12 Reinforcement learning

This chapter covers

- Grasping the fundamental principles underlying reinforcement learning
- Understanding the Markov decision process
- Comprehending the actor-critic architecture and proximal policy optimization
- Getting familiar with noncontextual and contextual multi-armed bandits
- Applying reinforcement learning to solve optimization problems

Reinforcement learning (RL) is a powerful machine learning approach that enables intelligent agents to learn optimal or near-optimal behavior through interacting with their environments. This chapter dives into the key concepts and techniques within RL, shedding light on its underlying principles as essential background knowledge. Following this theoretical exposition, the chapter will proceed to illustrate practical examples of employing RL strategies to tackle optimization problems.

12.1 Demystifying reinforcement learning

Reinforcement learning (RL) is a subfield of machine learning that deals with how an agent can learn to make decisions and take actions in an environment to achieve specific goals following a trial-and-error learning approach. The core idea of RL is that the agent learns by interacting with the environment, receiving feedback in the form of rewards or penalties as a result of its actions. The agent's objective is to maximize the cumulative reward over time.

Reinforcement learning

"Reinforcement learning problems involve learning what to do—how to map situations to actions—so as to maximize a numerical reward signal" (Richard Sutton and Andrew Barto, in their book *Reinforcement Learning* [1]).

Figure 12.1 outlines the common RL algorithms found in the literature. This classification divides RL problems into two main categories: Markov decision process (MDP) problems and multi-armed bandit (MAB) problems. The distinction between the two lies in how the agent's actions interact with and affect the environment.

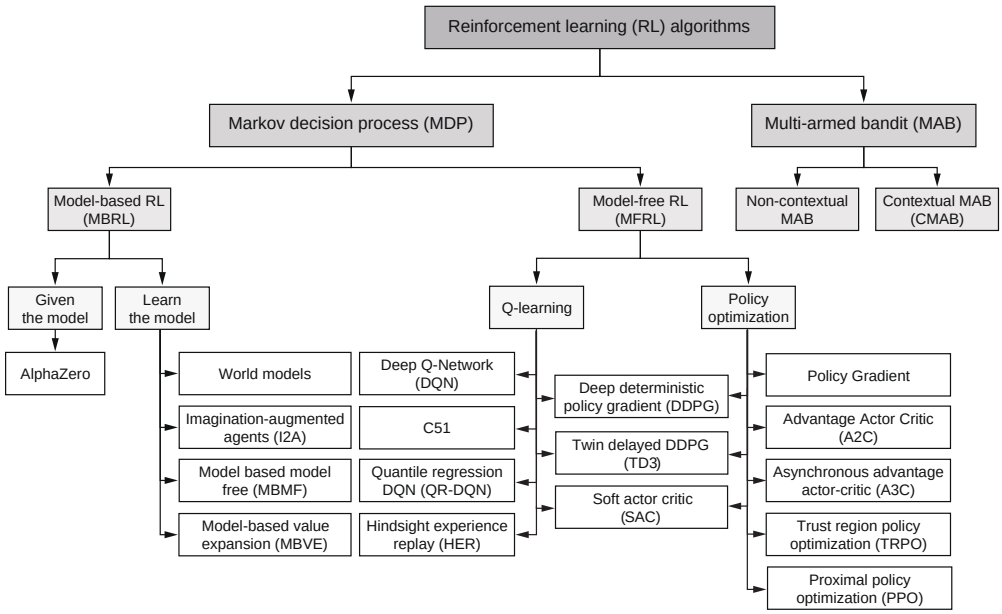


Figure 12.1 Reinforcement learning algorithm taxonomy

In MDP-based problems, the agent's actions influence the environment, and the agent must consider the consequences of its actions over multiple time steps, incorporating the notion of states and transitions. MAB problems, on the other hand, involve scenarios

where the agent faces a series of choices (arms) and aims to maximize the cumulative reward over time. Such problems are often used when there is no explicit state representation or long-term planning required. In contextual multi-armed bandit (CMAB) problems, the agent is presented with context or side information that is used to make more informed decisions. The expected reward of an arm (an action) is a function of both the action and the current context. This means the best action can change depending on the provided context. It is worth mentioning that MDPs are a more comprehensive framework that accounts for dynamic decision-making in a broader range of situations.

Before we explore how reinforcement learning can be applied to solve optimization problems, it is essential to understand several relevant reinforcement learning techniques. The following subsections will provide a detailed overview of these methods, and the subsequent sections will demonstrate their use in addressing optimization problems.

