Reinforcement Learning

Summary

Fabian Damken

July 28, 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).

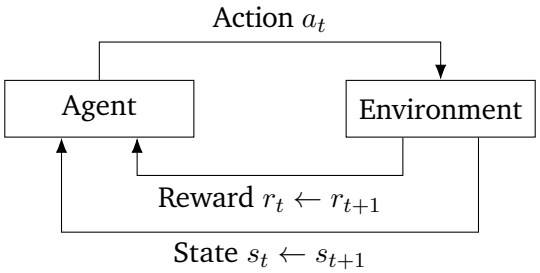


Figure 1.1: The Reinforcement Learning Cycle

	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

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, $\sqrt{n}(\hat{\mu}_n - \mu) \xrightarrow{D} \mathcal{N}(0, \sigma^2)$.

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):

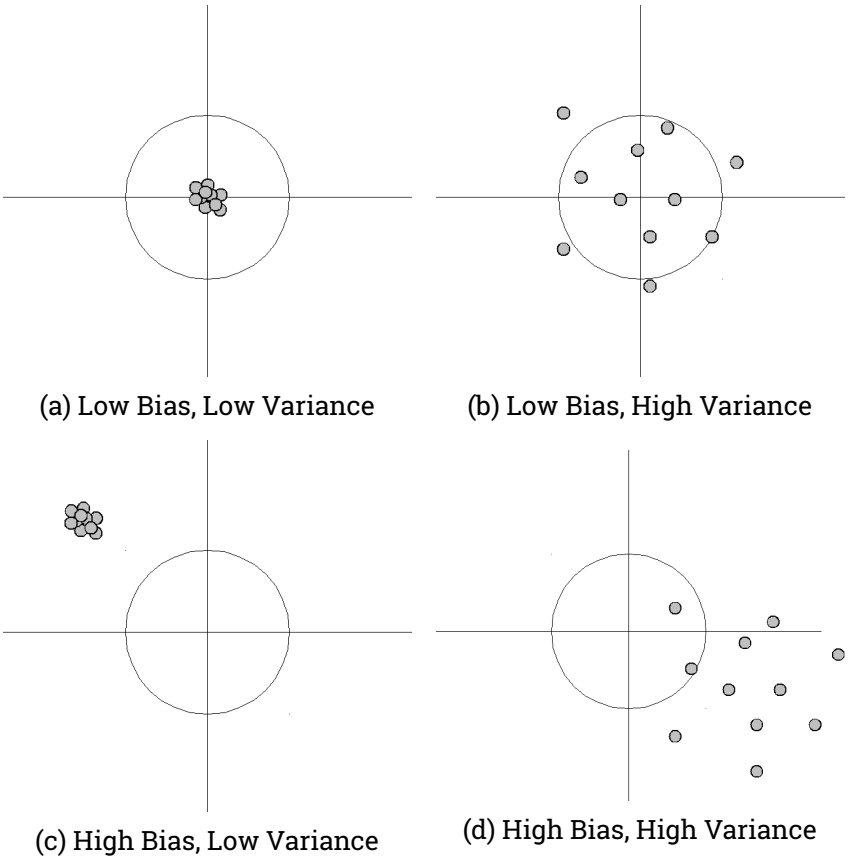
$$\mathbb{E}_{x\sim p}\big[f(x)\big] = \sum_x f(x)p(x) = \sum_x f(x)\frac{p(x)}{q(x)}q(x) = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$$
$$\mathbb{E}_{x\sim p}\big[f(x)\big] = \int f(x)p(x)\,\mathrm{d}x = \int f(x)\frac{p(x)}{q(x)}p(x)\,\mathrm{d}x = \mathbb{E}_{x\sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$$

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)

