQ and combines it with greedy policy improvement. A sketch is shown in algorithm 13.

Algorithm 13: Least-Squares Policy Iteration

```
   Input: transition data set $\mathcal{D} = \langle s_i, a_i, r_i, s'_i \rangle_{i=1}^N$
   Output: action-value function $Q$ and policy $\pi$
1  initialize $\pi$ arbitrarily
2  repeat
3      $Q \leftarrow$ LSTDQ$(\mathcal{D}, \pi)$
4      $\pi(s) \leftarrow \arg\max_{a\in\mathcal{A}} Q(s,a)$ for all $s \in \mathcal{S}$
5  until convergence
6  return $Q$, $\pi$
```

8.4.2 Fitted Q-Iteration

Another approach to offline RL is *fitted Q-iteration*. Given a data set of transitions, it solves a sequence of regression problems to find the action-value function. For regression trees and kernel averaging, there are also some stability guarantees. A sketch of the algorithm is given in algorithm 14.

Algorithm 14: Fitted Q-Iteration

```
   Input: transition data set $\mathcal{D} = \langle s_i, a_i, r_i, s'_i \rangle_{i=1}^N$, regressor $\hat{Q}$
   Output: action-value function $Q$
1  initialize $\hat{Q}^{(0)}$ arbitrarily
2  $k \leftarrow 0$
3  repeat
4      $\mathcal{T} \leftarrow \big\langle s_i,\ a_i,\ r_i + \gamma \max_{a\in\mathcal{A}} \hat{Q}^{(k)}(s'_i, a) \big\rangle_{i=1}^N$    // build training dataset
5      $Q^{(k+1)} \leftarrow$ freshly trained regressor using $\mathcal{T}$
6      $k \leftarrow k+1$
7  until convergence
8  return $Q^{(\infty)}$
```

8.5 Wrap-Up

- continuous problems in RL
- the need for function approximation
- usage of function approximation in RL
- consequences of function approximation
- challenged of off-policy training with function approximation
- Additional reading material:
  - Book: “Introduction to Reinforcement Learning” (Sutton and Barto, 2018), Chapters 9, 10, and 11

9 Policy Search

So far, we always learned/estimated a value function and extracted a policy from it. However, why bother to learn a value function and not directly learn a policy which is what we are actually interested in? This is the idea of *policy search*. Given a MDP, policy search can be formalized as an explicit optimization problem over the expected reward,

$$\pi^* = \arg\max_{\pi} \underbrace{\mathbb{E}_{\tau\sim\pi}[J(\tau)]}_{\mathcal{J}(\pi):=} = \arg\max_{\pi} \mathcal{J}_\pi, \tag{9.1}$$

where $J(\tau)$ is the cumulative discounted reward for a trajectory τ and the expectation is w.r.t. the initial state distribution, state dynamics, and policy. Policy search has some major advantages:

- obviously, no need to learn a value function which might be difficult
- we can encode domain knowledge into the policy
- the policy may be initialized with other methods (e.g., with a (suboptimal) result from imitation learning)
- we do not have to solve a maximization problem when executing the policy (i.e., there is no need to maximize the value function)

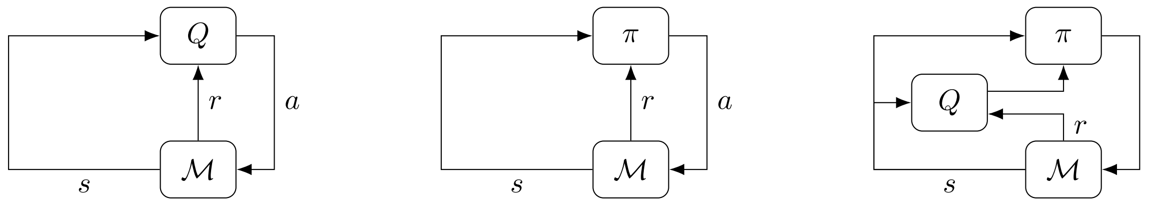
Of course, we also have some downsides such as no guarantees of convergence to the optimum (but instead to a local optima) and that it is pretty inefficient as policy search depends on MC rollouts in the environment. Hence, the methods usually exhibit high variance in the resulting policies. Figure 9.1 shows a comparison of value-based methods vs. policy search vs. actor-critic.

As policy search can be applied to both discrete and continuous control actions, we will confine ourselves to the discussion of a discrete set of continuous control variables, i.e., a vector[1] $a = (a_1, a_2, \dots, a_M)$. Like before, *exploration is crucial!* For continuous variables, the most common option is to use a Gaussian action distribution $\pi_{\boldsymbol{\theta}}(a \mid s) = \mathcal{N}\big(a \,\big|\, \mu_{\boldsymbol{\theta}}(s), \Sigma_{\boldsymbol{\theta}}(s)\big)$ where the mean and covariance matrix are given by a (usually learnable) state-dependent function approximator with parameters $\boldsymbol{\theta}$. This can, for instance, be a NN. For discrete actions, choosing an exploration strategy like a Boltzmann policy (see subsection 7.1.1) is usually a good choice. In fact, we will also confine ourselves to *parametric* policies, i.e., policies $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$ that govern our search space. We can then re-frame the optimization problem (9.1) as

$$\pi^* = \arg\max_{\pi\in\{\pi_{\boldsymbol{\theta}_i}\}} \mathcal{J}(\pi_{\boldsymbol{\theta}}) = \arg\max_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$$

That is, we optimize our objective just w.r.t. the policy's parameters $\boldsymbol{\theta}$. We will now go over to a general discussion of *policy gradient* methods which we will focus on in this chapter.

[1]Note that even though a is a vector now, we will stick to the lightface notation for consistency.



(a) Value-Based (Critic-Only)

- explicit value function
- implicit policy

(b) Policy Search (Actor-Only)

- no value function
- explicit policy

(c) Actor-Critic

- explicit value function
- explicit policy

Figure 9.1: Value-Based vs. Policy Search vs. Actor-Critic

9.1 Policy Gradient

In *policy gradient (PG)* methods, we maximize the objective $\mathcal{J}(\boldsymbol{\theta})$ directly by taking its gradient w.r.t. the policy parameters $\boldsymbol{\theta}$, the so-called *policy gradient* $\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}(\boldsymbol{\theta})$. algorithm 15 shows a (very rough) prescription on how this works.

Algorithm 15: Policy Search using Policy Gradient

```
  Input: environment, differential policy $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$
  Output: optimized policy $\pi$
1 initialize $\boldsymbol{\theta}$ arbitrarily
2 repeat
      // Collect data and perform gradient update.
3     $\mathcal{D} \leftarrow$ generate rollouts using $\pi_{\boldsymbol{\theta}}$
4     $\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}(\boldsymbol{\theta}) \leftarrow$ compute gradient using $\mathcal{D}$
5     $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}(\boldsymbol{\theta})$
6 until convergence
7 return $\pi_{\boldsymbol{\theta}^{(\infty)}}$
```

9.1.1 Computing the Gradient

The most pressing question of PG methods is how we actually compute the gradient as it goes through an expectation. The naive approach are *finite differences,* i.e., approximating the derivative w.r.t. the i-th component by

$$\frac{\partial}{\partial\theta_i}\mathcal{J}(\boldsymbol{\theta}) \approx \frac{1}{\epsilon}\big(\mathcal{J}(\boldsymbol{\theta} + \boldsymbol{e}_i\epsilon) - \mathcal{J}(\boldsymbol{\theta})\big)$$

with the i-th unit vector $\boldsymbol{e}_i$ and a sufficiently small ϵ. However, this black-box approach has an extremely high variance, is computationally inefficient, and scaled bad in the number of parameters. Also, exploration is performed on a parameter level rather than on policy level (as we vary parameters).

Least-Squares-Based Finite Differences

A more practical approach to using the finite difference gradient is *least-squares-based finite difference (LSFD)*. Instead of component-wise estimation, we evaluate the policy N times for small but arbitrary perturbations

$\{\boldsymbol{\delta\theta}^{[i]}\}_{i=1}^{N}$ and once for the unperturbed $\boldsymbol{\theta}$. With

$$\delta J^{[i]} := J(\boldsymbol{\theta} + \boldsymbol{\delta\theta}^{[i]}) - J(\boldsymbol{\theta}),$$

we can find a gradient estimate by solving

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{FD}} \mathcal{J}(\boldsymbol{\theta}) = \left(\boldsymbol{\delta\Theta}^{\top}\boldsymbol{\delta\Theta}\right)^{-1} \boldsymbol{\delta\Theta}^{\top} \boldsymbol{\delta J} \tag{9.2}$$

where $\boldsymbol{\delta\Theta} := \left(\boldsymbol{\delta\theta}^{[1]}, \boldsymbol{\delta\theta}^{[2]}, \dots, \boldsymbol{\delta\theta}^{[N]}\right)^{\top}$ and $\boldsymbol{\delta J} := \left(\delta J^{[1]}, \delta J^{[2]}, \dots, \delta J^{[N]}\right)^{\top}$ collect the perturbations in the parameters and the return.

Proof. For a differentiable $\mathcal{J}(\boldsymbol{\theta})$, we have its Taylor expansion

$$\mathcal{J}(\boldsymbol{\theta}) = \mathcal{J}(\boldsymbol{\theta}_0) + \left(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right)^{\top} \boldsymbol{\delta\theta} + \mathcal{O}\left(\boldsymbol{\delta\theta}^2\right)$$

around $\boldsymbol{\theta}_0$. With $\delta\mathcal{J}(\boldsymbol{\theta}) := \mathcal{J}(\boldsymbol{\theta}) - \mathcal{J}(\boldsymbol{\theta}_0)$ and by dropping the higher terms, we get an estimate of the gradient by solving

$$\delta\mathcal{J}(\boldsymbol{\theta}) = \left(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right)^{\top} \boldsymbol{\delta\theta}.$$

However, as we can not compute the left-hand-side directly, we replace it with its MC estimate $\delta J^{[i]}$:

$$\delta J(\boldsymbol{\theta})^{[i]} = \left(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right)^{\top} \boldsymbol{\delta\theta}^{[i]}.$$

Stacking up the equations for all $i = 1, 2, \dots, N$, we arrive at the over-determined system of linear equations,

$$\boldsymbol{\delta J} = \left(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right)^{\top} \boldsymbol{\delta\Theta}.$$

Solving this system by least squares yields (9.2). □

Likelihood-Ratio Trick

As the LSFD gradient still has high variance and is inexact. But we can do better: using the likelihood-ratio trick (subsection 2.2.5), we can compute the gradient of a MC estimate of $\mathcal{J}(\theta)$ exactly!

By the Markov property, the probability $p(\tau \mid \boldsymbol{\theta})$ of a trajectory τ can be decomposed into

$$p(\tau \mid \boldsymbol{\theta}) = \iota(s_0) \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)\, \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \tag{9.3}$$

where $\iota(s_0)$ is the initial state distribution. With this distribution, we can rewrite $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ as

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \int J(\tau)\, p(\tau \mid \theta)\, \mathrm{d}\tau = \int J(\tau)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}}\, p(\tau \mid \theta)\, \mathrm{d}\tau \tag{9.4}$$

where we can swap the limits under some mild conditions. By the log-ratio trick, we have $\boldsymbol{\nabla}_{\boldsymbol{\theta}} p(\tau \mid \boldsymbol{\theta}) = p(\tau \mid \boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})$ and (9.4) simplifies to an expectation over the gradient of the log-likelihood,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \int J(\tau)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}}\, p(\tau \mid \theta)\, \mathrm{d}\tau = \int p(\tau \mid \boldsymbol{\theta})\, J(\tau)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})\, \mathrm{d}\tau = \mathbb{E}_{\tau}\left[J(\tau)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})\right]. \tag{9.5}$$

And this expectation can be estimated by sampling! Hence, we just have to compute $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})$. With the decomposition (9.3), we can apply logarithm rules

$$\begin{aligned}
\log\left(\iota(s_0) \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)\, \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right) &= \log \iota(s_0) + \sum_{t=0}^{T-1} \log P(s_{t+1} \mid s_t, a_t) + \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \\
&= \sum_{t=0}^{T-1} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) + \mathrm{const}_{\boldsymbol{\theta}}
\end{aligned}$$

and everything except for the log-policy (which we model explicitly) is constant w.r.t. $\boldsymbol{\theta}$! Hence, we can compute the gradient in closed form if we choose π wisely and have

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta}) = \sum_{t=0}^{T-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t). \tag{9.6}$$

This is the simplest likelihood-ratio estimator we can imagine.