12.1.1 Markov decision process (MDP)

The purpose of learning is to form an internal model of the external world. This external world, or *environment*, can be abstracted using deterministic or nondeterministic (stochastic) models.

Consider a situation where you're planning to commute from your initial state (e.g., your home) to a designated goal state (e.g., your workplace). A deterministic path-planning algorithm such as A* (discussed in chapter 4) might provide you with multiple options: taking the train, which would take about 1 hour and 48 minutes; driving by car, which could take about 1 hour and 7 minutes; or biking, which could take about 3 hours and 16 minutes (figure 12.2a). These algorithms operate under the assumption that actions and their resulting effects are entirely deterministic.

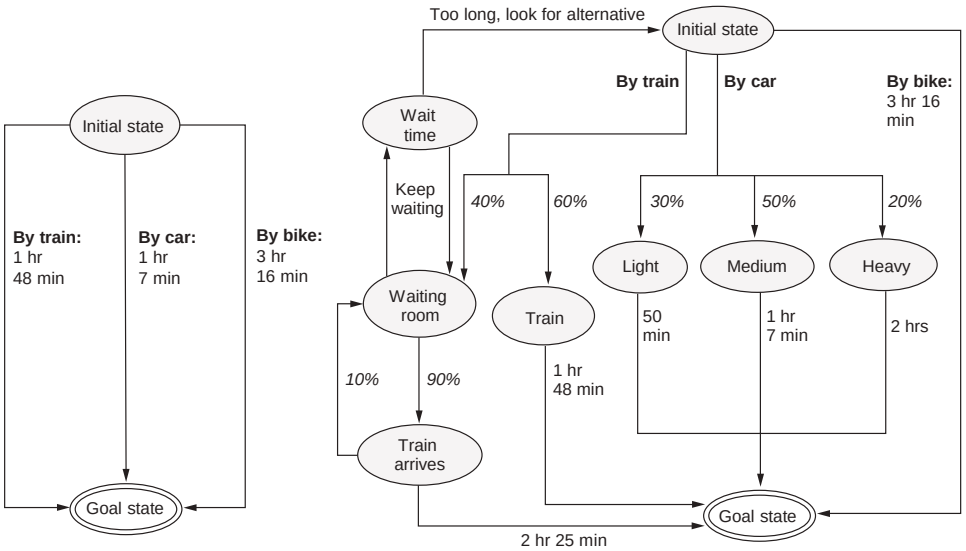


Figure 12.2 Deterministic vs. nondeterministic state transition models

However, if uncertainties come into play during your journey planning, you should resort to stochastic planning algorithms that operate on nondeterministic state transition models to enable planning under uncertainty. In these scenarios, transition probabilities between states become an integral part of your decision-making process.

For instance, let's consider the initial state (your home), as shown in figure 12.2b. If you choose to commute by train, there's a 60% chance that you'll be able to catch the train on time and reach your destination (your workplace) within the expected 1 hour and 48 minutes. However, there's a 40% chance that you could miss the train and need to wait for the next one. If you do end up waiting, there's a 90% chance the train will arrive on time and you'll catch it and arrive at your destination within a total time of 2 hours and 25 minutes. On the other hand, there's a 10% chance the train does not arrive, leading to an extended wait or even having to look for an alternative. On the other hand, if you choose to drive, there's a 30% chance that you'll encounter light traffic and reach your office in just 50 minutes. However, there's also a 50% likelihood of medium traffic delaying your arrival to 1 hour and 7 minutes. In the worst-case scenario, there's a 20% chance that heavy traffic could extend your travel time to 2 hours. If you choose to bike, the estimated travel time of 3 hours and 16 minutes is more predictable and less subject to change.

This scenario describes an environment that is fully observable and where the current state and actions taken completely determine the probability distribution of the next state. This is called a Markov decision process (MDP).

12.1.2 From MDP to reinforcement learning

MDP provides a mathematical framework for planning under uncertainty. It is used to describe an environment for reinforcement learning, where an agent learns to make decisions by performing actions and receiving rewards. The learning process involves trial and error, with the agent discovering which actions yield the highest expected cumulative reward over time.

As shown in figure 12.3, the agent interacts with the environment by being in a certain state $s_t \in S$, taking an action $a_t \in A$ at time t based on an observation o and by applying policy π, and then receiving a reward $r_t \in R$ and transitioning to a new state $s_{t+1} \in S$ according to the state transition probability T.

