A gentle introduction to Reinforcement Learning with Python code examples

Andja Denic[1]

29.9.2024.

[1]Faculty of Sciences and Mathematics, University of Nis

Chapter 1

Abstract

This paper gives a gentle introduction to the field of reinforcement learning throwing example of the popular Toy Text environment in Gymnasium Frozen Lake to explain the elements mentioned.

The paper is divided in 5 chapters.

- In **chapter 1: Introduction to Reinforcement Learning** basic elements (state, action, reward, gain, value function, policy, model) are defined and discussed. We also introduce Finite Markov Decision Process - mathematical framework of Reinforcement Learning.
- In **chapter 2: Bellman equation** is stated and discussed the most famous equation (system of linear equations) in RL using Frozen Lake as an example.
- In **chapter 3: Dynamic Programming** we introduce the first set of algorithms for estimating the optimal policy. DP algorithms assume knowledge of the dynamics of the environment (transition probabilities). DP algorithm is just numerical solution of a system of n linear equations with n unknowns. Python code for Frozen Lake is listed, solutions are visualized and discussed.
- In **chapter 4: Monte Carlo Method** we give the second algorithm for estimating the optimal policy that does not require knowledge of the dynamics of the environment. The Monte Carlo algorithm just uses the definition of the state-action value function as expected value, and the basic statistical method (Monte Carlo method) to estimate the state-action value function. Once we have estimation of a state-value function, we can easily estimate optimal policy. Python code for Frozen Lake is listed, solutions are visualized and discussed.
- In **chapter 5: Temporal Difference** we give the third set of algorithm for estimating the optimal policy that (just like MC) does not require knowledge of the dynamics of the environment. TD methods use estimates to estimate (they bootstrap) state-action value functions. They combine DP and MC methods in that sense. There are few versions of TD algorithm for estimating the optimal policy and the most popular are: SARSA (one-step TD), n-step SARSA (n-step TD), expected SARSA (one-step TD) and Q-learning (one-step TD). Python code for all of them for Frozen Lake is listed, solutions are visualized and discussed.

Keywords: Reinforcement Learning · Frozen Lake · Gymnasium · Python code · Monte Carlo · Dynamic Programming · Temporal Difference

Chapter 2

Introduction to Reinforcement Learning

In this chapter we aim to introduce basic elements of reinforcement learning (state, action, reward, gain, value function, policy, model). We also introduce Finite Markov Decision Process - mathematical framework of Reinforcement Learning.

Goals of the field of AI

One of the primary goals of the field of AI is to produce fully autonomous agents that interact with their environments to learn optimal behaviors, improving over time through trial and error. Crafting AI systems that are responsive and can effectively learn has been a long-standing challenge, ranging from robots, which can sense and react to the world around them, to purely software-based agents, which can interact with natural language and multimedia. [1]

Goals of the Reinforcement Learning

Reinforcement learning (RL) is learning what to do - how to map situations to actions - to maximize a numerical reward signal. The learner is not told which actions to take but instead must discover which actions yield the most cumulative reward by trying them. [2]

Formalization of the Reinforcement Learning

The problem of RL is formalized using the **Markov decision process (MDP)** framework. MDPs are a classical framework of sequential decision-making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. In some cases, reinforcement learning can be taken beyond MDPs, but here we won't look at those examples. [2]

Relationship between Reinforce Learning and Machine Learning

Reinforcement learning is a machine learning paradigm, alongside supervised learning and unsupervised learning.

Elements of Reinforcement Learning and Notation

The Agent-Environment Interface and Notation in Frozen Lake Example

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the **agent**. The thing it interacts with, comprising everything outside the agent, is called the **environment**. These interact continually, the agent selecting **actions** and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to **rewards**, special numerical values that the agent seeks to *maximize over time* through its choice of actions. [2]

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a **policy**, a **reward signal**, a **value function**, and, optionally, a **model** of the environment.

We will use the popular Gymnasium's Toy Text environment Frozen Lake, shown in the figure 2.1 to explain the mentioned elements.

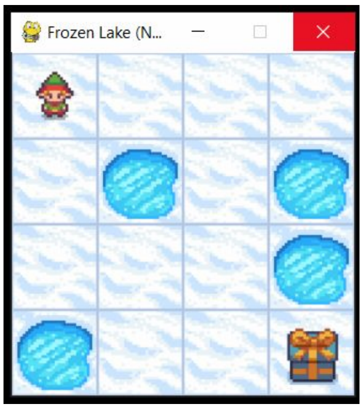


Figure 2.1: Frozen Lake environment

Gymnasium Frozen Lake is a simulated environment used for training reinforcement learning algorithms. It imitates a frozen lake with some icy patches and holes. The goal is to navigate an agent across the lake to a safe goal location without falling into a hole.

Gymnasium, previously known as OpenAI Gym, is a popular open-source Python library designed specifically for reinforcement learning.

The Frozen Lake is a 4x4 grid world.

The agent and environment interact at each of a sequence of discrete time steps: $t = 0, 1, 2, 3, ...$

Action space is a discrete, four-element set. At each time step agent takes one action from the action space ('left', 'down', 'right' or 'up'). **Action** indicates which direction to move the player. Possible actions are 0 (move left), 1 (move down), 2 (move right), and 3 (move up). A_t is a discrete random variable that takes values from action space. In Frozen Lake, A_t takes values 0, 1, 2 and 3. Numbers in Frozen Lake on picture 2.2 represent actions.

Observation space is a discrete set of all possible states. In Frozen Lake, it is a 16-element set shown in the picture 2.3. **States** at locations $(1,1), (1,2), ..., (4,5)$ are labeled as $0, 1, ..., 15$, respectively. S_t is a discrete random variable representing the state of an agent at the time step t. S_t takes values 0, 1, ..., 15.

Numbers in Frozen Lake on picture 2.3 represent states.

The episode ends if the agent moves into a hole (states 5, 7, 11, or 12) or the goal state (state 15).

The **environment** can be slippery or not, so the agent might not always move exactly in the intended direction. In the non-slippery environment, if the agent goes 'right' it ends up 'right'. That kind of environment is called **deterministic environment**. In the

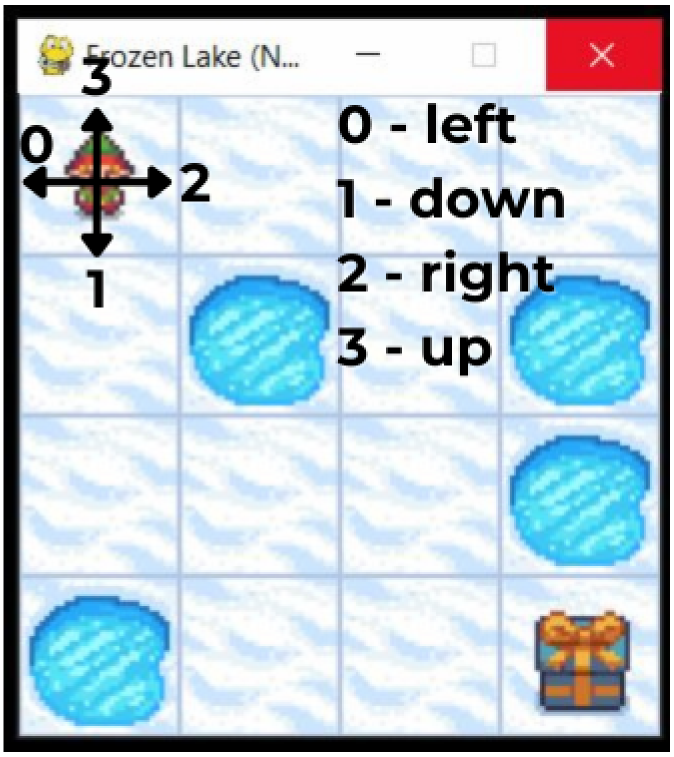


Figure 2.2: Frozen Lake environment with marked actions

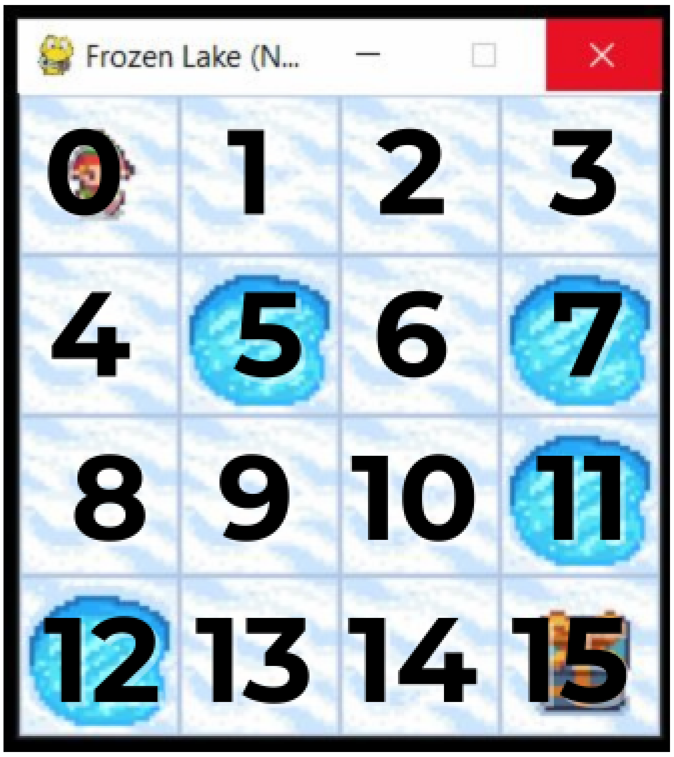


Figure 2.3: Frozen Lake environment with marked states

slippery environment, if the agent takes action 'right' it has the same probability of ending 'up', 'right', or 'down'. This adds complexity to the problem. That kind of environment is called **stohastic environment**

After agent takes the action a_t at a time step t from the state s_t, the environment sends to an agent a single number called the reward, denoted as r_t. In Frozen Lake agent gets a reward of +1 if it reaches the goal, and 0 otherwise (if it reaches the hole or frozen). R_t is a random variable representing the reward signal agent gets at the time step t.