9.1.2 REINFORCE and Baselines

We already saw the simple policy gradient (9.6). If we combine this with the MC estimate of (9.5), we arrive at the *REINFORCE* gradient estimator:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{RF}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=0}^{T_i-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\big(a_t^{[i]} \mid s_t^{[i]}\big)\right) J\big(\tau^{[i]}\big)\right].$$

While this estimator is nice, simple, and *unbiased,* but it has very high variance! In fact, a specific estimate is pretty useless and causes divergence pretty quickly. However, we can introduce a *baseline* to reduce the variance. Let $\boldsymbol{b}$ be the baseline with as many entries as the gradient (i.e., the vectors are of same length). Then we have the *REINFORCE* policy gradient with baseline,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{RF}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=0}^{T_i-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right) \odot \Big(J(\tau)\mathbf{1} - \boldsymbol{b}\Big)\right], \tag{9.7}$$

where the $\odot$ denotes element-wise multiplication and $\mathbf{1}$ is the one-vector. While this baseline does not affect the bias as its expectation vanishes,

$$\mathbb{E}_\tau\big[\boldsymbol{b} \odot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})\big] = \int p(\tau \mid \boldsymbol{\theta}) \boldsymbol{b} \odot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta}) \,\mathrm{d}\tau = \boldsymbol{b} \odot \int p(\tau \mid \boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta}) \,\mathrm{d}\tau = \boldsymbol{b} \odot \mathbf{0} = \mathbf{0},$$

it still reduces the variance. In each iteration, we can also find an optimal baseline.

Theorem 15 (REINFORCE Optimal Baseline)**.** *The optimal baseline for the REINFORCE gradient* (9.7) *is*

$$\boldsymbol{b}^{\mathrm{RF}} = \arg\min_{\boldsymbol{b}} \mathrm{Var}_\tau\Big[\overline{\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{RF}} \mathcal{J}(\boldsymbol{\theta})}\Big] = \frac{\mathbb{E}_\tau\big[\boldsymbol{v}_\tau^2 J(\tau)\big]}{\mathbb{E}_\tau\big[\boldsymbol{v}_\tau^2\big]} \tag{9.8}$$

with $\boldsymbol{v}_\tau \coloneqq \sum_{t=0}^{T-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$ *where division and exponentiation are element-wise and* $\overline{\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{RF}} \mathcal{J}(\boldsymbol{\theta})}$ *refers to the "MC target" of* (9.7).

Proof. The variance of the estimator decomposes as

$$\operatorname{Var}_\tau\left[\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right] = \mathbb{E}_\tau\left[\left(\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right)^2\right] - \left(\mathbb{E}_\tau\left[\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right]\right)^2$$

where the second term is invariant w.r.t. $\boldsymbol b$ by design (the baseline does not effect the expectation of the estimator and therefore it is unbiased for every baseline). Hence, the gradient w.r.t. $\boldsymbol b$ reduces to

$$\nabla_{\boldsymbol b}\operatorname{Var}_\tau\left[\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right] = \mathbb{E}_\tau\left[\nabla_{\boldsymbol b}\left(\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right)^2\right] \propto \mathbb{E}_\tau\left[\left(\nabla_{\boldsymbol b}\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right)\odot\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}\right] \tag{9.9}$$

where we swapped the limits[2] and used the chain rule. We can now take the derivative of $\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)}$ w.r.t. $\boldsymbol b$:

$$\nabla_{\boldsymbol b}\overline{\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)} = \nabla_{\boldsymbol b}\left[\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)\odot\Big(J(\tau)\mathbf{1}-\boldsymbol b\Big)\right] = \sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t).$$

Plugging this result back into (9.9) yields

$$\begin{aligned}
\nabla_{\boldsymbol b}\operatorname{Var}_\tau\left[\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)\right] &\propto \mathbb{E}_\tau\left[\left(\nabla_{\boldsymbol b}\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)\right)\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)\right] = \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)\odot\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)\right]\\
&= \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)\odot\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)\odot\Big(J(\tau)\mathbf{1}-\boldsymbol b\Big)\right]\\
&= \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)^2 J(\tau)\right] - \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}(a_t\mid s_t)\right)^2\odot\boldsymbol b\right] \overset{!}{=} \mathbf{0}
\end{aligned}$$

using the linearity of the expectation. Rearranging the terms then yields (9.8) as the optimal baseline. □

The whole process for calculating the gradient is summarized in algorithm 16 including the MC estimation of the optimal baseline.

Algorithm 16: REINFORCE Gradient Estimation with Optimal Baseline

```
    Input: transition dataset $\mathcal{D}$, differentiable policy $\pi_{\boldsymbol\theta}$ with parameters $\boldsymbol\theta$
    Output: policy gradient $\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)$
1   $\boldsymbol v^{[i]} \leftarrow \sum_{t=0}^{T_i-1}\nabla_{\boldsymbol\theta}\log\pi_{\boldsymbol\theta}\big(a_t^{[i]}\mid s_t^{[i]}\big)$
2   $\boldsymbol b \leftarrow \frac{\sum_{i=1}^N\big(\boldsymbol v^{[i]}\big)^2 J^{[i]}}{\sum_{i=1}^N\big(\boldsymbol v^{[i]}\big)^2}$        // compute the optimal baseline
3   $\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta) \leftarrow \frac{1}{N}\sum_{i=1}^N \boldsymbol v^{[i]}\odot\big(J\big(\tau^{[i]}\big)\mathbf{1}-\boldsymbol b\big)$    // estimate the policy gradient
4   return $\nabla_{\boldsymbol\theta}^{\mathrm{RF}}\mathcal{J}(\boldsymbol\theta)$
```

[2]Take a course on measure theory if you want mathematical rigor!

Example

9.1.3 GPOMDP

Even though a baseline reduces the variance of the REINFORCE gradient estimate, it still has fairly high variance as the returns are extremely noisy (remember from the MC methods that they are composed of a lot of random variables). We can further reduce the variance by not considering the total return $J(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$ but instead observing that *returns from the past do not depend on actions in the future,* i.e.,

$$\mathbb{E}\big[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) r_{k+1}\big] = 0 \quad \forall k \le t.$$

Plugging this result into (9.7) with a time-dependent baseline yields the *GPOMDP*[3] *gradient*

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{GP}} \mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}_\tau\left[\sum_{k=0}^{T-1}\sum_{t=0}^{k} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \odot \left(\gamma^k r_{k+1}\mathbf{1} - \boldsymbol{b}_k\right)\right]$$

with its optimal baseline

$$\boldsymbol{b}_k = \frac{\mathbb{E}_\tau\left[\boldsymbol{v}_{\tau,k}^2 \gamma^k r_{k+1}\right]}{\mathbb{E}_\tau\left[\boldsymbol{v}_{\tau,k}^2\right]} \qquad\qquad \boldsymbol{v}_{\tau,k} = \sum_{t=0}^{k} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t).$$

The derivation of this baseline is analogous to the REINFORCE case. The approach is summarized in algorithm 17.

Algorithm 17: GPOMDP

```
Input: transition dataset $\mathcal{D}$, differential policy $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$
Output: policy gradient $\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{GP}} \mathcal{J}(\boldsymbol{\theta})$
1 $\boldsymbol{v}_k^{[i]} \leftarrow \sum_{t=0}^{k} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\big(a_t^{[i]} \mid s_t^{[i]}\big)$
2 $\boldsymbol{b}_k^{[i]} \leftarrow \frac{\sum_{i=1}^N \big(\boldsymbol{v}_k^{[i]}\big)^2 \gamma^k r_{k+1}^{[i]}}{\sum_{i=1}^N \big(\boldsymbol{v}_k^{[i]}\big)^2}$   // compute the optimal baseline
3 $\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{GP}} \mathcal{J}(\boldsymbol{\theta}) \leftarrow \frac{1}{N}\sum_{i=1}^{N}\sum_{k=0}^{T-1} \boldsymbol{v}_k^{[i]} \odot \big(\gamma^k r_{k+1}^{[i]}\mathbf{1} - \boldsymbol{b}_k\big)$   // estimate the policy gradient
4 return $\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{GP}} \mathcal{J}(\boldsymbol{\theta})$
```

9.2 Natural Policy Gradient

So far, the gradient methods assumes that the optimization space is Euclidean. However, as we only modify the parameters of a probability distribution, distances between two parameter vectors are not preserved! That is, measuring the distance between parameters gives us close to no information about the distribution's distance. One idea is to use the Kullback-Leibler (KL) divergence of two distributions, but the KL is not symmetric and

[3]"GPOMDP" stands for "gradient of partially observable MDP (POMDP)" as this approach was first applied for POMDPs.

thus not a metric and therefore difficult to use in optimization problems (also, it is hard to compute). Instead we can use the *Fisher information matrix (FIM)*

$$\boldsymbol{F}_{\boldsymbol{\theta}} \doteq \mathrm{Var}_{\tau}\big[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)\big] = \mathrm{Var}_{\tau}\left[\sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\right]$$

as a second-order approximation of the KL divergence for small perturbations $\boldsymbol{\delta\theta}$ of the parameters, i.e.,

$$\mathrm{KL}\big(p_{\boldsymbol{\theta}+\boldsymbol{\delta\theta}}(\tau) \,\|\, p_{\boldsymbol{\theta}}(\tau)\big) \approx \boldsymbol{\delta\theta}^{\top} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\delta\theta}.$$

The core idea of the *natural* PG is to update the policy under a KL constraint ε. For a given "vanilla" update $\boldsymbol{\delta\theta}^{\mathrm{VG}}$, we find the natural gradient update $\boldsymbol{\delta\theta}^{\mathrm{NG}}$ by solving the following optimization problem:

$$\begin{aligned} \boldsymbol{\delta\theta}^{\mathrm{NG}} &= \arg\max_{\boldsymbol{\delta\theta}} \; \boldsymbol{\delta\theta}^{\top} \boldsymbol{\delta\theta}^{\mathrm{VG}} \\ \text{s.t.} \quad & \boldsymbol{\delta\theta}^{\top} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\delta\theta} \le \varepsilon \end{aligned} \tag{9.10}$$

Using Lagrangian multipliers, we find the solution of this optimization problem to be

$$\boldsymbol{\delta\theta}^{\mathrm{NG}} \propto \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}$$

with the optimal learning rate $\alpha(\varepsilon) = \sqrt{\varepsilon / \big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}$.

Proof. We first reformulate the inequality constraint as $\epsilon - \boldsymbol{\delta\theta}^{\top} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\delta\theta} \ge 0$. Then the Lagrangian of (9.10) is

$$\mathcal{L} = -\boldsymbol{\delta\theta}^{\top} \boldsymbol{\delta\theta}^{\mathrm{VG}} - \mu\big(\varepsilon - \boldsymbol{\delta\theta}^{\top} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\delta\theta}\big) \tag{9.11}$$

with the Lagrangian multiplier μ and a flipped sign in the objective to apply the Karush-Kuhn-Tucker (KKT) conditions for a minimization problem. Setting the gradient w.r.t. $\boldsymbol{\delta\theta}$ to zero then yields the optimal search direction:

$$\boldsymbol{\nabla}_{\boldsymbol{\delta\theta}} \mathcal{L} = -\boldsymbol{\delta\theta}^{\mathrm{VG}} + 2\mu \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\delta\theta} \overset{!}{=} 0 \quad \Longrightarrow \quad \boldsymbol{\delta\theta} = \frac{1}{2\mu} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}. \tag{9.12}$$

By plugging this result back into (9.11), we get the dual

$$\begin{aligned} \mathcal{G} &= -\frac{1}{2\mu}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}} - \mu\varepsilon + \frac{1}{4\mu}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}} \\ &= -\frac{1}{2\mu}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}} - \mu\varepsilon + \frac{1}{4\mu}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}} \\ &= -\mu\varepsilon - \frac{1}{4\mu}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}. \end{aligned}$$

Now compute the derivative of the dual w.r.t. the Lagrangian multiplier and set it to zero:

$$\frac{\partial \mathcal{G}}{\partial \mu} = -\varepsilon + \frac{1}{4\mu^2}\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}} \overset{!}{=} 0 \quad \Longrightarrow \quad \mu = \sqrt{\frac{\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}{4\varepsilon}}.$$

Plugging this result back into (9.12), we get the new search direction,

$$\boldsymbol{\delta\theta} = \frac{\boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}{2\sqrt{\frac{(\boldsymbol{\delta\theta}^{\mathrm{VG}})^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}{4\varepsilon}}} = \underbrace{\sqrt{\frac{\varepsilon}{\big(\boldsymbol{\delta\theta}^{\mathrm{VG}}\big)^{\top} \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}}}_{\alpha(\varepsilon)} \underbrace{\boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\delta\theta}^{\mathrm{VG}}}_{\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{NG}} \mathcal{J}(\boldsymbol{\theta})},$$

where $\alpha(\varepsilon)$ is the learning rate and $\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{NG}} \mathcal{J}(\boldsymbol{\theta})$ is the natural PG. □

With this approach, we find policy updates that are *independent* of the policy's parametrization and we actually take steps in the policy space rather than parameter space. Note, however, that the approximation using the FIM is only valid for small perturbations! Also note that the natural gradient (NG) is not a completely new approach, but rather an extension that can be used to improve the previous methods.