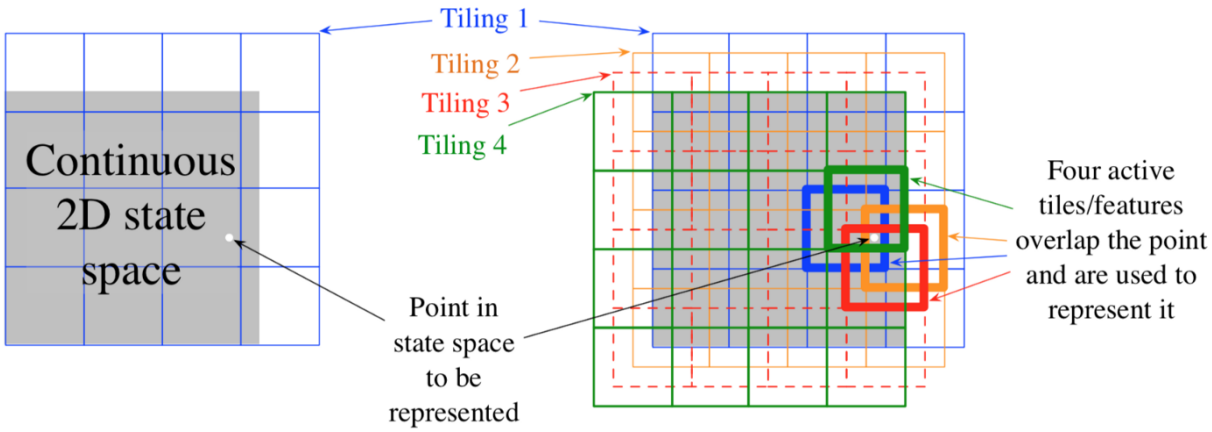


Figure 2.2: Tile Coding; Source: `https://towardsdatascience.com/reinforcement-learning-tile-coding-implementation-7974b600762b`

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:

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

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

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

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

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

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 $p_\theta(x)$:

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

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 $P_t(S_{t+1} = s' \mid S_t = s, S_{t-1} = k_{t-1}, \ldots, S_0 = k_0) = P_t(S_{t+1} = s' \mid S_t = s)$ 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 $\boldsymbol{p}_{t+k} = \boldsymbol{p}_t \boldsymbol{P}^k$.*

Actions?	**All states observable?** 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 $\boldsymbol{p}_0 = (p_{0,1}, p_{0,2}, \ldots, p_{0,n})$ 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

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

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 $S_t \in \mathcal{S}$, $n \coloneqq |\mathcal{S}|$, 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:

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

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 $J(\tau) \coloneqq J_0(\tau)$.

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 $V(s) \coloneqq \mathbb{E}_{\boldsymbol{P}}[J_t \mid s_t = s]$ 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* $V(s) = R(s) + \gamma \mathbb{E}\big[V(s_{t+1}) \,\big|\, s_t = s\big]$.

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

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 $\boldsymbol{P}^a_{ss'} = P(s' \mid s, a)$, 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., $\pi(a \mid s) = P(a \mid s)$.

Note that we can reduce a deterministic policy $a = \pi(s)$ to a stochastic one using $\pi(a \mid s) = \mathbb{1}\big[a = \pi(s)\big]$ for discrete and $\pi(a \mid s) = \delta\big(a - \pi(s)\big)$ 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

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

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 $V^\pi(s) \coloneqq \mathbb{E}_{\boldsymbol{P},\pi}[J_t \mid s_t = s]$ for any t. That is, the *expected* return starting from state s where the expectation is w.r.t. the state dynamics 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 $Q^\pi(s, a) \coloneqq \mathbb{E}_{\boldsymbol{P},\pi}[J_t \mid s_t = s, a_t = a]$ 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:

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

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:*

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

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:

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

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

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 $\left(V^{(k)}\right)_{k=0}^{\infty}$ 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:

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

□

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 π^*

initialize π arbitrarily
repeat
  // Policy Evaluation
  $k \leftarrow 0$
  initialize $Q^{(k)}$ arbitrarily
  repeat
    $V^{(k+1)}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(s \mid a) R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) Q^{(k)}(s', a)$
    $k \leftarrow k + 1$
  until *until convergence*
  // Recover the action-value function.
  $Q^{(\infty)}(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{(\infty)}(s')$
  // Policy Improvement
  $\pi(a \mid s) \leftarrow \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} Q^{(\infty)}(s,a) \\ 0 & \text{otherwise} \end{cases}$
until *until convergence*
return $V^{(\infty)}$, $Q^{(\infty)}$, π

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 π^*
// Find optimal state-value function.
$k \leftarrow 0$
initialize $V^{(k)}$ arbitrarily
repeat
  $V^{(k+1)}(s) = \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{(k)}(s') \right\}$
  $k \leftarrow k + 1$
until *until convergence*
// Recover optimal policy.
$\pi(a \mid s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^{(k)}(s') \right\} \\ 0 & \text{otherwise} \end{cases}$
return $V^{(\infty)}$, π