The agent's sole objective is to maximize the total reward it receives over the long run, not just the immediate reward, and because of this, we introduce the concept of **return** (sometimes also called **gain**). Return (gain) at the time step t is a random variable that represents the sum of all rewards after taking action a_t and is defined as:

$$G_t = R_t + R_{t+1} + R_{t+2} + ... + R_T \tag{2.1}$$

where T is a final time step.

More used concept is **discounted return** defined as:

$$G_t = R_t + \gamma \dot{R}_{t+1} + \gamma^2 R_{t+2} + ... + \gamma^T R_T \tag{2.2}$$

where γ, $0 \leq \gamma \leq 1$, is a hyper-parameter called **discount rate**. The discount rate determines the present value of future rewards. If $\gamma = 0$ agent seeks immediate reward and as γ approaches 1, the return objective takes future rewards into account more strongly so the agent becomes more farsighted.

When interacting with the environment, agent makes a sequence or trajectory (history) that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{2.3}$$

In reinforcement learning tasks we want to find the strategy that maximizes expected return. That strategy is called **policy**. The policy π defines agent's way of behaving in a given state. The probability of agent taking action a given that it is in the state s is denoted as $\pi(a|s)$. For example, if an agent in Frozen Lake is in the state $s = 0$ and it doesn't prefer any action over the other, probabilities of agent taking actions 0, 1, 2 or 3 are equal. We write:

$$\pi(a = 0|s = 0) = \pi(a = 1|s = 0) = \pi(a = 2|s = 0) = \pi(a = 3|s = 0) = 0.25 \tag{2.4}$$

More generally, when the agent is in the state s, the probability of making the action a is denoted: $\pi(a|s)$.

The goal of RL is to learn policy π.

Value function $v_\pi(s)$ maps state space to a number that represents, roughly speaking, the gain (total amount of reward) an agent can expect to accumulate over the future, starting from that state s and following the policy π. The value function is defined as:

$$v_\pi(s) = E_\pi[G_t|S_t = s], \forall s \tag{2.5}$$

We seek actions that bring about states of the highest value, not the highest reward because these actions obtain the greatest amount of reward for us over the long run.

For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. The goal of reinforcement learning is to find optimal policy, more precisely policy that maximizes value in each state.

Similarly to value function $v_\pi(s)$, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action s, and thereafter following policy π:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{2.6}$$

We call $q_\pi(s, a)$ action-value fuction.

Known and unknown variables

In the beginning, in Frozen Lake example, we have a knowledge of:

- observation space (all states s): $s \in \{0, 1, ..., 15\}$
- action space (all states a): $a \in \{0, 1, 2, 3\}$ meaning left, down, right and up, respectively.

In the beginning, in Frozen Lake example, we **don't know**:

- optimal policy π
- value function v_π
- action-value function q_π

Our goal is to estimate optimal policy, value function or/and action-value function. We do that by placing an agent to interact with the environment. The agent can interact with environment $n = 10$ times and make $n = 10$ trajectories:

- first trajectory: $s_0 = 0, a_0 = 1, r_0 = 0, s_1 = 4, a_1 = 2, r_1 = 0, s_3 = 5$
- second trajectory: $s_0 = 0, a_0 = 2, r_0 = 0, s_1 = 4, a_1 = 1, r_1 = 0, ..., s_{12} = 12$
- ...
- tenth trajectory: $s_0 = 0, a_0 = 3, r_0 = 0, s_1 = 0, ..., s_{20} = 15$

We use trajectories to estimate the optimal policy, value function, or/and action value function. In the following chapters, we will discuss in detail how to do that.

Finite Markov Decision Process

In mathematics, a Markov decision process (MDP) is a discrete-time stochastic control process. It provides a **mathematical framework** for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker (in robotics called agent).

Markov decision process is a specific statistical framework that relaxes the conditions: the next state depends only on the current state and action and doesn't depend on any action or state before that. That assumption is called Markov property. Markov property could be written as 2.8:

$$\forall h_t = (s_0, a_1, s_1, ...s_{t-1}, a_t) : \tag{2.7}$$

$$P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t) = P(S_{t+1} = s_{t+1} | H_t = h_t, S_t = s_t, A_t = a_t) \tag{2.8}$$

where h_t is history or trajectory at time step t.

Thus, the next state s_{t+1} depends on the current state s_t and the agent's action a_t. But given s_t and a_t it is conditionally independent of all previous states and actions (MDP satisfy the Markov property): $s_0, a_0, s_1, a_1, s_2, ..., s_{t-1}, a_{t-1}$.

Because conditions are so relaxed, they rarely reflect the situation in the real world. We still use the MDP framework because it has good theoretical results, and even if it does not reflect the real world, it still gives good results when applied.

In a *finite* MDP, the sets of states, actions, and rewards (S, A, and R) all have a finite number of elements. In this case, the random variables R_t and S_t have well-defined discrete probability distributions dependent only on the preceding state and action.

Relationship Between State-Value Function $v_\pi(s)$, State-action value Function $q_\pi(s,a)$

Given the policy π, let's recall that definitions for state-value function $v_\pi(s)$ and action-value function $q_\pi(s,a)$ are:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] \\ q_\pi(s,a) &= E_\pi[G_t|S_t = s, A_t = a] \end{aligned} \tag{2.9}$$

and they represent the expected agent's gain G_t from the state s (state s and action a) if it follows that policy π.

Given policy π and all state-values $v_\pi(s)$, we can easily calculate all action-values $q_\pi(s,a)$ using

$$q_\pi(s,a) = \sum_{r,s'} p(r,s'|s,a)(r + \gamma v_\pi(s')) \tag{2.10}$$

because:

$$\begin{aligned} q_\pi(s,a) &= E_\pi[G_t|S_t = s, A_t = a] = \\ &= \sum_{r,s'} p(r,s'|s,a) E_\pi[G_t|S_t = s, A_t = a, R_t = r, S_{t+1} = s'] = \\ &= \sum_{r,s'} p(r,s'|s,a) E_\pi[r + \gamma G_{t+1}|S_t = s, A_t = a, R_t = r, S_{t+1} = s'] = \\ &= \sum_{r,s'} p(r,s'|s,a)(r + \gamma E_\pi[G_{t+1}|S_t = s, A_t = a, R_t = r, S_{t+1} = s']) = \\ &= \sum_{r,s'} p(r,s'|s,a)(r + \gamma E_\pi[G_{t+1}|S_{t+1} = s']) = \\ &= \sum_{r,s'} p(r,s'|s,a)(r + \gamma v_\pi(s')) \end{aligned} \tag{2.11}$$

Also, given policy π and all action-values $q_\pi(s,a)$, we can easily calculate all state-values $v_\pi(s)$ using

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \tag{2.12}$$

because:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] = \\ &= \sum_a \pi(a|s) E_\pi[G_t|S_t = s, A_t = a] = \\ &= \sum_a \pi(a|s) q_\pi(s,a) \end{aligned} \tag{2.13}$$

Chapter 3

Bellman Equations

In this chapter Bellman equation, the most famous equation (system of linear equations) in RL, is stated and discussed. Frozen Lake environment is used as an example to better understand Bellman equation.

Bellman Equation in Frozen Lake

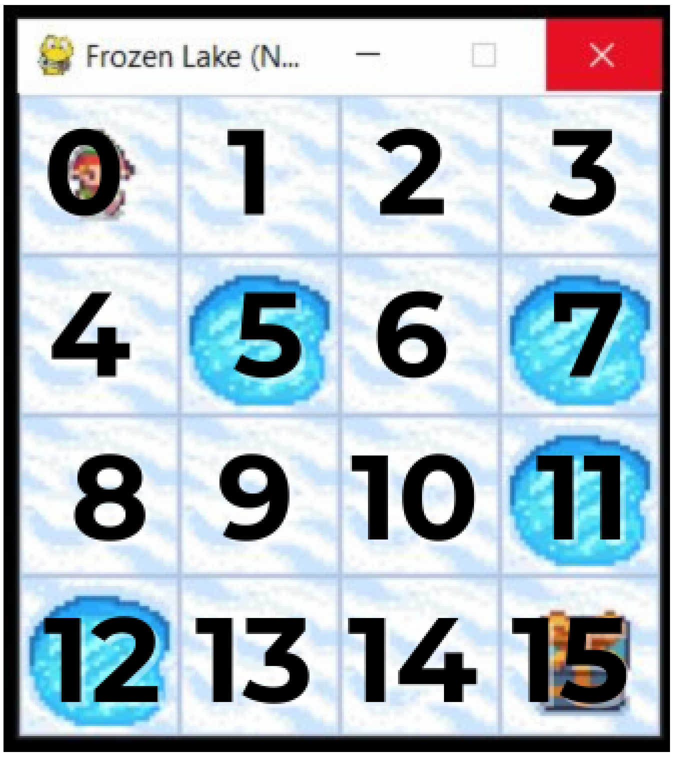


Figure 3.1: Frozen Lake environment with marked states

Numbers in Frozen Lake on picture 3.1 represent states.