9.3 The Policy Gradient Theorem

Definition 26 (Occupancy Measure)**.** Let $\rho^\pi(s)$ be the *occupancy measure* under policy π,

$$\rho^\pi(s) \coloneqq \sum_{t=0}^{\infty} \gamma^t P(s_t = s \,|\, \pi),$$

where the discount factor γ can be viewed as the *termination probability.*

Remark 7. *The occupancy measure is not a proper probability distribution as it is not normalized:*

$$\int_{\mathcal{S}} \rho^\pi(s)\,\mathrm{d}s = \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t P(s_t = s \,|\, \pi)\,\mathrm{d}s \overset{(4)}{=} \sum_{t=0}^{\infty} \gamma^t \int_{\mathcal{S}} P(s_t = s \,|\, \pi)\,\mathrm{d}s = \sum_{t=0}^{\infty} \gamma^t = \frac{1}{1-\gamma} \neq 1$$

Rather, it can be seen as the (expected) number of visits of a state s within a trajectory.

Definition 27 (Discounted State Distribution)**.** The *discounted state distribution*

$$d^\pi(s) \coloneqq (1-\gamma)\rho^\pi(s)$$

is the normalized occupancy measure and thus a proper probability distribution.

Theorem 16 (Policy Gradient Theorem)**.** *For any MDP we can compute the PG as*

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \int_{\mathcal{S}} \rho^{\pi_{\boldsymbol{\theta}}}(s) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \,|\, s)\big) Q^{\pi_{\boldsymbol{\theta}}}(s,a)\,\mathrm{d}a\,\mathrm{d}s$$

using the occupancy measure and action-value function.

Proof. We first write the objective $\mathcal{J}(\boldsymbol{\theta})$ in terms of the action-value function,

$$\mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}_\tau\big[J(\tau)\big] = \mathbb{E}_{s_0\sim\iota}\big[V^{\pi_{\boldsymbol{\theta}}}(s_0)\big] = \mathbb{E}_{s_0\sim\iota, a_0\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s_0)}\big[Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big], \tag{9.13}$$

and can therefore compute the gradient as

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{s_0\sim\iota, a_0\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s_0)}\big[Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big] \qquad (9.14)\\
&= \nabla_{\boldsymbol{\theta}} \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0)\pi(a_0 \,|\, s_0)\, Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&\overset{(*)}{=} \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0) \nabla_{\boldsymbol{\theta}} \big[\pi(a_0 \,|\, s_0)\, Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big]\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&= \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0)\Big[\big(\nabla_{\boldsymbol{\theta}}\pi(a_0 \,|\, s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0) + \pi(a_0 \,|\, s_0)\big(\nabla_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big)\Big]\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&= \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0)\big(\nabla_{\boldsymbol{\theta}}\pi(a_0 \,|\, s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\,\mathrm{d}a_0\,\mathrm{d}s_0 + \underbrace{\int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0)\pi(a_0 \,|\, s_0)\big(\nabla_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big)\,\mathrm{d}a_0\,\mathrm{d}s_0}_{(\#)}
\end{aligned}$$

[4] Again, rigor is for measure theory.

where we can exchange the limits in $(*)$ due to an application of some theorem from measure theory and in the last step we used the linearity of integration. We will now focus in the derivative of the action-value function w.r.t. the policy parameters. Using the Bellman expectation equation (3.3), we can write $\boldsymbol{\nabla}_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}$ in terms of the value function $V^{\pi_{\boldsymbol{\theta}}}$ gradient:

$$\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0) &= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Big[R(s_0,a_0) + \gamma \mathbb{E}_{s_1}\big[V^{\pi_{\boldsymbol{\theta}}}(s_1) \,|\, s_0,a_0\big] \Big] \\
&= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Big[R(s_0,a_0) + \gamma \int_{\mathcal{S}} P(s_1 \,|\, s_0,a_0) V^{\pi_{\boldsymbol{\theta}}}(s_1)\,\mathrm{d}s_1 \Big] \\
&= \gamma \int_{\mathcal{S}} P(s_1 \,|\, s_0,a_0)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\,\mathrm{d}s_1\,. \qquad (9.15)
\end{aligned}$$

Analogous to (9.13) and (9.14), we can decompose the state-value function:

$$\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1) &\overset{(*)}{=} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \int_{\mathcal{A}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\, Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1)\,\mathrm{d}a_1 \\
&\overset{(\dagger)}{=} \int_{\mathcal{A}} \big[\big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1) + \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1)\big) \big]\,\mathrm{d}a_1 \\
&\overset{(\ddagger)}{=} \int_{\mathcal{A}} \Big[\big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1) + \gamma \int_{\mathcal{S}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1) P(s_2 \,|\, s_1,a_1) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_2)\big)\,\mathrm{d}s_2 \Big]\,\mathrm{d}a_1 \\
&= \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1)\,\mathrm{d}a_1 + \gamma \int_{\mathcal{A}} \int_{\mathcal{S}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1) P(s_2 \,|\, s_1,a_1) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_2)\big)\,\mathrm{d}s_2\,\mathrm{d}a_1\,,
\end{aligned}$$

where in step $(*)$ we rewrote the action-value function, in step $(\dagger)$ we pushed the gradient into the integral, and in step $(\ddagger)$ we applied the Bellman expectation equation (3.3) analogous to (9.15). We can now plug these results into $(\#)$:

$$\begin{aligned}
(\#) &= \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0) \pi(a_0 \,|\, s_0) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} Q^{\pi_{\boldsymbol{\theta}}}(s_0,a_0)\big)\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&= \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0) \pi(a_0 \,|\, s_0) \left(\gamma \int_{\mathcal{S}} P(s_1 \,|\, s_0,a_0)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\,\mathrm{d}s_1 \right) \mathrm{d}a_0\,\mathrm{d}s_0 \\
&= \gamma \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi(a_0 \,|\, s_0) P(s_1 \,|\, s_0,a_0)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\,\mathrm{d}s_1\,\mathrm{d}a_0\,\mathrm{d}s_0 \qquad (9.16) \\
&= \gamma \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi(a_0 \,|\, s_0) P(s_1 \,|\, s_0,a_0) \Bigg(\int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1)\,\mathrm{d}a_1 \\
&\qquad + \gamma \int_{\mathcal{A}} \int_{\mathcal{S}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1) P(s_2 \,|\, s_1,a_1) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_2)\big)\,\mathrm{d}s_2\,\mathrm{d}a_1 \Bigg)\,\mathrm{d}s_1\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&= \gamma \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi(a_0 \,|\, s_0) P(s_1 \,|\, s_0,a_0) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1,a_1)\,\mathrm{d}a_1\,\mathrm{d}s_1\,\mathrm{d}a_0\,\mathrm{d}s_0 \\
&\qquad + \gamma^2 \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi(a_0 \,|\, s_0) P(s_1 \,|\, s_0,a_0) \pi_{\boldsymbol{\theta}}(a_1 \,|\, s_1) \\
&\qquad\qquad P(s_2 \,|\, s_1,a_1) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_2)\big)\,\mathrm{d}s_2\,\mathrm{d}a_1\,\mathrm{d}s_1\,\mathrm{d}a_0\,\mathrm{d}s_0\,.
\end{aligned}$$

We can now continue inserting $\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}$ an infinite number of times and with the abbreviation

$$\begin{aligned}
P(s = s_0 \,|\, \pi_{\boldsymbol{\theta}}) &= \iota(s_0) \\
P(s = s_1 \,|\, \pi_{\boldsymbol{\theta}}) &= \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi_{\boldsymbol{\theta}}(a_0 \,|\, s_0) P(s_1 \,|\, s_0, a_0) \,\mathrm{d}s_0 \,\mathrm{d}a_0 \\
&\vdots \\
P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) &= \int_{\mathcal{A}^t} \int_{\mathcal{S}^t} \iota(s_0) \prod_{k=0}^{t-1} \pi_{\boldsymbol{\theta}}(a_k \,|\, s_k) P(s_{k+1} \,|\, s_k, a_k) \,\mathrm{d}s_k \,\mathrm{d}a_k
\end{aligned} \tag{9.17}$$

where $\mathcal{A}^t$ and $\mathcal{S}^t$ are the t-times Cartesian product, we can write the policy gradient as

$$\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) &= \int_{\mathcal{S}} \int_{\mathcal{A}} \iota(s_0) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi(a_0 \,|\, s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0, a_0) \,\mathrm{d}a_0 \,\mathrm{d}s_0 + (\#) \\
&= \int_{\mathcal{S}} P(s = s_0 \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi(a_0 \,|\, s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0, a_0) \,\mathrm{d}a_0 \,\mathrm{d}s_0 \\
&\qquad + \gamma \int_{\mathcal{S}} P(s = s_1 \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi(a_1 \,|\, s_1)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_1, a_1) \,\mathrm{d}a_1 \,\mathrm{d}s_1 \\
&\qquad + \gamma^2 \int_{\mathcal{S}} P(s = s_2 \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi(a_2 \,|\, s_2)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_2, a_2) \,\mathrm{d}a_2 \,\mathrm{d}s_2 \\
&\qquad + \gamma^3 \int_{\mathcal{S}} P(s = s_3 \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi(a_3 \,|\, s_3)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_3, a_3) \,\mathrm{d}a_3 \,\mathrm{d}s_3 \\
&\qquad + \cdots \\
&= \sum_{t=0}^{\infty} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \,\mathrm{d}a_t \,\mathrm{d}s_t \,.
\end{aligned}$$

Note that we have to show this equivalence. First, let

$$\begin{aligned}
\big\{\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\big\}_n &:= \sum_{t=0}^{n} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \,\mathrm{d}a_t \,\mathrm{d}s_t \\
&\qquad \underbrace{+ \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \,|\, \pi_{\boldsymbol{\theta}}) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+1})\big) \,\mathrm{d}s_{n+1}}_{(\%)}
\end{aligned} \tag{9.18}$$

be the policy gradient we get after when terminating after n expansion. We want to show that $\big\{\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\big\}_n =$

$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ for all $n \in \mathbb{N}_0$. We show the equivalence by induction. For $n = 0$, we have

$$\begin{aligned}
\{\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\}_0 = \sum_{t=0}^{0} \gamma^t \int_{\mathcal{S}} P(s = s_t \mid \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \, \mathrm{d}a_t \, \mathrm{d}s_t \\
+ \gamma \int_{\mathcal{S}} P(s = s_1 \mid \pi_{\boldsymbol{\theta}}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\big) \, \mathrm{d}s_1 \\
= \int_{\mathcal{S}} P(s = s_0 \mid \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0, a_0) \, \mathrm{d}a_0 \, \mathrm{d}s_0 \\
+ \gamma \int_{\mathcal{S}} P(s = s_1 \mid \pi_{\boldsymbol{\theta}}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\big) \, \mathrm{d}s_1 \\
= \int_{\mathcal{S}} \iota(s_0) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \mid s_0)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_0, a_0) \, \mathrm{d}a_0 \, \mathrm{d}s_0 \\
+ \gamma \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} \iota(s_0) \pi_{\boldsymbol{\theta}}(a_0 \mid s_0) P(s_1 \mid s_0, a_0) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_1)\big) \, \mathrm{d}s_0 \, \mathrm{d}a_0 \, \mathrm{d}s_1
\end{aligned}$$

which exactly equals (9.16) combined with (9.14). This finishes the induction base. We now have to prove the induction step $n \to n+1$. Assume that the equivalence holds for some n. We can then explicitly expand the last term of the sum, $(\%)$ in (9.18), obtaining

$$\begin{aligned}
(\%) &= \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \mid \pi_{\boldsymbol{\theta}}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+1})\big) \, \mathrm{d}s_{n+1} \\
&= \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \mid \pi_{\boldsymbol{\theta}}) \Bigg(\int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_{n+1} \mid s_{n+1})\big) Q^{\pi_{\boldsymbol{\theta}}}(s_{n+1}, a_{n+1}) \, \mathrm{d}a_{n+1} \\
&\qquad + \gamma \int_{\mathcal{A}} \int_{\mathcal{S}} \pi_{\boldsymbol{\theta}}(a_{n+1} \mid s_{n+1}) P(s_{n+2} \mid s_{n+1}, a_{n+1}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+2})\big) \, \mathrm{d}s_{n+2} \, \mathrm{d}a_{n+1} \Bigg) \, \mathrm{d}s_{n+1} \\
&= \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \mid \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_{n+1} \mid s_{n+1})\big) Q^{\pi_{\boldsymbol{\theta}}}(s_{n+1}, a_{n+1}) \, \mathrm{d}a_{n+1} \, \mathrm{d}s_{n+1} \\
&\qquad + \gamma^{n+2} \int_{\mathcal{S}} \int_{\mathcal{A}} \int_{\mathcal{S}} P(s = s_{n+1} \mid \pi_{\boldsymbol{\theta}}) \, \pi_{\boldsymbol{\theta}}(a_{n+1} \mid s_{n+1}) \\
&\qquad\qquad P(s_{n+2} \mid s_{n+1}, a_{n+1}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+2})\big) \, \mathrm{d}s_{n+2} \, \mathrm{d}a_{n+1} \, \mathrm{d}s_{n+1} \\
&= \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \mid \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_{n+1} \mid s_{n+1})\big) Q^{\pi_{\boldsymbol{\theta}}}(s_{n+1}, a_{n+1}) \, \mathrm{d}a_{n+1} \, \mathrm{d}s_{n+1} \\
&\qquad + \gamma^{n+2} \int_{\mathcal{S}} P(s = s_{n+2} \mid \pi_{\boldsymbol{\theta}}) \big(\nabla_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+2})\big) \, \mathrm{d}s_{n+2}
\end{aligned}$$