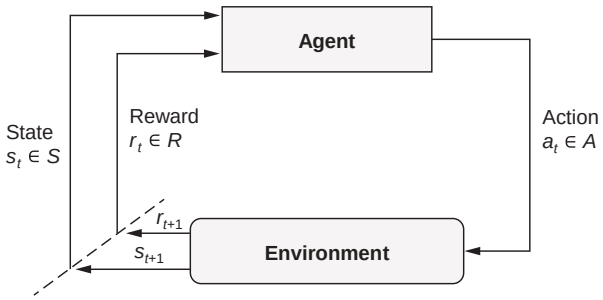


Figure 12.3 An agent learns through interaction with the environment.

The following terms are commonly used in RL:

- *A state,* s—This represents the complete and unfiltered information about the environment at a particular time step. An observation o is the partial or limited information that the agent can perceive from the environment at a given time step. When the agent is able to observe the complete state of the environment, we say that the environment is fully observable and can be modeled as an MDP. When the agent can only see part of the environment, we say that the environment is partially observable, and it should be modeled as a *partially observable Markov decision process* (POMDP).
- *An action set,* A—This represents the set of possible or permissible actions that an agent can take in a given environment. These actions can be discrete, like in the case of board games, or continuous, like in the case of robot controls or the lane-keep assist of an assisted or automated driving vehicle.
- *The policy*—This can be seen as the agent's brain. It is the decision-making strategy of the agent or the mapping from states or observations to actions. This policy can be deterministic, usually denoted by μ, or stochastic, denoted by π. A stochastic policy π is mainly the probability of selecting an action $a \in A$ given a certain state $s \in S$. A stochastic policy can also be parameterized and denoted by π_θ. This parameterized policy is a computable function that depends on a set of parameters (e.g., the weights and biases of a neural network), which we can adjust to change the behavior via an optimization algorithm.
- *A trajectory, τ (aka episode or rollout)*—This is a sequence of states and actions in the world, $\tau = (s_0, a_0, s_1, a_1, \ldots)$.
- *The expected return*—This refers to the cumulative sum of rewards that an agent can expect to receive over a future time horizon. It is a measure of the overall desirability or value of a particular state-action sequence or policy.

Let's consider a simple Reach the Treasure game where an agent tries to get a treasure and then exits. In this game, there are only four states, as illustrated in figure 12.4. Among them, state 0 represents a fire pit, and state 3 represents the exit—both are terminal states. State 1 contains the treasure, symbolized by a diamond. The game starts with the agent positioned in state 2 and has the options of moving left or right as actions. Upon reaching the treasure, the agent receives a reward of +10. However, falling into the fire pit results in a penalty of –10. Successfully exiting grants a reward of +4.