Here we want to find a state-value function $v_\pi(s) = ?$ in all states $s \in 0, 1, 2, ..., 15$ in Frozen Lake if we have a policy, for example, random policy - we don't prefer any of four actions over the other, in any state: $\pi(a|s) = \frac{1}{4}, \forall a \in \{0, 1, 2, 3\}, \forall s \in \{0, 1, ..., 15\}$

To figure out general formula, let's find $v_\pi(0) = ?$

The dynamic of a system are probabilities $p(s', r|s, a), \forall s, a, s', r$ in Frozen Lake depends on whether the lake is slippery or not. Here, let's assume the lake isn't slippery and the system is determinant, so for the state $s = 0$ we have:

$$\begin{aligned}
p(s' = 0, r = 0 | s = 0, a = 0) = 1 \\
p(s' = 4, r = 0 | s = 0, a = 1) = 1 \\
p(s' = 1, r = 0 | s = 0, a = 2) = 1 \\
p(s' = 0, r = 0 | s = 0, a = 3) = 1
\end{aligned} \tag{3.1}$$

Now we have:

$$\begin{aligned}
v_\pi(0) &= E_\pi[G_t | S_t = 0] = \\
&= E_\pi[R_t + \gamma G_{t+1} | S_t = 0] = \\
&= \sum_a p(a|s) \cdot E_\pi[R_t + \gamma G_{t+1} | S_t = 0, A_t = a] = \\
&= \sum_a \pi(a|s) \cdot (E_\pi[R_t | S_t = 0, A_t = a] + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a] = \\
&= \sum_a \frac{1}{4} \cdot (0 + \gamma E_\pi[G_{t+1} | S_t = 0, A_t = a] = \\
&= \frac{1}{4} \cdot (E_\pi[G_{t+1} | S_t = 0, A_t = 0]) + \\
&+ \frac{1}{4} \cdot (E_\pi[G_{t+1} | S_t = 0, A_t = 1]) + \\
&+ \frac{1}{4} \cdot (E_\pi[G_{t+1} | S_t = 0, A_t = 2]) + \\
&+ \frac{1}{4} \cdot (E_\pi[G_{t+1} | S_t = 0, A_t = 3]) + \\
&= \frac{1}{4} \cdot v_\pi(0) + \\
&+ \frac{1}{4} \cdot v_\pi(4) + \\
&+ \frac{1}{4} \cdot v_\pi(1) + \\
&+ \frac{1}{4} \cdot v_\pi(0) = \\
&= \frac{1}{4}(v_\pi(0) + v_\pi(4) + v_\pi(1) + v_\pi(0))
\end{aligned} \tag{3.2}$$

We see that $v_\pi(0)$ depends on value-function in following states $v_\pi(0), v_\pi(4), v_\pi(1)$. Moreover, if we do the procedure above for all 16 states, we will end up with the system of 16 equations with 16 unknowns $v_\pi(0), v_\pi(1), ...v_\pi(15)$. There are numerous methods to calculate solutions we will present in the following sections. That system is called the **Bellman equation**.

Bellman Equation - Formulation

Let's recall that (discounted) gain in time step t is defined as:

$$G_t = R_t + \gamma \dot{R}_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + ... \tag{3.3}$$

where γ is given hyper-parameter from $[0, 1]$. Gain satisfy following recurrent relationship:

$$G_t = R_t + \gamma G_{t+1} \tag{3.4}$$

Also, let's recall that state-value function $v_\pi(s)$ in state s is defined as:

$$v_\pi(s) = E_\pi[G_t | S_t = s] \tag{3.5}$$

Now we can use 3.4 in 3.5:

$$\begin{aligned}
v_\pi(s) &= E_\pi[G_t|S_t = s] = \\
&= E_\pi[R_t + \gamma G_{t+1}|S_t = s] = \\
&= \sum_{a,s',r} p(a, s', r|s) \cdot E_\pi[R_t + \gamma G_{t+1}|S_t = s, A_t = a, S_{t+1} = s', R_t = r] = \\
&= \sum_{a,s',r} p(a, s', r|s) \cdot E_\pi[R_t + \gamma G_{t+1}|s, a, s', r] = \\
&= \sum_{a,s',r} p(a, s', r|s) \cdot (E_\pi[R_t|s, a, s', r] + E_\pi[\gamma G_{t+1}|s, a, s', r]) = \\
&= \sum_{a,s',r} p(a, s', r|s) \cdot (r + \gamma E_\pi[G_{t+1}|s']) = \\
&= \sum_{a,s',r} p(s', r|s, a)\pi(a|s) \cdot (r + \gamma v_\pi(s')) = \\
&= \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_\pi(s'))
\end{aligned} \tag{3.6}$$

Here we used Bayes' theorem and definition of a policy:

$$p(a, s', r|s) = p(s', r|s, a)p(a|s) = p(s', r|s, a)\pi(a|s) \tag{3.7}$$

and property of an expected value of an event Y: $E[Y] = \sum\limits_{x} p(x) \cdot E[Y|x]$ Also, we used Markov property.

Equation 3.6 is called **Bellman equation**:
$v_\pi(s) = \sum\limits_{a} \pi(a|s) \sum\limits_{s',r} p(s', r|s, a) \cdot (r + \gamma v_\pi(s')), \forall s$

It gives us the recurrent relationship between the current state s, and all possible combinations of actions a, following states s' and rewards r.

Now let's assume that **dynamic of a system** is given, more precisely we know all probabilities $p(s', r|s, a), \forall s, a, s', r$. Also, we have the policy $\pi(a|s), \forall s, a$

Because the Bellman equation is true for each state s, and policy π and probabilities $p(s', r|s, a)$ are given, we have *system of n equations* with n unknown variables, where n is the number of all possible states in the environment. To find a state-value function $v_\pi(s)$ we can 'just' solve that system of linear equations.

In practice, n is a large number, so solving a linear system of n linear equations with n unknowns is computationally expensive with a time complexity of $O(n^3)$. So we find solutions numerically. We will show how to do that in the following chapter 'Policy Evaluation'.

Chapter 4

Dynamic Programming

In this chapter we introduce the first set of algorithms for estimating the optimal policy called Dynamic Programming. DP algorithms assume knowledge of the dynamics of the environment (transition probabilities). DP algorithm is just a numerical solution of a system of n linear equations with n unknowns. Python code for Frozen Lake is listed and solutions are visualized and discussed.

What is Dynamic Programing?

Dynamic programming refers to algorithms used to find optimal policies that assumes complete knowledge of the environment as an MDP. In other words, DP algorithms have access to probabilities $p(s', r|s, a), \forall s, \forall a$. That is **not** the case in real-life scenarios, but let's pretend, for now, that is the case.

Policy Evaluation

Policy evaluation (prediction) is the process of estimating values v_π and q_π with random variables V_k and Q_k for given policy π. We evaluate how 'good' the policy π is.

The iterative algorithm given below with the recursive relationship 4.1 is doing *policy evaluation*. The algorithm uses Bellman equation 3.6 and with the assumption that the problem is finite MDP, values V_k or Q_k converge to real values v_π or q_π when $k \to \infty$.

$$\begin{aligned} &V_0(s) = 0, \forall s \\ &V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_k(s')), \forall s, k = 1, 2, 3, ..., K \end{aligned} \tag{4.1}$$

Here, we assume that the following values are given as an input:

1. policy $\pi(a|s), \forall a, \forall s$
2. dynamic of a system $p(r, s'|s, a), \forall s, \forall a, \forall r, \forall s'$
3. discounting factor γ from $[0, 1)$

Once needed precision is achieved in K iterations, we approximate $v_\pi \approx V_k$. Good precision is usually achieved by requesting that $|V_k(s) - V_{k+1}(s)| <= \theta, \forall s \in S$ where θ is hyperparameter, usually $\theta = 10^{-8}$.

Once v_π is estimated, we can estimate q_π using 2.10.

A similar thing can be done directly for state-action function:

$$\begin{aligned} Q_0(s, a) &= 0, \forall s, \forall a \\ Q_{k+1}(s, a) &= \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma v_k(s')), \forall s, \forall a, k = 1, 2, 3, ..., K \\ V_{k+1}(s) &= \sum_a \pi(a|s) Q_{k+1}(s, a), \forall s, k = 1, 2, 3, ..., K \end{aligned}$$

Goal of RL: Find Optimal Policy

In the previous section, we've learned how to calculate v_π and q_π when policy π is given. But remember, that is not the goal of RL. The goal of RL is to learn the best possible policy π. In this chapter, we define the optimal policy and present basic algorithms for finding it.