where we split the integral by linearity and then applied (9.17). Plugging this result back into (9.18), we obtain the induction step

$$
\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) &= \left\{\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right\}_n \\
&= \sum_{t=0}^{n} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t \\
&\qquad + \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \,|\, \pi_{\boldsymbol{\theta}}) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+1})\big)\, \mathrm{d}s_{n+1} \\
&= \sum_{t=0}^{n} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t \\
&\qquad + \gamma^{n+1} \int_{\mathcal{S}} P(s = s_{n+1} \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_{n+1} \,|\, s_{n+1})\big) Q^{\pi_{\boldsymbol{\theta}}}(s_{n+1}, a_{n+1})\, \mathrm{d}a_{n+1}\, \mathrm{d}s_{n+1} \\
&\qquad + \gamma^{n+2} \int_{\mathcal{S}} P(s = s_{n+2} \,|\, \pi_{\boldsymbol{\theta}}) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{n+2})\big)\, \mathrm{d}s_{n+2} \\
&= \sum_{t=0}^{n+1} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t \\
&\qquad + \gamma^{(n+1)+1} \int_{\mathcal{S}} P(s = s_{(n+1)+1} \,|\, \pi_{\boldsymbol{\theta}}) \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} V^{\pi_{\boldsymbol{\theta}}}(s_{(n+1)+1})\big)\, \mathrm{d}s_{(n+1)+1} \\
&= \left\{\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right\}_{n+1}.
\end{aligned}
$$

Hence, $\left\{\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})\right\}_n = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ holds for all $n \in \mathbb{N}_0$.

In the limit $n \to \infty$, we can therefore drop the correction term $(\%)$ and obtain

$$
\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \sum_{t=0}^{\infty} \gamma^t \int_{\mathcal{S}} P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t\,.
$$

By swapping limits, plugging in the occupancy measure and relabeling $s_t \mapsto s$ and $a_t \mapsto a$, we obtain

$$
\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) &= \int_{\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t P(s = s_t \,|\, \pi_{\boldsymbol{\theta}}) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t \\
&= \int_{\mathcal{S}} \rho^{\pi_{\boldsymbol{\theta}}}(s_t) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t \,|\, s_t)\big) Q^{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\, \mathrm{d}a_t\, \mathrm{d}s_t \\
&= \int_{\mathcal{S}} \rho^{\pi_{\boldsymbol{\theta}}}(s) \int_{\mathcal{A}} \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \,|\, s)\big) Q^{\pi_{\boldsymbol{\theta}}}(s, a)\, \mathrm{d}a\, \mathrm{d}s\,.
\end{aligned}
$$

This is exactly the result of the policy gradient theorem. □

Remark 8. *Using the discounted state distribution $d^{\pi_{\boldsymbol{\theta}}}(s)$ and the log-ratio trick, we can rewrite the PG as*

$$
\begin{aligned}
\boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) &= \frac{1}{1-\gamma} \int_{\mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \int_{\mathcal{A}} Q^{\pi_{\boldsymbol{\theta}}}(s, a)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \,|\, s)\, \mathrm{d}a\, \mathrm{d}s \\
&= \frac{1}{1-\gamma} \int_{\mathcal{S}} d^{\pi_{\boldsymbol{\theta}}}(s) \int_{\mathcal{A}} \pi_{\boldsymbol{\theta}}(a \,|\, s)\, Q^{\pi_{\boldsymbol{\theta}}}(s, a)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \,|\, s)\, \mathrm{d}a\, \mathrm{d}s \\
&= \frac{1}{1-\gamma} \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot|s)} \big[Q^{\pi_{\boldsymbol{\theta}}}(s, a)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \,|\, s)\big] \\
&\propto \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot|s)} \big[Q^{\pi_{\boldsymbol{\theta}}}(s, a)\, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \,|\, s)\big]
\end{aligned} \tag{9.19}
$$

and we can readily apply MC-based techniques if we know the true action-value function.

Note how nicely the PG theorem connects policy search and gradient estimation with value functions. However, we need the action-value function to compute the gradient. If we estimate it using MC methods, the calculation is equivalent to GPOMDP (subsection 9.1.3). If we estimate the value function using TD learning (which is much more robust than MC), we finally arrive at *actor-critic* methods which are the current state of the art (SOTA) in RL.

9.3.1 Compatible Function Approximation

While using the value function reduces the variance of the PG estimate, now the gradient is biased! A solution is to only use *compatible* approximators.

Definition 28 (Compatible Function Approximation)**.** A value function approximation $\hat{Q}_{\boldsymbol{\omega}}(s,a)$ is *compatible* to the policy $\pi_{\boldsymbol{\theta}}$ if $\nabla_{\boldsymbol{\omega}} \hat{Q}_{\boldsymbol{\omega}}(s,a) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

Theorem 17 (Compatible Function Approximation Theorem)**.** *If $\hat{Q}_{\boldsymbol{\omega}}$ is a compatible to $\pi_{\boldsymbol{\theta}}$ and the value function parameters $\boldsymbol{\omega}$ minimize the mean squared error (MSE)*

$$\boldsymbol{\omega} = \arg\min_{\boldsymbol{\omega}} \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right)^2 \right],$$

then the policy gradient estimate using $\hat{Q}_{\boldsymbol{\omega}}$ is unbiased.

Proof. Let ϵ be the MSE

$$\epsilon := \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right)^2 \right]$$

then its gradient w.r.t. $\boldsymbol{\omega}$ is

$$\begin{aligned} \nabla_{\boldsymbol{\omega}} \epsilon &= 2\mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right) \nabla_{\boldsymbol{\omega}} \left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right) \right] \\ &= 2\mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right) \nabla_{\boldsymbol{\omega}} \hat{Q}_{\boldsymbol{\omega}}(s,a) \right] \\ &= 2\mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \hat{Q}_{\boldsymbol{\omega}}(s,a) \right) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right] \end{aligned}$$

where the last equality is due to the assumption of $\hat{Q}_{\boldsymbol{\omega}}$ being compatible to $\pi_{\boldsymbol{\theta}}$. As $\boldsymbol{\omega}$ minimizes ϵ, $\nabla_{\boldsymbol{\omega}} \epsilon = \mathbf{0}$ holds. Hence,

$$\mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[Q^{\pi_{\boldsymbol{\theta}}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right] = \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\hat{Q}_{\boldsymbol{\omega}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right],$$

and the gradient estimate is unbiased. □

We can write the compatible function approximation (CFA) as a linear combination of the policy log-gradient,

$$\hat{Q}_{\boldsymbol{\omega}}(s,a) = \left(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right)^{\top} \boldsymbol{\omega}.$$

Plugging the CFA into (9.19) yields

$$\begin{aligned} \nabla_{\boldsymbol{\theta}}^{\mathrm{CFA}} \mathcal{J}(\boldsymbol{\theta}) &= \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[Q^{\pi_{\boldsymbol{\theta}}}(s,a)\, \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right] \\ &= \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right)^{\top} \boldsymbol{\omega} \left(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right) \right] \\ &= \mathbb{E}_{s \sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s)} \left[\left(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right) \left(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) \right)^{\top} \right] \boldsymbol{\omega} = \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\omega} \end{aligned}$$

where the expectation is equal to the FIM as the second component of the variance, the expectation-square, vanishes as the expectation over $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)$ is zero (see (2.1)).

9.3.2 Episodic Natural Actor-Critic

Definition 29 (Advantage Function)**.** The *advantage function* for a policy π is

$$A^{\pi}(s,a) \coloneqq Q^{\pi}(s,a) - V^{\pi}(s)$$

and measures how good an action a is compared to the policy's behavior.

Obviously, the advantage function has zero mean w.r.t. to the policy π:

$$\mathbb{E}_{a\sim\pi}\big[A^{\pi}(s,a)\big] = \mathbb{E}_{a\sim\pi}\big[Q^{\pi}(s,a) - V^{\pi}(s)\big] = \mathbb{E}_{a\sim\pi}\big[Q^{\pi}(s,a)\big] - V^{\pi}(s) = V^{\pi}(s) - V^{\pi}(s) = 0$$

Episodic natural actor-critic (eNAC) combines the idea of CFA with NG but estimates the advantage function rather than the action-value function[5],

$$\hat{A}(s,a) = \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \,|\, a)\big)^{\top} \boldsymbol{\omega},$$

as it usually has lower variance. As the CFA and NG gradient are given by

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{CFA}} \mathcal{J}_{\boldsymbol{\theta}} = \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\omega} \qquad\qquad \boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{NG}} \mathcal{J}_{\boldsymbol{\theta}} = \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathcal{J}_{\boldsymbol{\theta}}.$$

Hence, the eNAC gradient is simply

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{eNAC}} \mathcal{J}_{\boldsymbol{\theta}} = \boldsymbol{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{F}_{\boldsymbol{\theta}} \boldsymbol{\omega} = \boldsymbol{\omega}.$$

However, we still have to compute $\boldsymbol{\omega}$! For this, we also need an approximation of the value function for the initial state. If the initial state does not change, this is just a constant and if it does, we choose a linear approximation $\hat{V}_{\boldsymbol{v}}(s) = \boldsymbol{\phi}^{\top}(s)\boldsymbol{v}$ using some features $\boldsymbol{\phi}(s)$ of the state s. Now we can write

$$J(\tau) = \hat{V}_{\boldsymbol{v}}(s_0) + \sum_{t=0}^{T-1} \gamma^t \hat{A}_{\boldsymbol{\omega}}(s_t, a_t) \tag{9.20}$$

which is a linear equation in terms of $\boldsymbol{v}$ and $\boldsymbol{\omega}$ as we know $J(\tau)$ from sampling. For a set of trajectories $\{\tau^{[i]}\}_{i=1}^{N}$, we can formulate (9.20) as a matrix-vector linear equation

$$\boldsymbol{J} = \boldsymbol{\Psi} \begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix}$$

with

$$\boldsymbol{J} = \begin{bmatrix} J(\tau^{[1]}) \\ J(\tau^{[2]}) \\ \vdots \\ J(\tau^{[N]}) \end{bmatrix} \qquad \boldsymbol{\Psi} = \begin{bmatrix} (\boldsymbol{\psi}^{[1]})^{\top} \\ (\boldsymbol{\psi}^{[2]})^{\top} \\ \vdots \\ (\boldsymbol{\psi}^{[N]})^{\top} \end{bmatrix} \qquad \boldsymbol{\psi}^{[i]} = \begin{bmatrix} \sum_{t=0}^{T_i-1} \gamma^t \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\big(a_t^{[i]} \,|\, s_t^{[i]}\big) \\ \boldsymbol{\phi}\big(s_0^{[i]}\big) \end{bmatrix}.$$

As this system is linear and overdetermined, we can evaluate $\boldsymbol{\omega}$ and $\boldsymbol{v}$ using least squares. Note that this is an *episodic* algorithm as the calculation of $J(\tau)$ requires a complete episode. algorithm 18 summarizes this approach.

[5]Note that this is still a CFA its expectation vanishes.