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

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 π
Output: approximate V^π

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

Algorithm 4: Every-Visit Monte-Carlo Policy Evaluation

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

5.1 Example

5.2 Wrap-Up

- approximation of value functions when the dynamics are not available
- differences of DP and MC

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

repeat
  initialize s
  while *s is not terminal* **do**
    take action $a \sim \pi(\cdot \mid s)$, observe reward r and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $V(s) \leftarrow V(s) + \alpha\delta$
    $s \leftarrow s'$
until *convergence*
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*

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

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)

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

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

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:

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

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):

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

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

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

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,

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

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 $a_{t+1} \sim \pi^Q(\cdot \mid s_{t+1})$ 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

initialize Q arbitrarily except $Q(\text{terminal}, \cdot) = 0$
repeat
  initialize s
  choose $a \sim \pi^Q(\cdot \mid s)$
  while *s is not terminal* **do**
    take action a, observe reward r and next state s'
    choose $a' \sim \pi^Q(\cdot \mid s')$
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
    $s \leftarrow s'$
    $a \leftarrow a'$
until *converged*
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

initialize Q arbitrarily except $Q(\text{terminal}, \cdot) = 0$
repeat
  $z(s, a) \leftarrow 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
  initialize s
  choose $a \sim \pi^Q(\cdot \mid s)$
  while *s is not terminal* **do**
    take action a, observe reward r and next state s'
    choose $a' \sim \pi^Q(\cdot \mid s')$
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $z(s, a) \leftarrow z(s, a) + 1$
    foreach $\tilde{s} \in \mathcal{S}$ **do**
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta z(s, a)$
      $z(s, a) \leftarrow \gamma\lambda z(s, a)$
    $s \leftarrow s'$
    $a \leftarrow a'$
until *converged*
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:

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

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

initialize Q arbitrarily except $Q(\text{terminal}, \cdot) = 0$
repeat
  initialize s
  while s *is not terminal* **do**
    take action $a \sim q^Q(\cdot \mid s)$, observe reward r and next state s'
    $\delta \leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
    $s \leftarrow s'$
until *converged*
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 $V(s) \leftarrow \mathbb{E}_{a,s'}\left[R(s,a) + \gamma V(s') \,\middle|\, s\right]$	TD Learning $V(s_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma V(s_{t+1})$
Q-Policy Iteration $Q(s,a) \leftarrow R(s,a) + \gamma \mathbb{E}_{s',a'}\left[Q(s',a') \,\middle|\, s,a\right]$	SARSA $Q(s_t,a_t) \overset{\alpha}{\leftarrow} r_{t+1} + \gamma Q(s_{t+1},a_{t+1})$
Q-Value Iteration $Q(s,a) \leftarrow R(s,a) + \gamma \mathbb{E}_{s'}\left[\max_{a' \in \mathcal{A}} Q(s',a') \,\middle|\, s\right]$	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

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 $t = 0, 1, \ldots$, we can then locally update the weights using the update rule

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

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 π, differentiable approximator $\hat{V}$ with parameters $\boldsymbol{w}$
Output: estimated parameters $\boldsymbol{w}$

initialize $\boldsymbol{w}$ arbitrarily
repeat
  $(s_0; r_1, s_1; r_2, s_2; \ldots; r_{T-1}, s_{T-1}; r_T) \leftarrow$ generate episode
  foreach $t = 0, 1, \ldots, T$ **do**
    $J_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big[V(s_t) - \hat{V}(s;\boldsymbol{w})\big]\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s_t;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}}$
until *convergence*
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:

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

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.