Optimal Policy

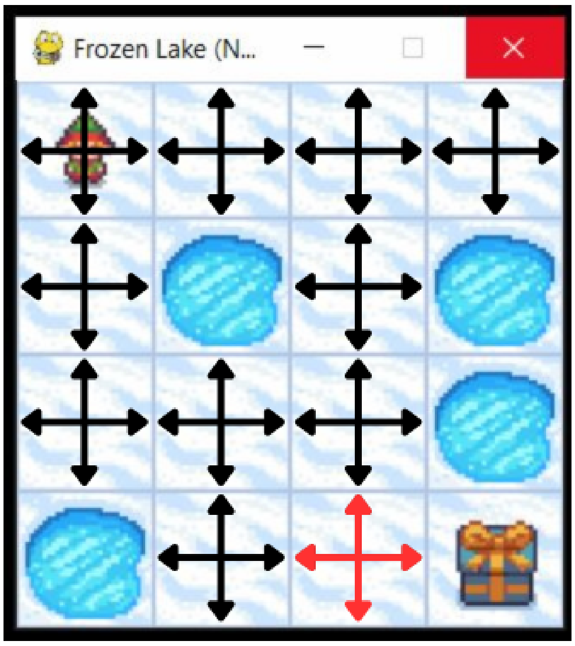


Figure 4.1: Random policy π_1

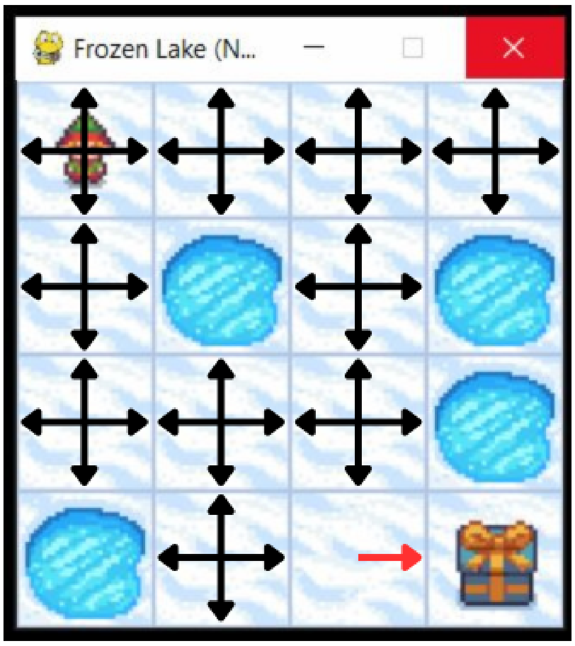


Figure 4.2: Policy π_2

Suppose in non-slippery Frozen Lake we have two policies π_1 and π_2 shown in the pictures above and defined as:

$$\pi_1(a|s) = \frac{1}{4}, \forall a \in \{0,1,2,3\}, \forall s \in \{0,1,...,15\} \tag{4.2}$$

$$\pi_1(a|s) = \begin{cases} 1, & for s = 14, a = 2 \\ 0, & for s = 14, a \in \{0,1,3\} \\ \frac{1}{4}, & otherwise \end{cases}$$

π_1 is random policy and π_2 differs just in $s = 14$. If following π_1 agent takes any action with the same probability from any state. If following π_2 agent takes any action with the same probability from any state except form state $s = 14$, when it (deterministically) goes to the right, and, by doing that receives reward $r = 1$ from the environment.

Which policy is better?

We intuitively know that policy π_2 is better than π_1. That is because when we calculate state-value functions following both policies they satisfy: $v_{\pi_1}(s) <= v_{\pi_2}(s), \forall s \in \{0,1,...,15\}$.

Using the iterative policy evaluation algorithm 4.1 from the previous chapter, we estimated state values function v_{π_1} for random policy π_1. The results are in picture 4.4 below.

We also estimated state values function v_{π_2} for random policy π_2. The results are in picture 4.6 below.

Solving RL problem means finding the policy that maximizes expected return.

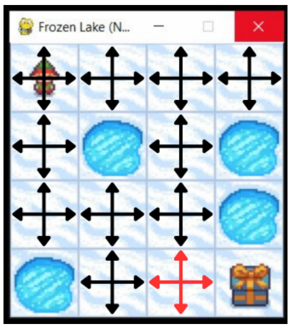
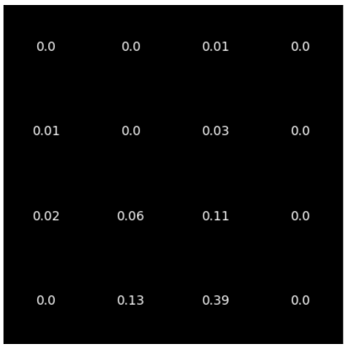


Figure 4.3: Random policy π_1



0.0	0.0	0.01	0.0
0.01	0.0	0.03	0.0
0.02	0.06	0.11	0.0
0.0	0.13	0.39	0.0

Figure 4.4: Estimated v_{π_1}

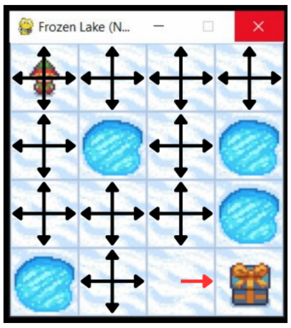
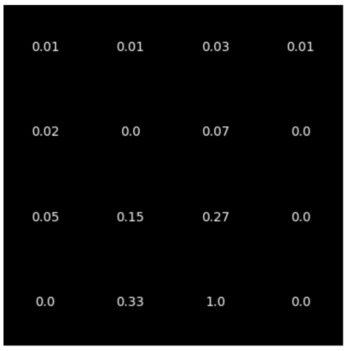


Figure 4.5: Random policy π_1



0.01	0.01	0.03	0.01
0.02	0.0	0.07	0.0
0.05	0.15	0.27	0.0
0.0	0.33	1.0	0.0

Figure 4.6: Estimated v_{π_1}

For finite MDPs, we can precisely define an optimal policy in the following way. If $v_{\pi_1}(s) <= v_{\pi_2}(s), \forall s$ then policy π_2 is better than π_1 which we write as $\pi_2 \geq \pi_1$. There is mathematical proof that in finite MDPs exists a policy that is better or equal to others. We call it an optimal policy. Although there may be more than one, we denote all the optimal policies by π^*. They share the same state-value function, called the optimal state-value function, denoted v_*, and defined as:

$$v_*(s) = \max_{\pi} v_\pi(s), \forall s \tag{4.3}$$

When optimal policy $\pi*$ shown in 4.7 is evaluated the state values 4.8 are obtained.

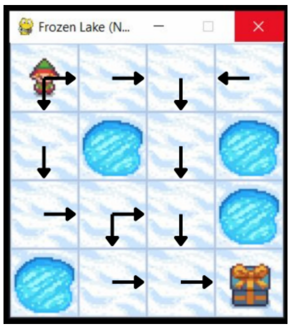
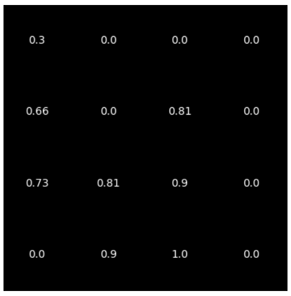


Figure 4.7: Optimal policy $\pi*$



0.3	0.0	0.0	0.0
0.66	0.0	0.81	0.0
0.73	0.81	0.9	0.0
0.0	0.9	1.0	0.0

Figure 4.8: $v*$

In the following chapters, we will present the most popular algorithms for finding optimal policies, such as the one in the picture 4.7.

Optimal policies also share the same optimal action-value function, denoted q_*, and defined as:

$$q_*(a|s) = \max_{\pi} q_\pi(a|s), \forall s, \forall a \tag{4.4}$$

Python code for DP policy evaluation

```
[label=python_policy_evaluation,
language=Python,
caption=DP Policy evaluation function,
frame=tb]

    def policy_evaluation(P: dict,
                          pi: np.ndarray = 0.25 * np.ones((16, 4)),
                          gamma: float = .9,
                          theta: float = 1e-8):
    '''
    :param P: transition probability matrix
    :param pi: policy we want to evaluate
    :param gamma: discount factor
    :param theta: dictate precision
    :return: state-value function for given policy pi
    '''

    v = np.zeros((16,))

    max_diff = 1e10
    while max_diff > theta:
        max_diff = 0
        for s in range(15):
            vs = 0
            for a in range(4):
                for prob, s_new, r, _ in P[s][a]:
                    vs += pi[s, a] * prob * (r + gamma * v[s_new])

            max_diff = max(abs(v[s] - vs), max_diff)
            v[s] = vs
    return v
```

Below in listing 4.1 is an example of how we use function **??** to evaluate $\pi*$ policy from the previous chapter.

Listing 4.1: Policy evaluation function

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt

# Non-slipery Frozen Lake
env = gym.make('FrozenLake-v1', desc=None,
map_name="4x4", is_slippery=False)
P = env.P  # transition matrix
env.close()

# defining optimal policy pi_star
pi_star = np.zeros((16, 4))
pi_star[0][1], pi_star[0][2], pi_star[9][1], pi_star[9][2] =
                                                    .5, .5, .5, .5
pi_star[1][2], pi_star[1][1], pi_star[3][0] = 1, 1, 1
pi_star[4][1], pi_star[6][1] = 1, 1
```