Algorithm 18: Episodic Natural Actor-Critic

```
Input: transition dataset $\mathcal{D}$, differential policy $\pi_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$
Output: policy gradient $\nabla_{\boldsymbol{\theta}}^{\mathrm{eNAC}} \mathcal{J}(\boldsymbol{\theta})$
// Compute the state features for the value function.
1 $\boldsymbol{\psi}^{[i]} \leftarrow \begin{bmatrix} \sum_{t=0}^{T_i-1} \gamma^t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\big(a_t^{[i]} \mid s_t^{[i]}\big) \\ \boldsymbol{\phi}\big(s_0^{[i]}\big) \end{bmatrix}$
// Fit the advantage and value function (for the initial state).
2 $\boldsymbol{J} \leftarrow \big[J\big(\tau^{[1]}\big) \quad J\big(\tau^{[2]}\big) \quad \cdots \quad J\big(\tau^{[N]}\big)\big]^\top$
3 $\boldsymbol{\Psi} \leftarrow \big[\big(\boldsymbol{\psi}^{[1]}\big) \quad \big(\boldsymbol{\psi}^{[2]}\big) \quad \cdots \quad \big(\boldsymbol{\psi}^{[N]}\big)\big]^\top$
4 $\begin{bmatrix} \boldsymbol{\omega} \\ \boldsymbol{v} \end{bmatrix} \leftarrow \big(\boldsymbol{\Psi}^\top \boldsymbol{\Psi}\big)^{-1} \boldsymbol{\Psi} \boldsymbol{J}$
5 return $\boldsymbol{\omega}$
```

9.4 Wrap-Up

- difference between value-based, policy search, and actor-critic methods
- importance of exploration in policy search
- Gaussian policies for continuous control
- approaches for policy gradient
  - finite difference gradient (black-box RL)
  - REINFORCE and the importance of the baseline
  - GPOMDP and the fact that rewards from the past do not depend on actions in the future
- Fisher information matrix and natural gradient
- policy gradient theorem and connection between value-based, policy search, and actor-critic
- compatible function approximation
- how eNAC combines CFA and NG
- Additional reading material:
  - Book: "Introduction to Reinforcement Learning" (Sutton and Barto, 2018), Chapters 13
  - Paper: "A Survey on Policy Search for Robotics" (Deisenroth, Neumann, and Peters, 2013), Chapters 1 to 2.4.1

10 Deep Value-Function Methods

So far, we only discussed RL methods relying on classical non-deep ML, even though it is theoretically possible to plug in a NN function approximator into, for instance, SARSA. However, as we will see in this chapter, employing NNs requires a bunch of tricks and hacks to make them working. While this might be annoying, it allows us to bring RL to high-dimensional problems and even learning from images! In this chapter we focus on value-based deep RL and the next chapter (chapter 11) covers actor-critic methods.

10.1 Deep Q-Learning: DQN

In deep Q-learning, we approximate the action-value function using a NN, i.e., $\hat{Q}_{\boldsymbol{\omega}}(s,a) \approx Q^{\pi}(s,a)$. Hence, this methods is also called the *deep Q-network (DQN)*. In *tabular* Q-learning (7.2), we just set the new action-value to a new estimate. Of course, this is not possible when the action-value function is a NN and we therefore instead minimize the expected squared TD error,

$$\mathcal{L}(\boldsymbol{\omega}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'\in\mathcal{A}} \hat{Q}_{\boldsymbol{\omega}}(s',a') - \hat{Q}_{\boldsymbol{\omega}}(s,a)\right)^2\right],$$

where the expectation is over a dataset $\mathcal{D}$ of trajectories. With this approach, we have a variety of problems:

1. loss contains bootstrapping, is off-policy, and of course works with function approximation; this is the *deadly triad* (section 8.3)
2. offline dataset $\mathcal{D}$ is assumed to be *unavailable* due to the problem's complexity; we can therefore not use offline algorithms, e.g., fitted Q-iteration
3. data has to be collected online, but training NNs is online RL can lead to *catastrophic forgetting*

Type	Name	Tackled Issue	Approach
Essential	Replay Buffer	distribution shift	reuse old transitions
	Target Network	instability	use a target network for the TD error
	Minibatch Updates	inefficiency	sample small minibatch
	Reward-Clipping	unstable optimization	clip the reward
Enhance.	Double DQN	overestimation	max. over target, evaluate with current
	Prioritized Replay Buffer	sample inefficiency	bias sampling towards large TD error
	Dueling DQN	recovering V and A	explicitly split NN output into V and A
	Noisy DQN	exploration	add noisy linear layers
	Distributional DQN	stochastic rewards	model return *distribution,* not *expectation*

Table 10.1: Essential Tricks and Enhancements for DQN

We now go over some established methods for tackling these problems and making DQN work in the first place. The complete deep Q-learning with all essential tricks (Table 10.1) employed is summarized in algorithm 19. While DQN is extremely powerful, it comes at high cost! Learning requires many samples, the algorithm is highly sensitive to hyperparameter tuning (e.g., the learning rate), and computation times are enormous.

Algorithm 19: Deep Q-Learning using Deep Q-Network

```
   Input: environment; differential approximator $\hat{Q}$ with parameters $\boldsymbol{\omega}$
   Output: estimated parameters $\boldsymbol{\omega}$
1  initialize replay buffer $\mathcal{D}$ to capacity $N$
2  initialize parameters $\boldsymbol{\omega}$ appropriately
3  initialize target parameters $\boldsymbol{\omega}' \leftarrow \boldsymbol{\omega}$
4  repeat
5      initialize $s$
6      while $s$ is not terminal do
           // Execute the policy.
7          take action $a \sim \epsilon$-greedy$(\hat{Q}_{\boldsymbol{\omega}})$, observe reward $r$ and next state $s'$
8          store transition $\langle s, a, r, s' \rangle$ in $\mathcal{D}$
           // Sample a minibatch of transitions from the replay buffer.
9          $\langle s_i, a_i, r_i, s'_i \rangle_{i=1}^{M} \sim \mathcal{D}$
           // Compute TD targets.
10         $y_i \leftarrow \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_{\boldsymbol{\omega}'}(s'_i, a') & \text{otherwise} \end{cases}$
           // Perform the optimization.
11         perform gradient descent step on $\big(y_i - \hat{Q}_{\boldsymbol{\omega}}(s_i, a_i)\big)^2$ w.r.t. $\boldsymbol{\omega}$
12         every $C$ steps, update target $\boldsymbol{\omega}' \leftarrow \boldsymbol{\omega}$
13 until convergence
14 return $\boldsymbol{\omega}$
```

10.1.1 Replay Buffer

Instead of using just the current data, we collect individual transitions in a *replay buffer* (with finite capacity). Due to off-policy updates, we can re-use transitions from the buffer. This reduces the negative impacts of the *distribution shift,* i.e., the change of the data distribution as our policy changes.

10.1.2 Target Network

Instead of directly updating the policy's NN in every step, we keep a copy $\boldsymbol{\omega}'$ of the policy network $\boldsymbol{\omega}$, the *target network,* for computing the TD error:

$$\mathcal{L}(\boldsymbol{\omega}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\big(r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_{\boldsymbol{\omega}'}(s', a') - \hat{Q}_{\boldsymbol{\omega}}(s, a)\big)^2\Big].$$

Periodically (e.g., every C steps), we copy the parameters $\boldsymbol{\omega} \rightarrow \boldsymbol{\omega}'$ to the target network. This avoids some instability induced by the function approximation as the TD target does not change as frequent as before.

10.1.3 Minibatch Updates

Instead of using all transitions stored in the replay buffer, we only use a subset (a *minibatch*) for computing the loss. This improves efficiency compared to training on all transitions. Also, random samples are closer to fulfilling the i.i.d.-property assumed by SGD compared to the temporally correlated trajectories.

10.1.4 Reward- and Target-Clipping

Instead of using the "real" reward, we clip the values between $[-1, 1]$. This also clips the TD error,

$$r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_{\boldsymbol{\omega}}(s', a') - \hat{Q}(s, a) \in [-1, 1],$$

at it turns out to be sufficient to use the *Huber loss* instead of the quadratic loss. This improves the stability of the optimization as values stay reasonably small.

10.1.5 Examples

10.2 DQN Enhancements

Even though DQN is already powerful as is, it still has some flaws we want to and are able to overcome. In this section we discuss some of these flaws and approaches to fixing them. All enhancements are also summarized in Table 10.1.

10.2.1 Overestimation and Double Deep Q-Learning

In DQN, we have the following loss:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}}\Big[\big(r + \gamma \max_{a' \in \mathcal{A}} \hat{Q}_{\boldsymbol{\omega}'}(s', a') - \hat{Q}_{\boldsymbol{\omega}}(s, a)\big)^2\Big].$$

However, it is known that the use of the maximum operator leads to overestimation of the action-value. This is not necessarily bad (as at least all state-action-pairs get overestimated), but can cause suboptimal performance in highly stochastic environments. We can fight this problem using the idea of *double Q-learning*. Instead of maximizing over $\hat{Q}_{\boldsymbol{\omega}'}(s', a')$, we find the action that maximizes the online policy $\hat{Q}_{\boldsymbol{\omega}}(s', a')$ and evaluate it using the target value function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}}\Big[\big(r + \gamma \hat{Q}_{\boldsymbol{\omega}'}\big(s', \arg\max_{a' \in \mathcal{A}} Q_{\boldsymbol{\omega}}(s', a')\big) - \hat{Q}_{\boldsymbol{\omega}}(s, a)\big)^2\Big].$$

Notice that the parameters of the action-value functions being used for maximization are different! We call this approach *double DQN (DDQN)*.

10.2.2 Prioritized Replay Buffer

The replay buffer might become extremely big, but it is not reasonable to assume that every transition conveys equal amounts of information! Instead, transitions resulting in a high TD error are more informative. An approach to tackle this is to weigh the samples using the TD error such that the probability of the i-th sample is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i = |\delta| + \epsilon$ with a small ϵ to avoid degenerate cases and a regularization α of the prioritization (where $\alpha = 0$ equals uniform sampling). However, altering the sampling like this introduces a bias in the loss estimate. Hence, we have to correct the bias using important sampling weights

$$w_i = \left(\frac{1}{NP(i)}\right)^{\beta}$$

where N is the total number of data points and β regulates the strength of the importance sampling (where $\beta = 1$ fully compensates the bias). For stability reasons, the weights are also normalized by $1/\max_i w_i$. In practice, the coefficient is usually annealed from $\beta_0 < 1$ to 1.

10.2.3 Dueling DQN

It may be helpful from time to time to recover the state-value and advantage from the action-value function. However, given a $Q = V + A$, we cannot recover the latter. The idea of *dueling DQN* is to split the output of the Q-network into two streams, $\hat{V}_{\boldsymbol{\omega},\boldsymbol{\beta}}$ and $\hat{A}_{\boldsymbol{\omega},\boldsymbol{\alpha}}$, i.e., a network with two regression heads parameterized by $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$, respectively. However, instead of just adding up the two outputs to get the action-value, dueling DQN uses

$$\hat{Q}_{\boldsymbol{\omega},\boldsymbol{\alpha},\boldsymbol{\beta}}(a,s) = V_{\boldsymbol{\omega},\boldsymbol{\beta}}(s) + \big(\hat{A}_{\boldsymbol{\omega},\boldsymbol{\alpha}}(s,a) - \max_{a' \in \mathcal{A}} \hat{A}_{\boldsymbol{\omega},\boldsymbol{\alpha}}(s,a')\big)$$

forcing $\max_{a\in\mathcal{A}} \hat{Q}_{\boldsymbol{\omega},\boldsymbol{\alpha},\boldsymbol{\beta}}(s,a) = V_{\boldsymbol{\omega},\boldsymbol{\beta}}(s)$.

10.2.4 Noisy DQN

Usually, exploration is performed directly on policy level using a ε-greedy policy. That is, we perturb the action given an action-value function. However, we can also achieve exploration by perturbing the action-value function directly by adding noise variables ε to the NN's layers. Optimization of the parameters is then w.r.t. the expected loss over the noise ε.

For instance, we can make a linear layer $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$ with $\boldsymbol{x} \in \mathbb{R}^p$, $\boldsymbol{y}, \boldsymbol{b} \in \mathbb{R}^q$, and $\boldsymbol{W} \in \mathbb{R}^{y \times p}$ noisy using

$$\tilde{\boldsymbol{y}} = \big(\boldsymbol{M}_W + \boldsymbol{\sigma}_W \odot \boldsymbol{\varepsilon}_W\big)\boldsymbol{x} + (\boldsymbol{\mu}_b + \boldsymbol{\sigma}_b \odot \boldsymbol{\varepsilon}_b)$$

where $\boldsymbol{M}_W \in \mathbb{R}^{q\times p}$, $\boldsymbol{\sigma}_W \in \mathbb{R}^{q\times p}$, $\boldsymbol{\mu}_b \in \mathbb{R}^q$, and $\boldsymbol{\Sigma}_b \in \mathbb{R}^q$ are learnable parameters and $\boldsymbol{\varepsilon}_W \sim \mathcal{N}(\boldsymbol{O}, \boldsymbol{I}) \in \mathbb{R}^{q\times p}$ and $\boldsymbol{\varepsilon}_b \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \in \mathbb{R}^q$ are noise variables. In practice, we only make the last layer noisy.

10.2.5 Distributional/Categorical DQN

So far, we have always estimated the action-value function directly. That is, we estimate the expected return starting from s with a:

$$Q^{\pi}(s,a) = \mathbb{E}[J_t \,|\, s_t = s, a_t = a].$$

The idea of *distributional DQN* is to not find a point estimate, but to model the whole return distribution directly[1]. For this, let $Z^{\pi}(s,a)$ be the *distributional* value function. To this end, we also consider the reward $R(s,a)$ to be a distribution over rewards. To reason about how the distributional value transitions, let P^{π} be the transition operator such that

$$(P^{\pi}Z^{\pi})(s,a) \overset{D}{=} Z^{\pi}(s',a')$$

[1]Note that we do not model the epistemic uncertainty of our model, but the intrinsic aleatoric uncertainty of the environment!

with $s' \sim P(\cdot \,|\, s, a)$ and $a' \sim \pi(\cdot \,|\, s')$. Also, we define the *distributional Bellman operator* T such that

$$(T^\pi Z^\pi)(s, a) \overset{D}{=} R(s, a) + \gamma P^\pi Z^\pi(s, a). \tag{10.1}$$

Notice that we have three sources of randomness here: the reward, the transition, and the next state-value distribution.