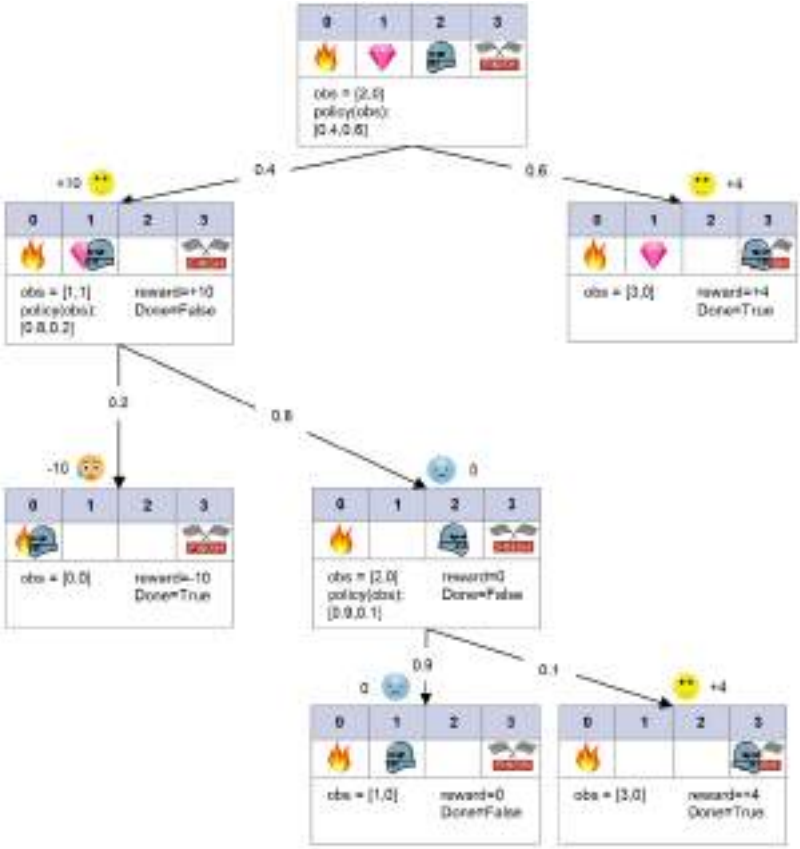


Figure 12.4 Reach the Treasure game

The state *s* provides the complete information about the environment, including the agent's position and the locations of the fire pit, treasure, and exit (state 0 is the fire pit, state 1 is the treasure, state 2 is the current location, and state 3 is the exit). The game's observations are collected and presented as an observation vector, such as obs = [2,0], indicating that the agent is in state 2 and does not perceive the presence of the treasure. This is a partial view of the environment, as the agent does not have access to the complete state information, such as the locations of the fire pit or exit. The policy denotes the probabilities assigned to moving left or right based on the observation. For instance, a policy(obs) of [0.4, 0.6] signifies a 40% chance of moving left and a 60% chance of moving right. In the single trajectory shown in figure 12.4, we can calculate the expected return as follows: expected return (R) = 0.4 * (10) + 0.6 * (4) + 0.4 * 0.2 * (–10) + 0.4 * 0.8 * (0) + 0.4 * 0.8 * 0.9 * (0) + 0.4 * 0.8 * 0.1 * (4) = 5.728.

The goal in RL is to learn an optimal policy that maximizes the expected cumulative discounted reward. Value iteration, policy iteration, and policy gradients are different iterative methods used in reinforcement learning to achieve this goal. The value function in RL defines the expected cumulative reward of the agent starting from a particular state or state-action pair, following a certain policy. There are two types of value functions: the state-value function $V(s)$ and the action-value function $Q(s,a)$. The state-value function $V(s)$ estimates the expected cumulative future rewards an agent may obtain, starting from a particular state s and following a policy π. It quantifies the desirability of a state based on its projected future rewards. The state-value function $V(s)$ is given by the following formula:

$$V^{\pi}(s) = E_{\pi}\left[R_t | s_t = s\right] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \quad \textbf{12.1}$$

where

- $V^{\pi}(s)$ is the expected return when starting in s and following π thereafter.
- $E_{\pi}[\,]$ denotes the expected value, given that the agent follows policy π.
- t is any time step.
- γ is the discount factor, typically a value between 0 and 1, representing the present value of future rewards relative to immediate rewards. The purpose of discounting is to prioritize immediate rewards more heavily, reflecting the preference for rewards received sooner rather than later. A discount factor close to 0 makes the agent *myopic* (i.e., focused on immediate rewards), while a discount factor close to 1 makes the agent more *farsighted* (i.e., considering future rewards).

The action-value function, $Q(s, a)$, estimates the expected cumulative future rewards an agent can achieve by taking a specific action a from a given state s and following a policy π. It quantifies the "goodness" of taking a specific action in a specific state under a given policy. The action-value function $Q(s, a)$ is given by the following formula:

$$\begin{aligned} Q^{\pi}(s, a) &= E_{\pi}\left[R_t | s_t = s, a_t = a\right] \\ &= E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right] \end{aligned} \quad \textbf{12.2}$$

In *policy iteration methods*, we first compute the value $V^{\pi}(s)$ of each state following an initial policy, and we use these value estimates to improve the policy. This means that from an initial policy we repeatedly alternate between policy evaluation and policy improvement steps (until convergence).

Policy gradient methods learn a parameterized policy that is used by the agent to choose actions. The goal is to find the values of the policy parameters that maximize the expected cumulative reward over time.

Policy gradient

Policy gradient is a model-free policy-based method that doesn't require explicit value function representation. The core idea is to favor actions that lead to higher returns while discouraging actions that result in lower rewards. This iterative process refines the policy over time, aiming to find a high-performing policy.