```
pi_star[8][2], pi_star[10][1] = 1, 1
pi_star[13][2], pi_star[14][2] = 1, 1

v_star = policy_evaluation(P, pi_star)
print(v_star)
```

Policy Improvement

Policy Improvement Theorem is a mathematical result that shows us how to improve a given (initial) policy π. If we have an arbitrary policy π and we calculate q_π using policy evaluation 4.1, we can improve that policy (making it a bit closer to the optimal policy) by:

$$\pi'(a|s) = \begin{cases} 1, & for a = \arg\max_a q(s,a) \\ 0, & otherwise \end{cases}$$

The theorem states that:

$$v_\pi(s) \leq v_{\pi'}(s), \forall s$$

Policy improvement is just a procedure of once improving policy π.

By knowing policy evaluation and policy improvement we can start with random policy and, iteratively, find optimal policy. Algorithm that does that is called **policy iteration** and is given in the following section.

Policy Iteration

Policy iteration (algorithm for estimating optimal policy) is done in the following steps:

1. Initialize policy π_0 (for example as random policy)
2. Find $v_{\pi_n}, \forall s$ iteratively in K steps, using policy evaluation 4.1:

$$V_0(s) = 0, \forall s$$
$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \cdot (r + \gamma v_k(s')), \forall s, k = 1,2,3,...K$$
$$v_{\pi_n} := V_K$$

3. Calculate $q_{\pi_n}, \forall a, \forall s$ using 2.10:

$$q_{\pi_n}(s,a) = \sum_{r,s'} p(r,s'|s,a)(r + \gamma v_{\pi_n}(s'))$$

4. Improve policy $\forall s$ using policy improvement:

$$\pi_{n+1}(a|s) = \begin{cases} 1, & for a = \arg\max_a q_{\pi_n}(s,a) \\ 0, & otherwise \end{cases}$$

5. do steps 2. to 4. for n=1, 2, ..., N

The presented algorithm 4 converges to optimal policy $\pi*$ when $n \to \infty$. In real-life scenarios, we say that policy converged when current v_{π_n} and next state value function $v_{\pi_{n+1}}$ are 'close to each other', more precisely when $|v_{\pi_n} - v_{\pi_{n+1}}| < \theta$. θ is hyperparameter, usually $\theta = 10^{-8}$.

In algorithm 4, we assume that the following values are given as input:

1. dynamic of a system $p(r, s'|s, a), \forall s, \forall a, \forall r, \forall s'$
2. discounting factor γ from $[0, 1]$
3. K (determine how good estimation v_{π_n} is for given π_n) or K
4. N (determine how 'close' estimation π_n is to π_*) or K

Once needed precision is achieved in N iterations, we approximate $\pi \approx \pi_n$.

Note that if in step 4. there is more than one action a that maximizes $q_k(s, a)$, we can choose the random one between them.

Python code for DP policy iteration

In 4.2 is listed a Python code for **DP policy iteration (using iterative policy evaluation) algorithm for estimating the optimal policy** in Frozen Lake:

Listing 4.2: DP policy iteration algorithm for estimating optimal policy

```
def q_value(P: dict, v: np.ndarray, s: int, gamma: float = 1):
    '''
    returns q values for each action for given state s
    '''
    q = np.zeros((4, ))
    for a in range(4):
        for prob, s_new, r, _ in P[s][a]:
            q[a] += prob * (r + gamma * v[s_new])
    return q


def policy_iteration(P: dict,
                     pi: np.ndarray = 0.25 * np.ones((16, 4)),
                     theta: float = 1e-8,
                     gamma: float = 0.9):
    converge = False
    q = np.zeros((16, 4))
    while not converge:
        # 1. policy evaluation
        v = policy_evaluation(P, pi, gamma=gamma, theta=theta)

        # 2. policy improvement
        pi_prim = np.zeros((16, 4))
        for s in range(15):
            q[s] = q_value(P, v, s, gamma)
            max_el = np.max(q[s])
            greedy_actions = []
            for a in range(4):
                if q[s][a] == max_el:
                    greedy_actions.append(a)
            pi_prim[s, greedy_actions] = 1 / len(greedy_actions)

        # 3. stop if pi converged
        if np.max(abs(policy_evaluation(P, pi)[0] -
            policy_evaluation(P, pi_prim)[0])) < theta * 1e2:
            converge = True

        # 4. Replace policy with new policy
```

Listing 4.3: Example of using DP policy iteration algorithm for estimating optimal policy

```
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
import gymnasium as gym


# Non-slipery Frozen Lake
env = gym.make('FrozenLake-v1', desc=None,
               map_name="4x4", is_slippery=False)
P = env.P  # transition matrix
env.close()

pi_random = 0.25 * np.ones((16, 4))
pi_dp = policy_iteration(P=P,
                         pi=pi_random,
                         theta=1e-8,
                         gamma=0.9)
plot_policy(pi_dp, 'Non-slipery Frozen Lake: DP policy')
```

```
        pi = copy.copy(pi_prim)

    return pi
```

In listing 4.3 is given code example that starts with random policy and uses DP policy iteration (4.2) to estimate optimal policy.

Output is optimal policy, as we can see in the figure. 4.9. Figure is made using function listed in 4.4.

Listing 4.4: DP policy iteration algorithm for estimating optimal policy

```
import matplotlib.pyplot as plt


def plot_policy(matrix: np.ndarray, text: str):
# Define actions and their corresponding arrow directions
actions = ['left', 'down', 'right', 'up']
dx = [-1, 0, 1, 0]
dy = [0, -1, 0, 1]

# Create a 4x4 plot
fig, axs = plt.subplots(4, 4, figsize=(12, 12))
fig.suptitle(f'{text}', ha='center', fontsize=16)

# Iterate through the matrix and plot arrows
for idx, cell in enumerate(matrix):
    i, j = divmod(idx, 4)
    max_prob = np.max(cell)
    for action, p in enumerate(cell):
        color = 'red' if p == max_prob else 'black'
        axs[i, j].arrow(0.5, 0.5, dx[action] * 0.2 * p,
            dy[action] * 0.2 * p, head_width=0.05,
                head_length=0.1, fc=color, ec=color)
```

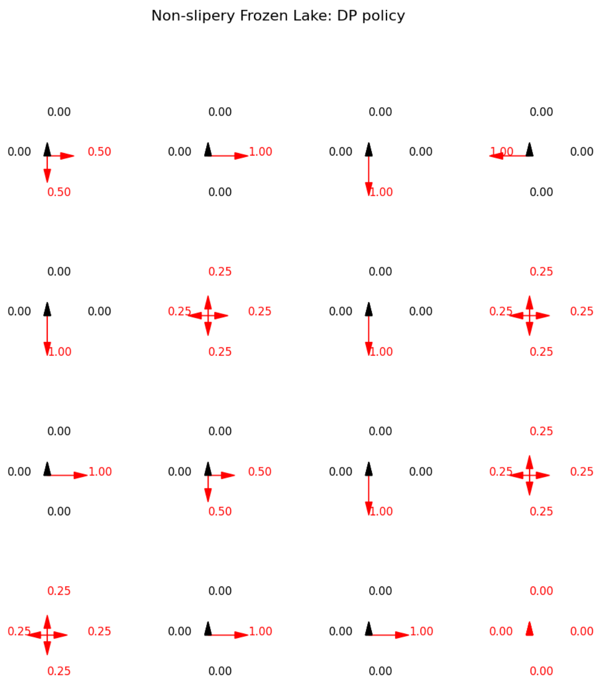


Figure 4.9: Optimal policy estimated using DP policy iteration

```
            axs[i, j].text(0.5 + dx[action] * 0.3, 0.5 +
                dy[action] * 0.3, f'{p:.2f}', color=color, fontsize=12)
        axs[i, j].set_xlim(0, 1)
        axs[i, j].set_ylim(0, 1)
        axs[i, j].axis('off')
    plt.show()
```

Generalized Policy Iteration (GPI)

Finding an optimal policy in policy iteration algorithm 4 was computationally expensive because, for each policy improvement, we have to, iteratively, find v_π. A better algorithm for finding an optimal policy is called **Generalized Policy Iteration (GPI)**. GPI refers to the general idea of letting policy-evaluation and policy improvement processes interact. GPI is broader class of functions and it can be used for non-DP algorithms.

Dynamic Programing General Policy Iteration (DP GPI)

GPI DP algorithm is:

1. Initialize policy π_0 (for example as a random policy)
2. Do the **single sweep** in policy evaluation 4.1:

$$V_{k+1}(s) := \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \cdot (r + \gamma V_k(s')) \tag{4.5}$$

3. Calculate $Q_{k+1}, \forall a, \forall s$ using 2.10:

$$Q_{k+1}(s, a) = \sum_{r,s'} p(r, s'|s, a)(r + \gamma V_{k+1}\pi(s'))$$

4. Improve policy $\forall s$:

5. do steps 2. to 4. for k=1, 2, 3...

Note that the only difference between GPI DP and DP policy iteration is in step 2.

Python code for DP GPI

In listing 4.5 is Python code for **DP GPI algorithm for estimating the optimal policy in Frozen Lake**. Note that the only difference between DP policy iteration in (4) and DP GPI (4.5) algorithm is in the step marked with 1.

Listing 4.5: DP GPI policy iteration algorithm for estimating optimal policy

```
def DP_GPI(P: dict,
           pi: np.ndarray = 0.25 * np.ones((16, 4)),
           theta: float = 1e-8,
           gamma: float = 0.9):
    converge = False
    q = np.zeros((16, 4))
    v = np.zeros((16,))
    max_diff = 0

    while not converge:
        # 1. single sweep in policy evaluation
        for s in range(15):
            vs = 0
            for a in range(4):
                for prob, s_new, r, _ in P[s][a]:
                    vs += pi[s, a] * prob * (r + gamma * v[s_new])
            max_diff = max(abs(v[s] - vs), max_diff)
            v[s] = vs

        # 2. policy improvement
        pi_prim = np.zeros((16, 4))
        for s in range(15):
            q[s] = q_value(P, v, s, gamma)
            max_el = np.max(q[s])
            greedy_actions = []
            for a in range(4):
                if q[s][a] == max_el:
                    greedy_actions.append(a)
            pi_prim[s, greedy_actions] = 1 / len(greedy_actions)