In practice, we approximate the distributional value function by discretizing the return space into K so-called *atoms*. These atoms $z_i := V_{\min} + i\Delta z$ with $\Delta z = (V_{\max} - V_{\min})/(N-1)$ are equidistant across the return space. For each atom z_i, its probability is given by

$$\rho_i(s, a) = \frac{\exp f_{\boldsymbol{\omega},i}(s, a)}{\sum_{j=1}^{K} \exp f_{\boldsymbol{\omega},j}(s, a)}$$

where $f_{\boldsymbol{\omega},i}(s, a)$ is the i-th entry of a function approximator $\boldsymbol{f}_{\boldsymbol{\omega}}$ which we want to learn. Hence, we can now write the categorical value distribution approximation $\hat{Z}_{\boldsymbol{\omega}}(s, a) \approx Z^\pi(s, a)$ as

$$\hat{Z}_{\boldsymbol{\omega}}(s, a) = \begin{bmatrix} \rho_1(s, a) & \rho_2(s, a) & \cdots & \rho_K(s, a) \end{bmatrix}^\top.$$

While we can now explicitly compute the update (10.1), we have a problem: the update shifts the support of our distribution such that $T^\pi \hat{Z}_{\boldsymbol{\omega}}$ and $\hat{Z}_{\boldsymbol{\omega}}$ have close to disjoint support. Hence, we apply a projection Φ to project the distribution back to the original support. The projection has the following effect (on the i-th component),

$$\big(\Phi T^\pi \hat{Z}_{\boldsymbol{\omega}}(s, a)\big)_i = \sum_{j=1}^{K} \operatorname{clip}_1^1 \left(1 - \frac{\big|\operatorname{clip}_{V_{\min}}^{V_{\max}}(T z_j) - z_i\big|}{\Delta z} \right) \rho_j\big(s', \pi(s')\big),$$

where we define the effect of the Bellman operator to an atom as $Tz = r + \gamma z$. Here, π is greedy w.r.t. $\mathbb{E}[\hat{Z}_{\boldsymbol{\omega}}]$ and $\langle s, a, r, s' \rangle$ are from a sampled transition. We also define the clipping operation

$$\operatorname{clip}_a^b : \mathbb{R} \to [a, b] : x \mapsto \max\{a, \min\{b, x\}\}$$

to clip any value to $[a, b]$. To find new parameters $\boldsymbol{\omega}_{\text{new}}$, we now minimize the KL divergence between the next and current value function distribution:

$$\boldsymbol{\omega}_{\text{new}} = \arg\min_{\boldsymbol{\omega}'} \operatorname{KL}\big(\Phi T Z_{\boldsymbol{\omega}'}(s, a) \,\big|\, Z_{\boldsymbol{\omega}}(s, a)\big).$$

Again, this is w.r.t. a sampled transition $\langle s, a, r, s' \rangle$. As we used a categorical approximation, this algorithm is also called *categorical DQN.* For more details on it, see the original paper "A Distributional Perspective on Reinforcement Learning" (Bellemare et al., 2017), especially sections 3.3 and 4.

10.2.6 Rainbow

Rainbow DQN just throws every of the previous methods,

- double and dueling DQN for fighting the estimation bias,
- prioritized replay buffer for sample-efficiency,
- noise DQN for exploration, and
- and distributional DQN for dealing with uncertain returns,

into one algorithm and uses them all. This is possible as luckily, all these methods are compatible. With a correct implementation, rainbow is extremely powerful, although hard to implement. Additionally, rainbow uses the n-step return to estimate the action-value function. See Figure 10.1 for a comparison of rainbow to each method alone.

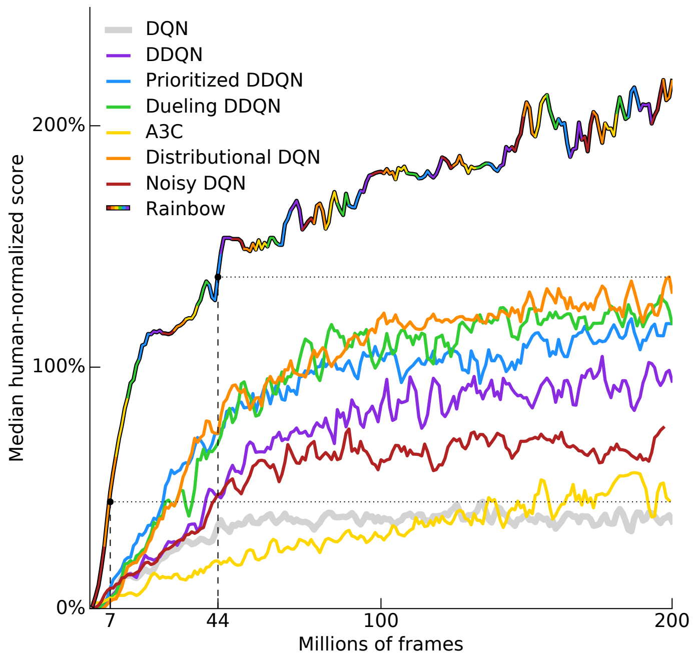


Figure 10.1: Median performance of Rainbow DQN across 57 Atari games; Taken from "Rainbow: Combining Improvements in Deep Reinforcement Learning" (Hessel et al., 2018).

10.3 Other DQN-Based Exploration Techniques

In this section we explore some other techniques based on DQN, mainly to improve exploration.

10.3.1 Count-Based Exploration

Some environment such as jump-n-run games are simply too difficult to be solved. Some reasons for this are sparse rewards (i.e., you only rarely get any reward such as at the end of the episode), complex dynamics, and a high probability of failures (such that it is hard to success by chance). One approach for tackling this problem is to alter the extrinsic (environmental) rewards $R_{\text{ext}}(s, a)$ by adding an *intrinsic* rewards $R_{\text{int}}(s, a)$:

$$R_{\text{tot}}(s, a) = R_{\text{ext}}(s, a) + R_{\text{int}}(s, a).$$

In order to guide the agent towards exploration, we increase the intrinsic reward in places the agent has visited only a few times,

$$R_{\text{int}}(s, a) = \beta\big(N(s) + 0.01\big)^{-1/2},$$

where $N(s)$ is the number of times s was visited and β is a hyper-parameter. We call the idea of exploring where we are uncertain *optimism in the face of uncertainty* and the concrete approach *count-based exploration.* While this approach is straightforward for discrete state-spaces, we cannot just count occurrences in continuous spaces. Instead, we have to define some notion of *similarity* (usually found using unsupervised learning like clustering, kernel density estimation (KDE), auto-encoders, etc.) and count similar states. These counts are then *pseudo-counts.*

10.3.2 Curiosity-Driven Exploration

In *curiosity-driven exploration*, we also add an intrinsic reward $R_{\text{int}}(s, a)$, but we use a notion of *curiosity* and want to explore where the knowledge of our model is bad. We hypothesize that states are more interesting if the distance of the predicted next state to the actual next state is large. Hence, we use the intrinsic reward

$$R_{\text{int}}(s, a) = \frac{\eta}{2} \big\| \boldsymbol{\phi}(\tilde{s}') - \boldsymbol{\phi}(s') \big\|_2^2$$

where $\tilde{s}'$ is the predicted and s' is the actual next state and η is a hyper-parameter.

10.3.3 Ensemble-Driven Exploration

Instead of learning a single action-value function, in *ensemble-driven exploration* we train multiple value functions as an ensemble. The ensemble itself is usually implemented by as single NN with κ regression heads. During training, each head is trained with different samples where the samples are chosen according to some masking distribution and the value function for acting in the environment is chosen once per trajectory. The corresponding algorithm is called *bootstrapped DQN*.

10.4 Wrap-Up

- curse of dimensionality and its effect in RL
- deep learning in RL for handling high-dimensional problems
- problems of deep RL and some techniques addressing them
- the DQN algorithm and how to set up experiments with it
- enhancing DQN by improving function approximation and sample usage
- improving key problems of RL, e.g., exploration, by combining deep learning techniques and DQN
- Additional reading material:
  - Paper: “Playing Atari with Deep Reinforcement Learning” (Mnih et al., 2013)
  - Paper: “Deep Reinforcement Learning with Double Q-Learning” (Hasselt et al., 2016)
  - Paper: “A Distributional Perspective on Reinforcement Learning” (Bellemare et al., 2017)
  - Paper: “Curiosity-Driven Exploration by Self-Supervised Prediction” (Pathak et al., 2017)

11 Deep Actor-Critic

In this chapter we move from “just” value-function methods to the SOTA of deep RL: deep actor-critic methods. These methods explicitly model both a policy (the “actor”) and the value function (the “critic”), cf. Figure 9.1. Of course, our models are only approximate, so the policy gradient estimate will be biased and we will replace the true objective $\mathcal{J}(\boldsymbol{\theta})$ with a surrogate object8ive $\mathcal{L}(\boldsymbol{\theta})$ that is easier to compute. As this surrogate objective is a central concept, we will start our discussion of actor-critic methods with it.

Before, we have to clarify some nomenclature: while in TD learning the difference between on- and off-policy algorithms is whether the samples come from the actual policy or a behavioral policy, respectively, in deep actor-critic methods we use these terms differently. *On-policy* methods updates the policy only with samples from the previous policy while *off-policy* methods use a replay buffer (note that as long as the policy does not change “much,” we still reuse old samples in on-policy methods). We broaden this distinction as with the TD definition, almost all deep actor-critic methods would be “on-policy.”

11.1 Surrogate Loss/Objective

We start our discussion of deep actor-critic methods with the important *surrogate objective*. First, why do we need the surrogate objective in the first place? According to the policy gradient theorem (PGT) (Theorem 16), the PG is an expectation under the *discounted* state distribution (27) induced by the policy π. The *discounted* state distributions implies that we have a probability $1-\gamma$ of terminating a sequence at every transition; hence, we waste a lot of samples! An alternative is to use the *undiscounted* distribution and discount the gradient, but this just causes later gradients to be very small. Also, this approach is very difficult to implement in off-policy settings (where we sample from a replay buffer). In deep actor-critic, we trade off the mathematical rigor of optimizing the “correct” objective for performance and instead use a surrogate objective.

Theorem 18 (Kakade-Langford-Lemma). *Let π and q be arbitrary policies, then $\mathcal{J}(\pi)$ can be expressed as*

$$\mathcal{J}(\pi) = \mathcal{J}(q) + \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t A^q(s_t, a_t)\right] \tag{11.1}$$

where A^q is the advantage function w.r.t. q and d^π is the discounted state distribution w.r.t. π.

Proof. By the definition of the advantage and action-value function, we can write

$$A^q(s_t, a_t) \doteq Q^q(s_t, a_t) - V^q(s_t, a_t) \doteq \mathbb{E}_{s_{t+1}}\big[R(s_t, a_t) + \gamma V^q(s_{t+1}) - V^q(s_t)\big].$$

hence, we can rewrite the expectation in (11.1) as follows:

$$\begin{aligned}\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t A^q(s_t,a_t)\right] &= \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\big(R(s_t,a_t)+\gamma V^q(s_{t+1})-V^q(s_t)\big)\right]\\ &= \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)\right]+\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\big(\gamma V^q(s_{t+1})-V^q(s_t)\big)\right]\\ &= \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t)\right]+\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t+1}V^q(s_{t+1})-\gamma^t V^q(s_t)\right].\end{aligned}$$

By definition, the left expectation equals $\mathcal{J}(\pi)$. Comparing with (11.1), what remains to be shown is that the right expectation equals $-\mathcal{J}(q)$:

$$\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t+1}V^q(s_{t+1})-\gamma^t V^q(s_t)\right]=\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=1}^{\infty}\gamma^t V^q(s_t)-\sum_{t=0}^{\infty}\gamma^t V^q(s_t)\right]=\mathbb{E}_{s_0\sim\iota}\big[V^q(s_0)\big]=-\mathcal{J}(q).$$

With this result and by plugging everything back in, we get the Kakade-Langford-lemma. □

Let $\pi_{\boldsymbol{\theta}}$ and q be arbitrary policies, then we can apply the Kakade-Langford-lemma (Theorem 18) and write the objective as

$$\mathcal{J}(\pi_{\boldsymbol{\theta}})=\mathcal{J}(q)+\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t A^q(s_t,a_t)\right]=\mathcal{J}(q)+\mathbb{E}_{s\sim d^{\pi_{\boldsymbol{\theta}}}(\cdot),a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[A^q(s,a)\big]$$

where we swapped the limits of expectation and sum. Note that this is still equivalent to the original objective, but with a baseline policy q. As already discussed, it is difficult of computing the gradient of this objective as it contains $d^{\pi_{\boldsymbol{\theta}}}$. Instead, we replace $d^{\pi_{\boldsymbol{\theta}}}$ with the discounted state distribution w.r.t. q, d^q. This yields the surrogate objective:

$$\mathcal{L}_q(\pi_{\boldsymbol{\theta}}):=\mathcal{J}(q)+\mathbb{E}_{s\sim d^q(\cdot),a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[A^q(s,a)\big].$$

Notice that this objective is consistent in the sense that if $q=\pi_{\boldsymbol{\theta}}$, the surrogate objective and true objective are equivalent and so are their gradients. As $\mathcal{J}(q)$ is constant w.r.t. $\boldsymbol{\theta}$, we often use the following surrogate objective:

$$\mathcal{L}(\pi_{\boldsymbol{\theta}})=\mathbb{E}_{s\sim d^q(\cdot),a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[A^q(s,a)\big].$$

Also, we approximate the advantage function $\hat{A}(s,a)\approx A(s,a)$. We select q to be the previous policy $\pi_{\boldsymbol{\theta}_k}$ and $\pi_{\boldsymbol{\theta}}$ to be the policy we are currently optimizing, i.e., taking the gradient w.r.t. its parameter $\boldsymbol{\theta}$. Additionally, most of the algorithms we discuss introduce another approximation by not using the discounted state distribution d^q but using the undiscounted one, u^q.