Instead of explicitly estimating the value function, policy gradient methods work by computing an estimator of the policy gradient and plugging it in to a stochastic gradient ascent algorithm. Policy gradient loss L^{PG} is one of the most commonly used gradient estimators:

$$L^{PG}(\theta) = \widehat{E}_t \left[\log \pi_\theta(a_t, s_t) \widehat{A}_t \right] \quad \textbf{12.3}$$

where

- The expectation E_t indicates the empirical average over a finite batch of samples.
- π_θ is a stochastic policy that takes the observed states from the environment as an input and suggests actions to take as an output.
- $\hat{A}_t$ is an estimate of the advantage function at time step t. This estimate basically tries to assess the relative value of the selected action in the current state. The advantage function represents the advantage of taking a particular action in a given state, compared to the expected value. It is calculated as the difference between the expected rewards from executing the suggested action (which often has the highest Q-value) and the estimated value function of the current state:

$$A(s, a) = Q(s, a) - V(s) \quad \textbf{12.4}$$

where

- $Q(s, a)$ is the action-value function (also known as the Q-value), which represents the expected cumulative rewards from taking action a in state s following the policy.
- $V(s)$ is the state-value function, which represents the expected cumulative rewards from state s following the policy.

As you can see in equation 12.4, if the advantage function is positive, indicating that the observed return is higher than the expected value, the gradient will be positive. This positive gradient means that the probabilities of the actions taken in that state will be increased in the future to enhance their likelihood. On the other hand, if the advantage function is negative, the gradient will be negative. This negative gradient implies that the probabilities of the selected actions will be decreased if similar states are encountered in the future.

12.1.3 Model-based vs. model-free RL

Reinforcement learning is categorized into two main types: model-based RL (MBRL) and model-free RL (MFRL). This classification is based on whether the RL agent possesses a model of the environment or not. The term *model* refers to an internal representation of the environment, encompassing its transition dynamics and reward function. Table 12.1 summarizes the differences between these two categories.

Table 12.1 Model-based RL (MBRL) versus model-free RL (MFRL)

Aspects	Model-based RL (MBRL)	Model-free RL (MFRL)
Environment model	Uses a known model or learns a model of the environment (i.e., transition probabilities)	Skips models and directly learns what action to take when (without necessarily finding out the exact model of the action)
Rewards	Typically known or learned	Unknown or partially known. Model-free RL learns directly from the rewards received during interaction with the environment.
Actions	Selected to maximize the expected cumulative reward using the model	Selected to maximize the expected cumulative rewards based on the history of experiences
Policy	Policy learning is accomplished by learning a model of the environment dynamics.	Policy learning is achieved through trial and error, directly optimizing the policy based on observed experiences.
Design and tuning	MBRL can have a higher initial design and tuning effort due to model complexity. However, advancements are simplifying this process.	Requires less initial effort. However, MFRL hyperparameter tuning can also be challenging, especially for complex tasks.
Examples	AlphaZero, world models, and imagination-augmented agents (I2A)	Q-learning, advantage actor-critic (A2C), asynchronous advantage actor-critic (A3C), and proximal policy optimization (PPO)

Based on how RL algorithms learn and update their policies from collected experiences, RL algorithms can also be classified as off-policy and on-policy RL. Off-policy methods learn from experiences generated by a policy different from the one being updated, while on-policy methods learn from experiences generated by the current policy being updated. Both on-policy and off-policy methods are often considered *model-free* because they directly learn policies or value functions from experiences without explicitly constructing a model of the environment's dynamics, distinguishing them from model-based approaches. Table 12.2 summarizes the differences between off-policy and on-policy model-free RL methods.

Table 12.2 Off-policy versus on-policy RL methods

Aspects	Off-policy RL methods	On-policy RL methods
Learning approach	Learn from experiences generated by a different policy than the one being updated	Learn from experiences generated by the current policy being updated
Sample efficiency	Typically more sample-efficient due to reusing past experiences (recorded data)	Typically less sample-efficient, as the batch of experiences is discarded after each policy update (past experiences are not explicitly stored)
Policy evaluation	Can learn the value function and policy separately, enabling different algorithms (e.g., Q-learning, DDPG, TD3)	Policy evaluation and improvement are typically intertwined in on-policy algorithms (e.g., REINFORCE, A2C, PPO).
Pros	More sample-efficient, can learn from diverse experiences, enables reuse of past data, useful if large amounts of prior data are available	Simpler and more straightforward, avoids off-policy correction, can converge to better local optima, suitable for scenarios with limited data or online learning
Cons	Requires careful off-policy correction, less suitable for online learning or tasks with limited data	Less sample-efficient, discards past experiences, limited exploration diversity, may converge to suboptimal policies

The following two subsections provide more details about A2C and PPO as examples of the on-policy methods used in this chapter.