        # 3. stop if pi converged
        if max_diff < theta * 1e2:
            converge = True

        # 4. Replace policy with new policy
        pi = copy.copy(pi_prim)

    return pi
```

Dynamical Programming in practice

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. In practice, these methods usually converge much faster than their theoretical worst-case run times, particularly if they are started with good initial value functions or policies.

Chapter 5

Monte Carlo Mehod

In this chapter we give the second algorithm for estimating the optimal policy: Monte Carlo, that does not require knowledge of the dynamics of the environment. MC method just uses the definition of the state-action value function as expected value, and the basic statistical method (Monte Carlo method) to estimate the state-action value function. Once we have estimation of a state-value function, we can easily estimate optimal policy. Python code for Frozen Lake is listed, solutions are visualized and discussed.

What is Monte Carlo method?

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. For example Monte Carlo method can be used for numerical integration (to approximate integrals that cannot be solved analiticaly).

Here, the name "Monte Carlo Methods" is used for RL algorithms that use the idea that action-value function $q_\pi(s,a)$ is equal to the average of returns following that state-action pair, over infinitely many sampled trajectories. Bare in mind that Monte Carlo methods are widely used in statistics, and science, in general.

Monte Carlo Method

DP algorithms need knowledge of environment dynamics (transition matrix $p(r,s'|s,a)$), which is impossible to achieve in practice. Monte Carlo methods don't require environment dynamics. We say that MC methods are **model-free**.

Because we don't have access to a model, we cannot calculate $q_\pi(s,a)$ once we have $v_\pi(s)$ using $q_\pi(s,a) = \sum\limits_{r,s'} p(r,s'|s,a)(r+\gamma v_\pi(s'))$. To improve the given policy π we need access to $q_\pi(s,a)$ for each s and a. Because of that, when we use MC methods, we evaluate π using $q_\pi(s,a)$, not $v_\pi(s)$.

MC algorithms require the possibility to sample M trajectories

$$H_m = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_{T_m}, a_{T_m}, r_{T_m}), m = 1, 2, ...M$$

where M is a large number.

MC methods, just as DP methods, assume the **MDP framework**.

MC policy evaluation

Monte Carlo method use the definition

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$

and estimate $q_\pi(s,a)$ with $Q(s,a)$ that is arithmetic mean of gains $g_t = r_t + \gamma t_{t+1} + ...$ (when agent is in the state s_t and takes action a_t) over all sampled episodes.

When using Monte Carlo method we update $Q(s,a)$ for visited pairs (s,a) once the whole sampled episode is over. That is because we need to calculate the return g_t in the MC evaluation update rule:

$$Q'(s_t,a_t) = Q(s_t,a_t) + \alpha(G_t - Q(s_t,a_t)), \forall t = 1..T \tag{5.1}$$

MC GPI

Just like in DP methods, MC methods use the idea of GPI - they simultaneously do prediction (evaluation) and improvement of a policy. That means they simultaneously bring $Q_\pi(s,a)$ closer to real value $q_\pi(s,a)$ while improving policy π. Even done in parallel, these two processes approach the optimal values $q*(s,a)$ and $\pi*$.

1. **Policy evaluation step:** Once we have a policy π, we want to evaluate it, meaning we want to estimate action-values $q(s,a), \forall s, \forall a$. We denote that estimate $Q(s,a)$. In Monte Carlo evaluation to estimate $Q(s,a)$ we follow steps:

   (a) sample an episode $H_m = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_{T_m}, a_{T_m}, r_{T_m})$ following current policy π

   (b) for each state-action pair (s_t, a_t) update $Q(s_t,a_t)$ to be average across all returns accumulated during the first m simulations. It is not hard to see that the update rule will be:

   $$\begin{aligned} Q(s_t,a_t) &:= \frac{m-1}{m}Q(s_t,a_t) + \frac{1}{m}G_t = \\ &= Q(s_t,a_t) + \frac{1}{m}(G_t - Q(s_t,a_t)), \forall t = 1..T_m \end{aligned}$$

   where G_t is return in episode m at time step t.

   Here, we use first-time visit algorithm meaning that if specific state-action pair (s', a') is visited more then once in a single episode, we count only the first visit.

   The update rule above is often changed to have constant α instead of $\frac{1}{m}$, where α is hyperparameter:

   $$Q(s_t,a_t) := Q(s_t,a_t) + \alpha(G_t - Q(s_t,a_t)), \forall t = 1..T_m$$

2. **Policy improvement step:** Greedy improved policy $\pi*$ is defined as:

   $$\pi'(a|s) = \begin{cases} 1, & fora = \arg\max_a Q(s,a) \\ 0, & otherwise \end{cases}$$

   If improvement is greedy, using this method many state-action pairs won't be visited, and we cannot calculate the averages for those. Because of that, the concept of ϵ **greedy improved policy** is neglected, meaning that improved policy π' is defined as:

   $$\pi'(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{S}, & fora = \arg\max_a Q(s,a) \\ \frac{\epsilon}{S}, & otherwise \end{cases}$$

   where S is the number of states, and ϵ is hyperparameter. If $a = \arg\max_a Q(s,a)$ is true for more than one action a, choose action randomly.

Python code for First-Visit, on-policy, constant α, ϵ-greedy Monte Carlo control for estimating the optimal policy for Frozen Lake

Below is a Python code that uses **First-Visit, on-policy, constant α, ϵ-greedy Monte Carlo control for estimating the optimal policy**:

Listing 5.1: First-Visit, on-policy, constant α, ϵ-greedy Monte Carlo control for estimating the optimal policy for Frozen Lake

```
q = np.zeros((16, 4))
pi = .25 * np.ones((16, 4)) # random policy

for n_episode in range(1, N_episodes + 1):

    trajectory = sample_episode(pi=pi,
                     is_slippery_bool=is_slippery_bool)

    G = 0 # gain
    for step in reversed(range(len(trajectory.states))):
                                        # start from the last step
        s, a, r = trajectory.states[step],
                  trajectory.actions[step],
                  trajectory.rewards[step]

        first_visit = True
        for s_prev, a_prev in
                zip(trajectory.states[0:step-1],
                trajectory.actions[0:step-1]):
            if s_prev == s and a_prev == a:
                first_visit = False

        if first_visit:  # than update q(s,a)
                         # else skip pair (s,a)
            G = r + gamma * G
            q[s, a] += (G - q[s, a]) / n_episode

            a_star = np.max(q[s, :])
            greedy_actions = []
            for i in range(4):
                if q[s, i] == a_star:
                    greedy_actions.append(i)
            greedy_action = random.choice(greedy_actions)
            pi[s, :] = epsilon / 4
            pi[s, greedy_action] += 1 - epsilon
pi_out = policy2greedy(pi=pi)
return pi_out, q
```

We sampled a single episode in 5.1 using a given policy with the function sample_episode. Code for sample_episode is listed in 5.2.

Listing 5.2: Sample an episode in Frozen Lake

```
from collections import namedtuple
```

```
import gymnasium as gym


episode = namedtuple('episode',
                     'states, actions, rewards, terminated, truncated')


def sample_episode(pi: np.ndarray = 0.25 * np.ones((16, 4)),
                   is_slippery_bool: bool = True) -> episode:
    '''
    function simulates a single episode
    of a game following the given policy pi

    INPUTS
    pi: (16, 4) numpy array
        - pi[s, a] is probability of agent
          taking action a given he is in state s
        - for given s: we choose actions 0, 1, 2, 3
          with probabilities pi[s, 0], pi[s, 1], pi[s, 2], pi[s, 3],
          respectively

    OPUTPUTS:
    episode: named tuple that collects trajectory information
    {s0, a0, r0, ..., s_T, A_T, R_T}
        - single episode has form:
      episode(states=[0, 1, 2], actions=[1, 2, 3],
          rewards=[0, 0, 1], terminated=True, truncated=False)
    '''
    env = gym.make('FrozenLake-v1', desc=None,
    map_name="4x4", is_slippery=is_slippery_bool)