11.2 Advantage Actor-Critic (A2C)

Advantage actor-critic (A2C) is the simplest deep actor-critic methods and just uses MC estimates of the value and advantage function. Subsequently, it just follows the gradient of the surrogate loss,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})=\mathbb{E}_{s\sim d^q,a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[\big(\boldsymbol{\nabla}_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a\,|\,s)\big)\hat{A}(s,a)\big],$$

where we use the log-ratio trick again. By replacing the d^q with u^q and just computing the expectation by sampling transitions, we arrive at the A2C objective

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \Big(\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}\big(a^{[i]} \,|\, s^{[i]}\big) \Big) \hat{A}\big(s^{[i]}, a^{[i]}\big).$$

This estimator is similar to the PGT estimate (9.19), but are sampling from the *undiscounted* state distribution. A2C is summarized in algorithm 20.

Algorithm 20: Advantage Actor-Critic (A2C)

```
   Input: ordered transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^{N}$; differentiable approximators $\pi_{\boldsymbol{\theta}}$ and $\hat{V}_{\boldsymbol{\omega}}$ with
          parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$
1
   Output: optimized parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$
   // Compute Monte-Carlo advantage.
2  $q_{\text{next}} \leftarrow V(s_N)$
3  foreach $i = N, N-1, \dots, 1$ do
4      if $s_i$ is terminal then
5          $q_{\text{next}} \leftarrow r_i$
6      else
7          $q_{\text{next}} \leftarrow r_i + \gamma q_{\text{next}}$
8      $V'(s_i) \leftarrow q_{\text{next}}$
9      $A(s_i, a_i) \leftarrow V'(s_i) - V_{\boldsymbol{\omega}}(s_i)$
10 $\boldsymbol{\omega} \leftarrow$ fit value function $V_{\boldsymbol{\omega}}$ using $\mathcal{D}$ and $V'$
   // Optimize surrogate objective.
11 $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{(s,a,\cdot,\cdot) \in \mathcal{D}} \log \pi_{\boldsymbol{\theta}}(a \,|\, s) A(s, a)$
12 $\boldsymbol{\theta} \leftarrow$ maximize $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$
13 return $\boldsymbol{\theta}$, $\boldsymbol{\omega}$
```

11.3 On-Policy Methods

In this section we discuss a variety of on-policy methods which, in the deep actor-critic sense, are on-policy as they do not use a replay buffer.

11.3.1 Trust-Region Policy Optimization (TRPO)

In *trust-region actor-critic (TRPO),* we essentially use the NG but optimize the policy only in a "trust region" around the old policy where we are sure to have enough information. This idea is formalized in the following optimization problem:

$$\begin{aligned} \boldsymbol{\theta}^* &= \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \\ \text{s.t.} \quad & \mathbb{E}_{s \sim u^q}\Big[\mathrm{KL}\big(q(a \,|\, s) \,\big\|\, \pi_{\boldsymbol{\theta}}(a \,|\, s)\big)\Big] \le \varepsilon \end{aligned}$$

where we use the importance sampling surrogate objective

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{s,a\sim q}\left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{q(a \mid s)}\hat{A}(s,a)\right].$$

Remember that q is the policy of the previous iteration and $\pi_{\boldsymbol{\theta}}$ is the policy we are currently optimizing. With this approach, we get an *off-policy objective,* but *on-policy data.* The advantage is estimated using *generalized advantage estimation (GAE)*

$$\hat{A}^{\mathrm{GAE}(\lambda)}(s_t, a_t) = \sum_{\ell=0}^{\infty}(\gamma\lambda)^{\ell}\delta_{t+\ell}^{V}$$

with the TD error $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$.

To implement TRPO, we perform a few simplifications to make the problem tractable, assuming $\pi_{\boldsymbol{\theta}^*} \approx q$:

1. approximate the KL divergence with the FIM
2. compute the NG $\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{NG}} = \boldsymbol{F}_{\boldsymbol{\theta}}^{-1}\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}$
3. use line search to find the optimal step size along the NG direction

However, as the FIM is high-dimensional and inverting it is not tractable for large NNs with millions of parameters, we reformulate the calculation of the NG as a linear system $\boldsymbol{F}_{\boldsymbol{\theta}}\boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{NG}} = \boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta}}$. Subsequently, we can use conjugate gradient (CG) to solve this system efficiently.

Nevertheless, computing the FIM explicitly is still costly. Instead, we can use automatic differentiation to efficiently compute Fisher-vector products $\boldsymbol{F}_{\boldsymbol{\theta}}\boldsymbol{x}$ directly without computing the FIM. We do this using the definition of the FIM and "push in" the multiplication with $\boldsymbol{x}$:

$$\boldsymbol{F}_{\boldsymbol{\theta}}\boldsymbol{x} = \big(\boldsymbol{\nabla}_{\boldsymbol{\theta}}\boldsymbol{\nabla}_{\boldsymbol{\theta}}\,\mathrm{KL}(\pi_{\boldsymbol{\theta}'} \,\|\, \pi_{\boldsymbol{\theta}})\big)\boldsymbol{x} = \boldsymbol{\nabla}_{\boldsymbol{\theta}}\boldsymbol{\nabla}_{\boldsymbol{\theta}}\big(\mathrm{KL}(\pi_{\boldsymbol{\theta}'} \,\|\, \pi_{\boldsymbol{\theta}})\boldsymbol{x}\big).$$

With automatic differentiation, this yields an efficient way of calculating $\boldsymbol{F}_{\boldsymbol{\theta}}\boldsymbol{x}$.

The whole algorithm is summarized in algorithm 21.

Algorithm 21: Trust-Region Policy Optimization

```
Input: sufficiently on-policy transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^N$, policy approximator $\pi_{\boldsymbol{\theta}}$, value
       approximator $V_{\boldsymbol{\omega}}$, previous policy $q$
Output: optimized parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$
1 $V', A \leftarrow$ compute using GAE of $\mathcal{D}$ using $V_{\boldsymbol{\omega}}$
2 $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N}\sum_{(s,a,\cdot,\cdot)\in\mathcal{D}}\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{q(a \mid s)}A(s,a)$    // compute surrogate objective
3 $\boldsymbol{b} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{\theta}}^{\mathrm{VG}}\mathcal{L}(\boldsymbol{\theta})$    // compute vanilla gradient
4 $\boldsymbol{d} \leftarrow \boldsymbol{F}_{\boldsymbol{\theta}}^{-1}\boldsymbol{b}$    // compute with CG and Fisher-vector product
5 $\boldsymbol{\theta} \leftarrow$ line search in direction $\boldsymbol{d}$ w.r.t. objective $\mathcal{L}(\boldsymbol{\theta})$ and constraint $\mathrm{KL}(q \,\|\, \pi_{\boldsymbol{\theta}})$
6 $\boldsymbol{\omega} \leftarrow$ fit value function using $\mathcal{D}$ and $V'$
7 return $\boldsymbol{\theta}$, $\boldsymbol{\omega}$
```

11.3.2 Proximal Policy Optimization (PPO)

Another trust-region-based approach is *proximal policy optimization (PPO)*. In PPO, we include the constraint present in TRPO into the objective by clipping the importance sampling weight:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{s,a\sim q}\left[\operatorname{clip}_{-\varepsilon}^{+\varepsilon}\left(\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{q(a \mid s)}\right)\hat{A}(s,a)\right].$$

This clipping prevents policy updates that deviate too much from the sampling (previous) distribution q and is therefore an *implicit* formulation of the KL constraint. This approach is summarized in algorithm 22.

Algorithm 22: Proximal Policy Optimization

```
Input: sufficiently on-policy transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s_i' \rangle_{i=1}^N$, policy approximator $\pi_{\boldsymbol{\theta}}$, value
       approximator $V_{\boldsymbol{\omega}}$, previous policy $q$
Output: optimized parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$
1 $V', A \leftarrow$ compute using GAE of $\mathcal{D}$ using $V_{\boldsymbol{\omega}}$
2 $\boldsymbol{\omega} \leftarrow$ fit value function using $\mathcal{D}$ and $V'$
3 for $N \leftarrow 1, 2, \dots, N$ do
4     split $\mathcal{D}$ into $M$ minibatches $\{\mathcal{B}_m\}_{m=1}^M$ of size $N$
5     for $m \leftarrow 1, 2, \dots, M$ do
          // Maximize surrogate objective.
6         $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{N}\sum_{(s,a,\cdot,\cdot)\in\mathcal{B}_m} \operatorname{clip}_{-\varepsilon}^{+\varepsilon}\left(\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{q(a \mid s)}\right)\hat{A}(s,a)$
7         $\boldsymbol{\theta} \leftarrow$ maximize $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$, e.g., using Adam
8 return $\boldsymbol{\theta}$, $\boldsymbol{\omega}$
```

11.4 Off-Policy Methods

In this section we discuss a variety of off-policy methods which, in the deep actor-critic sense, are off-policy as they use a replay buffer.

11.4.1 Deep Deterministic Policy Gradient (DDPG)

The *deep deterministic policy gradient (DDPG)* is based on the *deterministic* PGT:

Theorem 19 (Deterministic Policy Gradient Theorem). *Let $\mu_{\boldsymbol{\theta}} : \mathcal{S} \to \mathcal{A}$ be a deterministic policy. Then the policy gradient can be computed as*

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mathcal{J}_{\boldsymbol{\theta}} \propto \mathbb{E}_{s\sim d^{\mu_{\boldsymbol{\theta}}}}\left[\big(\boldsymbol{\nabla}_{\boldsymbol{\theta}}\mu_{\boldsymbol{\theta}}(s)\big)\boldsymbol{\nabla}_a Q^{\mu_\theta}(s,a)\big|_{a=\mu_{\boldsymbol{\theta}}(s)}\right]$$

where $d^{\mu_{\boldsymbol{\theta}}}$ is the discounted state distribution w.r.t. $\mu_{\boldsymbol{\theta}}$.

For Gaussian policies, the deterministic PGT is a limit case of the PGT as $\sigma^2 \to 0$. Similar results can be found for other classes of policies.

However, the deterministic PGT assumes knowledge of the true action-value function which we do not have. Hence, to develop a practical algorithm, we include the following tricks:

- use a noisy policy to estimate Q^{π}
  - Gaussian noise
  - truncated Gaussian noise to respect action limits
  - Ornstein-Uhlenbeck process for correlated exploration
  - ...
- then update the deterministic policy parameters using the deterministic PGT
- use replay memory and target networks (for both actor and critic) to improve stability

The whole approach, including these tricks, is summarized in algorithm 23.