$\boldsymbol{\phi}(s)$, 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 π, differentiable approximator $\hat{V}$ with parameters $\boldsymbol{w}$
Output: estimated parameters $\boldsymbol{w}$

initialize $\boldsymbol{w}$ arbitrarily
repeat
  initialize s
  while *s is not terminal* **do**
    take action $a \sim \pi(\cdot \mid s)$, observe reward r and next state s'
    $\delta \leftarrow r + \gamma\hat{V}(s';\boldsymbol{w}) - \hat{V}(s;\boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\delta\boldsymbol{\nabla}_{\boldsymbol{w}}\hat{V}(s;\boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}}$
    $s \leftarrow s'$
until *convergence*
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 π, differentiable approximator $\hat{Q}$ with parameters $\boldsymbol{w}$
Output: estimated parameters $\boldsymbol{w}$

initialize $\boldsymbol{w}$ arbitrarily
repeat
  initialize s
  choose $a \sim \pi^Q(\cdot \mid s)$
  while ***true*** **do**
    take action $a \sim \pi(\cdot \mid s)$, observe reward r and next state s'
    if s' *is terminal* **then**
      break
    choose $a' \sim \pi^Q(\cdot \mid s')$ $\delta \leftarrow r + \gamma \hat{Q}(s', a'; \boldsymbol{w}) - \hat{Q}(s, a; \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \delta \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{Q}(s_t, a_t; \boldsymbol{w})\big|_{\boldsymbol{w}=\boldsymbol{w}_t}$
    $s \leftarrow s'$
    $a \leftarrow a'$
until *convergence*
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 $\boldsymbol{b} \coloneqq r_{t+1}\boldsymbol{\phi}(s_t)$ and $\boldsymbol{A} \coloneqq \boldsymbol{\phi}(s_t)\big(\boldsymbol{\phi}(s_t) - \gamma\boldsymbol{\phi}(s_{t+1})\big)^\top$. 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,

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

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 π

initialize π arbitrarily
repeat
  $Q \leftarrow \text{LSTDQ}(\mathcal{D}, \pi)$
  $\pi(s) \leftarrow \arg\max_{a\in\mathcal{A}} Q(s,a)$ for all $s \in \mathcal{S}$
until *convergence*
return Q, π

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

initialize $\hat{Q}^{(0)}$ arbitrarily
$k \leftarrow 0$
repeat
  $\mathcal{T} \leftarrow \big\langle s_i,\ a_i,\ r_i + \gamma \max_{a\in\mathcal{A}} \hat{Q}^{(k)}(s'_i, a)\big\rangle_{i=1}^N$ // build training dataset
  $Q^{(k+1)} \leftarrow$ freshly trained regressor using $\mathcal{T}$
  $k \leftarrow k + 1$
until *convergence*
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

9 Policy Search

9.1 Policy Gradient

9.1.1 Computing the Gradient

Finite Differences

Least-Squares-Based Finite Differences

Likelihood-Ratio Trick

9.1.2 REINFORCE

Gradient Variance and Baselines

Example

9.1.3 GPOMDP

9.2 Natural Policy Gradient

9.3 The Policy Gradient Theorem

9.3.1 Actor-Critic

9.3.2 Compatible Function Approximation

Example

9.3.3 Advantage Function

9.3.4 Episodic Actor-Critic

9.4 Wrap-Up

10 Deep Reinforcement Learning

10.1 Deep Q-Learning: DQN

10.1.1 Replay Buffer

10.1.2 Target Network

10.1.3 Minibatch Updates

10.1.4 Reward- and Target-Clipping

10.1.5 Examples

10.2 DQN Enhancements

10.2.1 Overestimation and Double Deep Q-Learning

10.2.2 Prioritized Replay Buffer

10.2.3 Dueling DQN

10.2.4 Noisy DQN

10.2.5 Distributional DQN

10.2.6 Rainbow

10.3 Other DQN-Bases Methods

10.3.1 Count-Based Exploration

10.3.2 Curiosity-Driven Exploration

10.3.3 Ensemble-Driven Exploration

10.4 Wrap-Up

11 Deep Actor-Critic

11.1 Surrogate Loss

11.1.1 Kakade-Langford-Lemma

11.1.2 Practical Surrogate Loss

11.2 Advantage Actor-Critic (A2C)

11.3 On-Policy Methods

11.3.1 Trust-Region Policy Optimization (TRPO)

Practical Implementation

Conjugate Gradient

11.3.2 Proximal Policy Optimization (PPO)

11.4 Off-Policy Methods

11.4.1 Deep Deterministic Policy Gradient (DDPG)

11.4.2 Twin Delayed DDPG (TD3)

11.4.3 Soft Actor-Critic (SAC)

11.5 Wrap-Up

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