    s, info = env.reset(seed=42)
    terminated, truncated = False, False
    states, actions, rewards = [], [], []
    while not truncated and not terminated:
        a = np.random.choice([0, 1, 2, 3], p=pi[s, :])
        s_new, r, terminated, truncated, _ = env.step(a)
        states.append(s)
        actions.append(a)
        rewards.append(r)
        s = s_new
    env.close()
    return episode(states, actions, rewards, terminated, truncated)
```

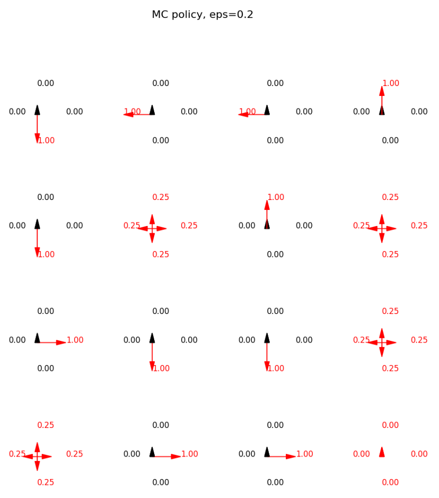
ϵ hyperparameter

Results of MC estimation for different ϵ values are shown on figures 5.1, 5.2, 5.3 and 5.4.

ϵ hyperparameter dictates how much agent explore new knowledge and exploit previous knowledge.

1. $\epsilon = 0$ - agent 100% times exploit (takes the best action given by estimated policy π). This is greedy policy improvement.

2. $\epsilon = 0.4$ - agent 60% times exploit (takes the best action given by estimated policy π) and 40% times explore (takes random action)



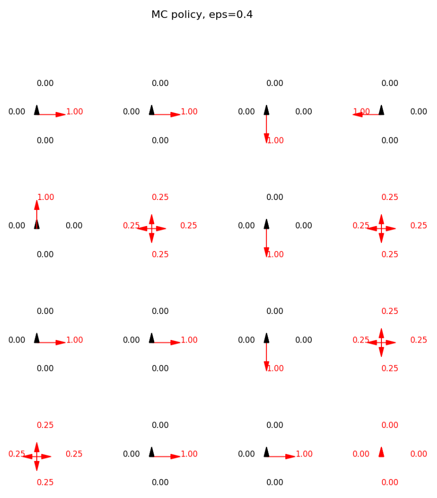


Figure 5.1: MC estimation of optimal policy $\epsilon = 0.2$

Figure 5.2: MC estimation of optimal policy $\epsilon = 0.4$

3. $\epsilon = 1$ - agent 100% explore (takes random action)

ϵ should be high at the beginning, meaning that the agent should explore more at the beginning. As the agent gains experience, ϵ should be lower and agent should exploit previous experience. Algorithm 5.1 can be adjusted accordingly.

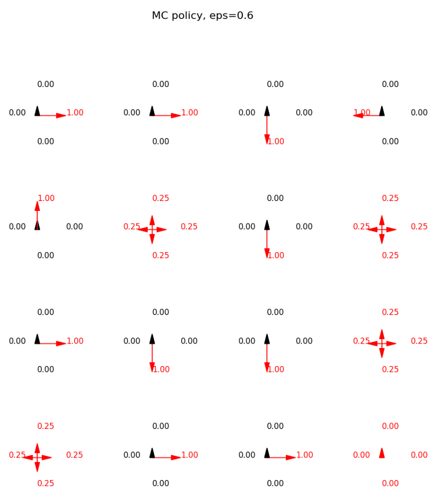


Figure 5.3: MC estimation of optimal policy $\epsilon = 0.6$

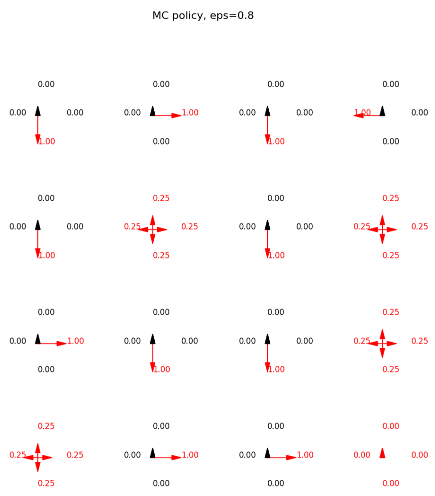


Figure 5.4: MC estimation of optimal policy $\epsilon = 0.8$

Chapter 6

Temporal Difference (TD)

In this chapter we give the third set of algorithms for estimating the optimal policy that (just like MC) does not require knowledge of the dynamics of the environment. TD methods use estimates to estimate (they bootstrap) state-action value functions. They combine DP and MC methods in that sense. There are a few versions of TD algorithm for estimating the optimal policy and the most popular are: SARSA (one-step TD), n-step SARSA (n-step TD), expected SARSA (one-step TD) and Q-learning (one-step TD). Python code for all of them for Frozen Lake is listed, solutions are visualized and discussed.

DP, MC and TD: similarities and differences in policy evaluation step

Monte Carlo method use the definition

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$

and estimate $q_\pi(s,a)$ with $Q(s,a)$ that is arithmetic mean of gains $g_t = r_t + \gamma t_{t+1} + ...$ (when agent is in the state s_t and takes action a_t) over all sampled episodes.

When using Monte Carlo method we update $Q(s,a)$ for visited pairs (s,a) once the whole sampled episode is over. That is because we need to calculate the return g_t in the MC evaluation update rule:

$$Q'(s_t,a_t) = Q(s_t,a_t) + \alpha(G_t - Q(s_t,a_t)), \forall t = 1..T \tag{6.1}$$

DP method use Bellman's equation

$$q_\pi = \sum_{r,s'} p(r,s'|s,a) \cdot \big(r + \gamma v_\pi(s')\big)$$

where transition probabilities $p(r,s'|s,a)$ are known. The only unknown in the previous equation is $v_\pi(s')$. We estimate $v_\pi(s')$ with $V(s')$ from the previous step. That estimation is noted Q'_π and DP update rule is:

$$Q'_\pi = \sum_{r,s'} p(r,s'|s,a) \cdot \big(r + \gamma V(s')\big)$$

In **Temporal Difference (TD) learning** there is just one difference from Monte Carlo method - at the time t we estimate the value of gain G_t instead of waiting for the end of the episode to get access to it.

For TD we use following:

$$\begin{aligned} q_\pi(s_t,a_t) &\overset{\text{def}}{=} E_\pi[G_t|S_t = s_t, A_t = a_t] = \\ &= E_\pi[R_t + \gamma G_{t+1}|S_t = s_t, A_t = a_t] = \\ &= E_\pi[R_t + \gamma E_\pi[G_{t+1}|S_{t+1}, A_{t+1}]|S_t = s_t, A_t = a_t] = \\ &= E_\pi[R_t + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s_t, A_t = a_t] \end{aligned} \tag{6.2}$$

Remember that $q_\pi(S_{t+1}, A_{t+1})$ is unknown and in TD we estimate it with $Q(S_{t+1}, A_{t+1})$ from previous step, just like in DP method. We sample episodes and estimate $q_\pi(s_t, a_t)$ with $Q(s_t, a_t$ that is arithmetic mean of $r_t + \gamma Q(s_{t+1}, a_{t+1})$.

Let's say we have information about first $t+1$ time steps in current episode $H_{1:(t+1)} = (s_0, a_0, r_0, ...s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1})$. **One-step TD update rule** in time-step t would be:

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha((r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)) \tag{6.3}$$

To get a better approximation than one-step TD update rule 6.3, at the time t we can use information about the next two steps (instead of one): $a_{t+1}, r_{t+1}, s_{t+2}, a_{t+2}, r_{t+2}, s_{t+3}$. That is how we get **2-step TD update rule**:

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma r_{t+1} + \gamma^2 Q(s_{t+2}, a_{t+2}) - Q(s_t, a_t)), \forall t = 1..T-2$$

because:

$$\begin{aligned} G_t &\overset{\text{def}}{=} R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... + \gamma^T r_T = \\ &= R_t + \gamma R_{t+1} + \gamma^2 G_{t+2} \approx \\ &\approx r_t + \gamma r_{t+1} + \gamma^2 Q(s_{t+2}, a_{t+2}) = \end{aligned}$$

n-step TD update rule is:

$$\begin{aligned} Q'(s_t, a_t) = Q(s_t, a_t) + \\ \alpha(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_n + \gamma^{n+1} Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t)), \\ \forall t = 1..T-n \end{aligned}$$

TD and Bootstrapping

TD methods update their estimates based in part on other estimates. They learn a guess from a guess—they **bootstrap**.

For example, let's look at one-step TD update rule $Q'(s_t, a_t) = Q(s_t, a_t) + \alpha((r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t))$. We update $Q'(s_t, a_t)$- the estimation of $q_\pi(s_t, a_t)$, and that estimate is based on other estimates:

1. $Q(s_{t+1}, a_{t+1})$- estimate of $q_\pi(s_{t+1}, a_{t+1})$ and $r_t + \gamma Q(s_{t+1}, a_{t+1})$- estimate of G_t.

Mathematical Proof of TD Learning Method

For any fixed policy π, TD(0) has been proved to converge to v_π for a constant step-size parameter α if it is sufficiently small.

One-step TD Algorithm for Estimating the Optimal Policy

Just like in DP and MC control methods, one-step TD control methods for estimating the optimal policy consist of two steps:

1. **Policy evaluation step:** Once we have an improved policy π, we want to evaluate it, meaning we want to estimate action-values $q_\pi(s, a), \forall s, \forall a$ based on the previous estimations $Q(s, a), \forall s, \forall a$. We denote the updated estimate $Q'(s, a)$. In one-step TD evaluation to find $Q'(s, a)$ we follow steps:

   (a) from starting state s_0 sample action a_0 following policy π and observe r_0 and s_1. Then, from state s_1 sample action a_1 following policy π and observe r_1 and s_2. We now have first part of trajectory: $(s_0, a_0, r_0, s_1, a_1, r_1, s_2)$

(b) for state-action pair (s_0, a_0) update $Q(s_0, a_0)$:

$$Q'(s_0, a_0) := Q(s_0, a_0) + \alpha(r_0 + \gamma Q(s_1, a_1) - Q(s_0, a_0))$$

(c) from state s_2 sample action a_2 following policy π and observe r_2 and s_3. Now known part of an episode is: $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, s_3)$

(d) for state-action pair (s_2, a_2) update $Q(s_2, a_2)$:

$$Q'(s_1, a_1) = Q(s_1, a_1) + \alpha\left((r_1 + \gamma Q(s_2, a_2) - Q(s_1, a_1)\right)$$

(e) from state s_3 sample action a_3 following policy π and observe r_3 and s_4. Now known part of an episode is: $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_4)$

(f) for state-action pair (s_3, a_3) update $Q(s_3, a_3)$:

$$Q'(s_2, a_2) = Q(s_2, a_2) + \alpha\left((r_2 + \gamma Q(s_3, a_3) - Q(s_2, a_2)\right)$$

(g) ... repeat until the end of an episode.

2. **Policy improvement step:** Once again we use ϵ **greedy policy improvement** where improved policy π' is defined as:

$$\pi'(a|s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{S}, & for a = \arg\max_a Q(s,a) \\ \dfrac{\epsilon}{S}, & otherwise \end{cases}$$

where S is the number of states, and ϵ is hyperparameter. If $a = \arg\max_a Q(s,a)$ is true for more then one action a, choose action randomly.

These two steps follow the pattern of **generalized policy iteration (GPI)**.

SARSA, Expected SARSA and Q-learning

Note that there are a few different ways on how to do policy evaluation step in TD methods. The most famous ones are SARSA, Q-learning, and Expected SARSA. Update formula in the policy evaluation step for those are:

1. SARSA (On-policy one-step TD Control):

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[(r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)]$$

2. n-step SARSA (On-policy n-step TD Control):

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t)]$$

3. Q-Learning (Off-policy TD control):

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)]$$

4. Expected SARSA:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[(r_t + \gamma \sum_{a_{t+1}} \pi(a_{t+1}, s_{t+1}) Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Which method converge faster?

Until now, no one has been able to prove mathematically that one method, out of MC and TD, converges faster than the other. In practice, however, TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks.

Python code for one-step TD control methods: SARSA, Expected SARSA, and Q-learning

In 6.1 is given function that return estimation of optimal policy that follows One-step TD control methods: SARSA and Expected SARSA.

Listing 6.1: One-step TD control methods: SARSA and Expected SARSA