Algorithm 23: Deep Deterministic Policy Gradient (DDPG)

```
   Input: transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s'_i \rangle_{i=1}^{N}$, policy approximator $\pi_{\hat{\boldsymbol{\theta}}}$, action-value function $Q_{\hat{\boldsymbol{\omega}}}$
   Output: optimized target parameters $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}$
1  sample a minibatch $\mathcal{B}$ from $\mathcal{D}$
   // Compute action-value estimate from $\mathcal{B}$.
2  foreach $(s, a, r, s') \in \mathcal{B}$ do
3      if $s'$ is terminal then
4          $q_{\text{next}} \leftarrow 0$
5      else
6          $q_{\text{next}} \leftarrow Q_{\hat{\boldsymbol{\omega}}}\big(s', \mu_{\hat{\boldsymbol{\theta}}}(s')\big)$
7      $Q'(s, a) \leftarrow r + \gamma q_{\text{next}}$
8  $\boldsymbol{\omega} \leftarrow$ fit action-value function $Q_{\boldsymbol{\omega}}$ using $\mathcal{D}$ and $Q'$
   // Maximize the surrogate objective.
9  $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{|\mathcal{B}|} \sum_{(s,\cdot,\cdot,\cdot) \in \mathcal{B}} Q_{\boldsymbol{\omega}}\big(s, \mu_{\boldsymbol{\theta}}(s)\big)$
10 $\boldsymbol{\theta} \leftarrow$ maximize $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$, e.g., using Adam
   // Mix target and current parameters.
11 $\hat{\boldsymbol{\theta}} \leftarrow \tau \boldsymbol{\theta} + (1 - \tau)\hat{\boldsymbol{\theta}}$
12 $\hat{\boldsymbol{\omega}} \leftarrow \tau \boldsymbol{\omega} + (1 - \tau)\hat{\boldsymbol{\omega}}$
13 return $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}$
```

11.4.2 Twin Delayed DDPG (TD3)

Despite all tricks, the DDPG still has three major issues:

1. the action-value function is often overestimated which may lead to divergence
2. the gradient estimate might have high variance as the current value estimate might not have converged to $Q^{\pi_{\boldsymbol{\theta}}}$
3. deterministic policies tend to overfit to peaks in the value function which are often due to poor estimates rather than the environment

All of these issue arise from the action-value function being an estimate and not the true value function! *Twin Delayed DDPG (TD3)* tackles these issues by adding some minor modifications to DDPG, namely:

1. train two value functions and using their minimum (twin); tackles the overestimation
2. delay policy updates and only update after a fixed amount of steps (delayed); tackles gradient variance
3. add noise to compute the next value; tackles overfitting

The whole approach is summarized in algorithm 24.

Algorithm 24: Twin Delayed DDPG (TD3)

```
Input: transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s'_i \rangle_{i=1}^N$, policy approximator $\pi_{\hat{\boldsymbol{\theta}}}$, action-value functions $Q_{\hat{\boldsymbol{\omega}}_i}$,
       $i \in \{0, 1\}$
Output: optimized target parameters $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}$
1  sample a minibatch $\mathcal{B}$ from $\mathcal{D}$
   // Compute action-value estimate from $\mathcal{B}$.
2  foreach $(s, a, r, s') \in \mathcal{B}$ do
3      if $s'$ is terminal then
4          $q_{\text{next}} \leftarrow 0$
5      else
           // Add noise and clip to match action bounds.
6          $a_{\text{next}} \leftarrow \mu_{\hat{\boldsymbol{\theta}}}(s')$
7          $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$
8          $\epsilon_{\text{clip}} \leftarrow \text{clip}_{-\delta_\epsilon}^{+\delta_\epsilon}(\epsilon)$
9          $a_{\text{noisy}} \leftarrow a + \epsilon_{\text{clip}}$
10         $a_{\text{clip}} \leftarrow \text{clip}_{a_{\min}}^{a_{\max}}(a_{\text{noisy}})$
11         $q_{\text{next}} \leftarrow \min_i Q_{\hat{\boldsymbol{\omega}}_i}\left(s', a_{\text{clip}}\right)$
12     $Q'(s, a) \leftarrow r + \gamma q_{\text{next}}$
13 $\boldsymbol{\omega} \leftarrow$ fit action-value functions $Q_{\boldsymbol{\omega}_i}$, $i \in \{0, 1\}$ using $\mathcal{D}$ and $Q'$
14 if policy update should take place then
       // Maximize the surrogate objective.
15     $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \frac{1}{|\mathcal{B}|} \sum_{(s,\cdot,\cdot,\cdot) \in \mathcal{B}} Q_{\boldsymbol{\omega}_0}\left(s, \mu_{\boldsymbol{\theta}}(s)\right)$
16     $\boldsymbol{\theta} \leftarrow$ maximize $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$, e.g., using Adam
17 else
       // Do not change the policy parameters.
18     $\boldsymbol{\theta} \leftarrow \hat{\boldsymbol{\theta}}$
   // Mix target and current parameters.
19 $\hat{\boldsymbol{\theta}} \leftarrow \tau \boldsymbol{\theta} + (1 - \tau) \hat{\boldsymbol{\theta}}$
20 $\hat{\boldsymbol{\omega}}_i \leftarrow \tau \boldsymbol{\omega}_i + (1 - \tau) \hat{\boldsymbol{\omega}}_i$ for $i \in \{0, 1\}$
21 return $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}_i$, $i \in \{0, 1\}$
```

11.4.3 Soft Actor-Critic (SAC)

Opposed to DDPG (and TD3), *soft actor-critic (SAC)* uses a stochastic policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ with the following objective:

$$\mathcal{J}^\alpha(\boldsymbol{\theta}) = \mathcal{J}(\boldsymbol{\theta}) + \alpha \mathbb{H}[\pi_{\boldsymbol{\theta}}],$$

where $\mathbb{H}[\pi_{\boldsymbol{\theta}}] = -\mathbb{E}_{s\sim d^{\pi_{\boldsymbol{\theta}}}(\cdot), a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[\log \pi_{\boldsymbol{\theta}}(a\,|\,s)\big]$ is the policy's (expected) entropy and α is a hyper-parameter. As this objective is very difficult to optimize, SAC uses the surrogate objective

$$\mathcal{L}^{\alpha}(\boldsymbol{\theta}) = \mathbb{E}_{s\sim u^q, a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \alpha \log \pi_{\boldsymbol{\theta}}(a\,|\,s)\big]. \tag{11.2}$$

This objective equals the expectation of the *soft value function*

$$V_{\alpha}^{\pi_{\boldsymbol{\theta}}}(s) \coloneqq \mathbb{E}_{a\sim\pi(\cdot\,|\,s)}\big[Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \alpha \log \pi_{\boldsymbol{\theta}}(a\,|\,s)\big]$$

under the undiscounted state distribution u^q. But this surrogate objective is still an expectation w.r.t. the actions (opposed to DDPG and TD3 where the action was deterministic and just "plugged in")! Hence, its gradient is hard to compute. Instead of using the usual log-ratio trick, SAC used the reparametrization trick (see subsection 2.2.6) which exhibits less variance. Let $g_{\boldsymbol{\theta}}(s;\varepsilon)$ be the reparameterized policy, i.e., $a = g_{\boldsymbol{\theta}}(s;\varepsilon)$, with $\varepsilon \sim p(\cdot)$ being the noise. We can then rewrite the objective (11.2) as

$$\mathcal{L}^{\alpha}(\boldsymbol{\theta}) = \mathbb{E}_{s\sim u^q, a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[Q^{\pi_{\boldsymbol{\theta}}}(s,a) - \alpha \log \pi_{\boldsymbol{\theta}}(a\,|\,s)\big] = \mathbb{E}_{s\sim u^q, \varepsilon\sim p}\big[Q^{\pi_{\boldsymbol{\theta}}}\big(s, g_{\boldsymbol{\theta}}(s;\varepsilon)\big) - \alpha \log \pi_{\boldsymbol{\theta}}\big(g_{\boldsymbol{\theta}}(s;\varepsilon)\,|\,s\big)\big] \tag{11.3}$$

and we can readily compute its gradient by applying the chain rule. So far, the objective (11.3) depends on the hyper-parameter α which is very difficult to optimize. Instead, it would be easier to specify a target entropy $\bar{\mathbb{H}}$ and selecting α accordingly. Hence, SAC automatically tunes α by solving

$$\alpha^* = \arg\min_{\alpha} \mathcal{L}(\alpha) = \mathbb{E}_{s\sim u^q, a\sim\pi_{\boldsymbol{\theta}}(\cdot\,|\,s)}\big[-\alpha \log \pi_{\boldsymbol{\theta}}(a\,|\,s) - \alpha\bar{\mathbb{H}}\big].$$

Also, SAC used squashed Gaussian policies, i.e., the actions are pushed through a tanh after sampling to ensure they lie in $[-1, +1]$. The mean and covariance of the Gaussian itself are the policy and are therefore state-dependent.

11.5 Wrap-Up

- difficulties of the PGT in practice
- simplification in deep RL using surrogate objectives
- on-policy methods using samples from the current policy
  - A2C, the simplest extension
  - TRPO, optimizing the policy inside a trust-region
  - PPO, a simpler way to the trust-region approach
- using CG for computing the NG in large NNs
- off-policy methods using a replay memory
  - DDPG, using the deterministic PGT
  - TD3, ensuring more robust gradient estimates by avoiding overfitting and overestimation
  - SAC, extending the deterministic approach to stochastic policies using reparametrization
- Additional reading material:
  - Paper: "Approximately Optimal Approximate Reinforcement Learning" (Kakade and Langford, 2002), Lemma 6.1

Algorithm 25: Soft Actor-Critic (SAC)

```
   Input: transition dataset $\mathcal{D} = \langle s_i, a_i, r_i, s'_i \rangle_{i=1}^{N}$, policy approximator $\pi_{\hat{\boldsymbol{\theta}}}$, action-value function $Q_{\hat{\boldsymbol{\omega}}_i}$,
          $i \in \{0, 1\}$
   Output: optimized target parameters $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}_i$, $i \in \{0, 1\}$
1  sample a minibatch $\mathcal{B}$ from $\mathcal{D}$
   // Minimize the entropy objective.
2  $\mathcal{L}(\alpha) = \frac{1}{N} \sum_{(s,a,r,s') \in \mathcal{B}} -\alpha \log \pi_{\hat{\boldsymbol{\theta}}}(a \mid s) - \alpha \bar{\mathbb{H}}$
3  $\alpha \leftarrow$ minimize $\mathcal{L}(\alpha)$ w.r.t. $\alpha$
   // Compute soft action-value estimate from $\mathcal{B}$.
4  foreach $(s, a, r, s') \in \mathcal{B}$ do
5      if $s'$ is terminal then
6          $q_{\text{next}} \leftarrow 0$
7      else
8          $a' \sim \pi_{\boldsymbol{\theta}}(\cdot \mid s')$
9          $q_{\text{next}} \leftarrow \min_i Q_{\hat{\boldsymbol{\omega}}_i}(s', a') - \log \pi_{\boldsymbol{\theta}}(a' \mid s')$
10     $Q'(s, a) \leftarrow r + \gamma q_{\text{next}}$
11 $\boldsymbol{\omega}_i \leftarrow$ fit action-value functions $Q_{\boldsymbol{\omega}_i}$, $i \in \{0, 1\}$ using $\mathcal{D}$ and $Q'$
   // Maximize the surrogate objective.
12 $\mathcal{L}(\boldsymbol{\theta}) \leftarrow 0$
13 foreach $(s, a, r, s') \in \mathcal{B}$ do
       // Use the reparametrization trick.
14     $\varepsilon \sim p(\cdot)$
15     $a' \leftarrow g_{\boldsymbol{\theta}}(s; \varepsilon)$
16     $\mathcal{L}(\boldsymbol{\theta}) \leftarrow \mathcal{L}(\boldsymbol{\theta}) + \frac{1}{|\mathcal{B}|}\big(Q_{\boldsymbol{\omega}_0}(s, a') - \alpha \log \pi_{\boldsymbol{\theta}}(a' \mid s)\big)$
17 $\boldsymbol{\theta} \leftarrow$ maximize $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$
   // Mix target and current parameters.
18 $\hat{\boldsymbol{\theta}} \leftarrow \tau \boldsymbol{\theta} + (1 - \tau) \hat{\boldsymbol{\theta}}$
19 $\hat{\boldsymbol{\omega}}_i \leftarrow \tau \boldsymbol{\omega}_i + (1 - \tau) \hat{\boldsymbol{\omega}}_i$, $i \in \{0, 1\}$
20 return $\hat{\boldsymbol{\theta}}$, $\hat{\boldsymbol{\omega}}_i$, $i \in \{0, 1\}$
```

- Paper: "Asynchronous Methods for Deep Reinforcement Learning" (Mnih et al., 2016)
- Paper: "Trust Region Policy Optimization" (Schulman et al., 2015)
- Paper: "Proximal Policy Optimization Algorithms" (Schulman et al., 2017)
- Paper: "Continuous Control with Deep Reinforcement Learning" (Lillicrap et al., 2015)
- Paper: "Addressing Function Approximation Error in Actor-Critic Methods" (Fujimoto et al., 2018)
- Paper: "Soft Actor-Critic Algorithms and Applications" (Haarnoja et al., 2018)

12 Frontiers

12.1 Partial Observability

12.2 Hierarchical Control

12.2.1 The Options Framework

12.3 Markov Decision Processed Without Reward

12.3.1 Intrinsic Motivation

12.3.2 Inverse Reinforcement Learning

12.4 Model-Based Reinforcement Learning

12.5 Wrap-Up