```
from utils import *


def sarsa_control(expected_sarsa: bool = False,
                  N_episodes: int = 1000,
                  alpha: float = 0.1,
                  epsilon: float = 0.1,
                  gamma: float = 0.9,
                  is_slippery_bool: bool = False):
    q = np.zeros((16, 4))
    pi = .25 * np.ones((16, 4))
    env = gym.make('FrozenLake-v1', desc=None, map_name="4x4", is_slippery
    s, info = env.reset(seed=42)

    for n_episode in range(1, N_episodes + 1):
        a = np.random.choice([0, 1, 2, 3], p=pi[s, :])
        terminated, truncated = False, False

        while not terminated and not truncated:
            # take action a and observe reward r,
            # following state s_new (terminated, truncated)
            s_new, r, terminated, truncated, _ = env.step(a)

            # sample action a_new from s_new following current policy
            a_new = np.random.choice([0, 1, 2, 3], p=pi[s_new, :])

            if not expected_sarsa:
                q[s, a] += alpha * (r + gamma * q[s_new, a_new] - q[s, a])
            else:
                expected_q = 0
                for action in range(4):
                    expected_q += pi[s_new, action] * q[s_new, action]
                q[s, a] += alpha * (r + gamma * expected_q - q[s, a])

            # update policy pi for state s
            a_star = np.max(q[s, :])
            greedy_actions = []
            for i in range(4):
                if q[s, i] == a_star:
                    greedy_actions.append(i)
            greedy_action = random.choice(greedy_actions)
            pi[s, :] = epsilon / 4
            pi[s, greedy_action] += 1 - epsilon

            s = s_new
            a = a_new
        s, info = env.reset()

    env.close()
    pi = policy2greedy(pi)
    return pi
```

We use 6.1 in 6.2 and get the estimations of optimal policy given in 6.1 and 6.2.

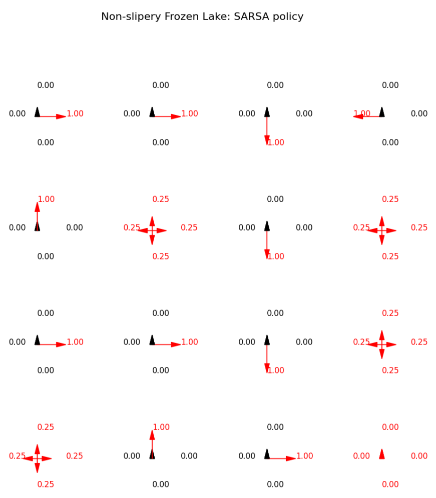


Figure 6.1: SARSA estimation of optimal policy

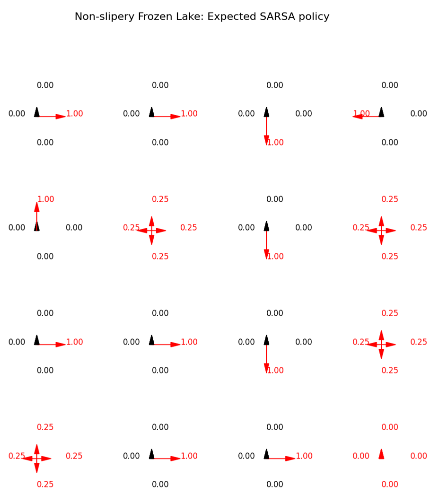


Figure 6.2: Expected SARSA estimation of optimal policy

Listing 6.2: Usage of One-step TD control method: Q-learning

```
# SARSA Non-slippery Frozen Lake

pi_sarsa = sarsa_control(expected_sarsa=False,
                         N_episodes=10000,
                         epsilon=0.1,
                         is_slippery_bool=False)
plot_policy(pi_sarsa,
            'Non-slipery Frozen Lake: SARSA policy')
mean_return = calculate_mean_return(pi=pi_sarsa,
              N_runs=50000, gamma=0.9, is_slippery_bool=False)
print(f'Non-slipery Frozen Lake:  SARSA {mean_return=}')


# Expected SARSA Non-slippery Frozen Lake

pi_exp_sarsa = sarsa_control(expected_sarsa=True,
        N_episodes=10000, epsilon=0.1, is_slippery_bool=False)
plot_policy(pi_exp_sarsa,
        'Non-slipery Frozen Lake: Expected SARSA policy')
mean_return = calculate_mean_return(pi=q_exp_sarsa,
        N_runs=50000, gamma=0.9, is_slippery_bool=False)
print(f'Non-slipery Frozen Lake:  Expected SARSA {mean_return=}')
```

In 6.3 is given Python function that returns estimation of optimal policy that follows

One-step TD control method Q-learning.

Listing 6.3: One-step TD control method: Q-learning

```
def q_learning(N_episodes: int = 1000,
               alpha: float = 0.01, epsilon: float = 0.1, gamma: float = 0.9,
               is_slippery_bool: bool = False):
    q = np.zeros((16, 4))
    pi = .25 * np.ones((16, 4))
    env = gym.make('FrozenLake-v1', desc=None, map_name="4x4", is_slippery=is_
    s, info = env.reset(seed=42)

    for n_episode in range(1, N_episodes + 1):
        terminated, truncated = False, False

        while not terminated and not truncated:
            # sample action a from state s following policy pi
            a = np.random.choice([0, 1, 2, 3], p=pi[s, :])

            # take action a and observe reward r, following state s_new (termi
            s_new, r, terminated, truncated, _ = env.step(a)

            # update q[s, a]
            q_max = np.max(q[s_new, :])
            greedy_actions = []
            for action in range(4):
                if q[s_new, action] == q_max:
                    greedy_actions.append(action)
            a_new_star = random.choice(greedy_actions)
            q[s, a] = q[s, a] + alpha * (r + gamma * q[s_new, a_new_star] - q[

            # update policy pi for state s
            a_star = np.max(q[s, :])
            greedy_actions = []
            for action in range(4):
                if q[s, action] == a_star:
                    greedy_actions.append(action)
            greedy_action = random.choice(greedy_actions)
            pi[s, :] = epsilon / 4
            pi[s, greedy_action] += 1 - epsilon

            s = s_new

        s, info = env.reset()

    env.close()
    pi = policy2greedy(pi)
    return pi
```

We use 6.3 in 6.4 and get the estimations of optimal policy given in 6.3.

Listing 6.4: Usage of One-step TD control method: Q-learning

```
# Q-learning Non-slippery Frozen Lake

pi_q_learning = q_learning(N_episodes=10000, is_slippery_bool=False)
plot_policy(pi_q_learning, 'Non-slippery Frozen Lake: Q-Learning policy')
```

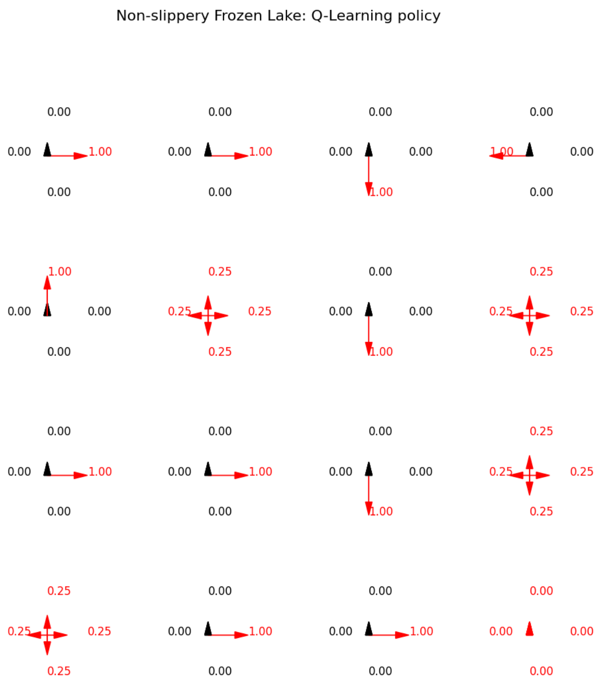


Figure 6.3: Q-learning estimation of optimal policy

```
mean_return = calculate_mean_return(pi_q_learning, N_runs=50000, is_slippe
print(f'Non-slipery Frozen Lake: Q-Learning {mean_return=}')
```

Bibliography

[1] Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine **34**(6), 26–38 (2017). https://doi.org/10.1109/MSP.2017.2743240

[2] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018), `http://incompleteideas.net/book/the-book-2nd.html`