12.1.4 Actor-critic methods

Figure 12.5 shows the advantage actor-critic (A2C) architecture as an example of actor-critic methods. As the name suggests, this architecture consists of two models: the *actor* and the *critic*. The actor is responsible for learning and updating the policy. It takes the current state as input and outputs the probability distribution over the actions that represent the policy. The critic, on the other hand, focuses on evaluating the action suggested by the actor. It takes the state and action as input and estimates the advantage of taking that action in that particular state. The advantage represents how much better (or worse) the action is compared to the average action in that state based on expected future rewards. This feedback from the critic helps the actor learn and update the policy to favor actions with higher advantages.

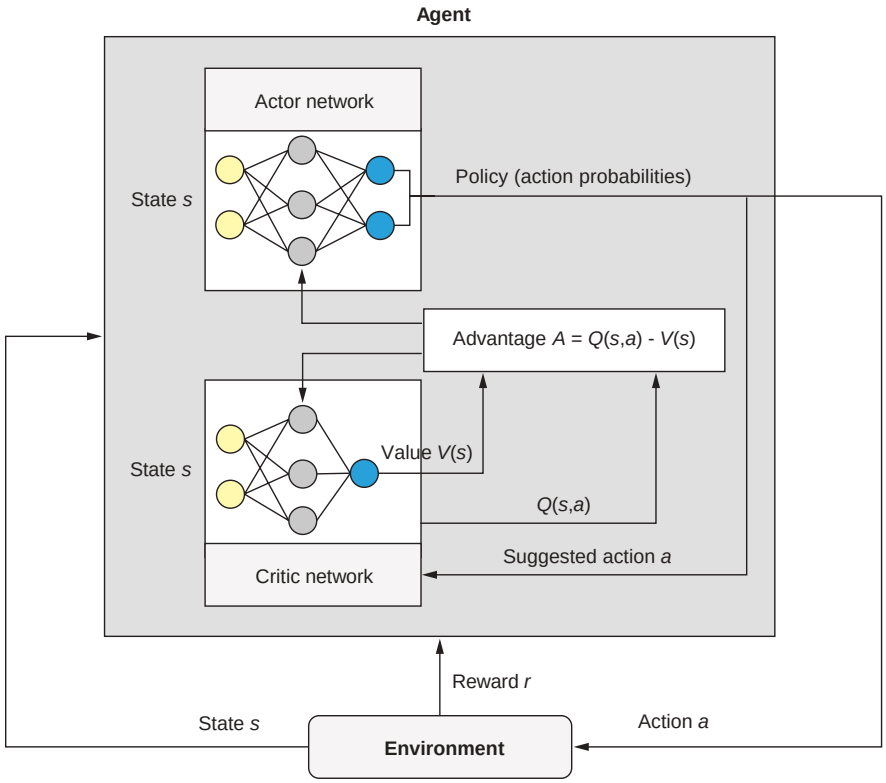


Figure 12.5 The advantage actor-critic (A2C) architecture

A2C is a synchronous, model-free algorithm that aims to learn both the policy (the actor) and the value function (the critic) simultaneously. It learns an optimal policy by iteratively improving the actor and critic networks. By estimating advantages, the algorithm can provide feedback on the quality of the actions taken by the actor. The critic network helps estimate the value function, providing a baseline for the advantages calculation. This combination allows the algorithm to update the policy in a more stable and efficient manner.

12.1.5 Proximal policy optimization

The proximal policy optimization (PPO) algorithm is an on-policy model-free RL designed by OpenAI [2], and it has been successfully used in many applications such as video gaming and robot control. PPO is based on the actor-critic architecture.

In RL, the agent generates its own training data through interactions with the environment. Unlike supervised machine learning, which relies on static datasets, RL's training data is dynamically dependent on the current policy. This dynamic nature leads to constantly changing data distributions, introducing potential instability during training. In the policy gradient method explained previously, if you continuously apply gradient ascent on a single batch of collected experiences, it can lead to updates that push the parameters of the network too far from the range where the data was collected. Consequently, the advantage function, which provides an estimate of the true advantage, becomes inaccurate, and the policy can be severely disrupted. To address this problem, two primary variants of PPO have been proposed: PPO-penalty and PPO-clip.

PPO-PENALTY

In *PPO-penalty*, a constraint is incorporated in the objective function to ensure that the policy update does not deviate too much from the old policy. This idea is the basis of trust region policy optimization (TRPO). By enforcing a trust region constraint, TRPO restricts the policy update to a manageable region and prevents large policy shifts. PPO-penalty is primarily inspired by TRPO and uses the following unconstrained objective function, which can be optimized using stochastic gradient ascent:

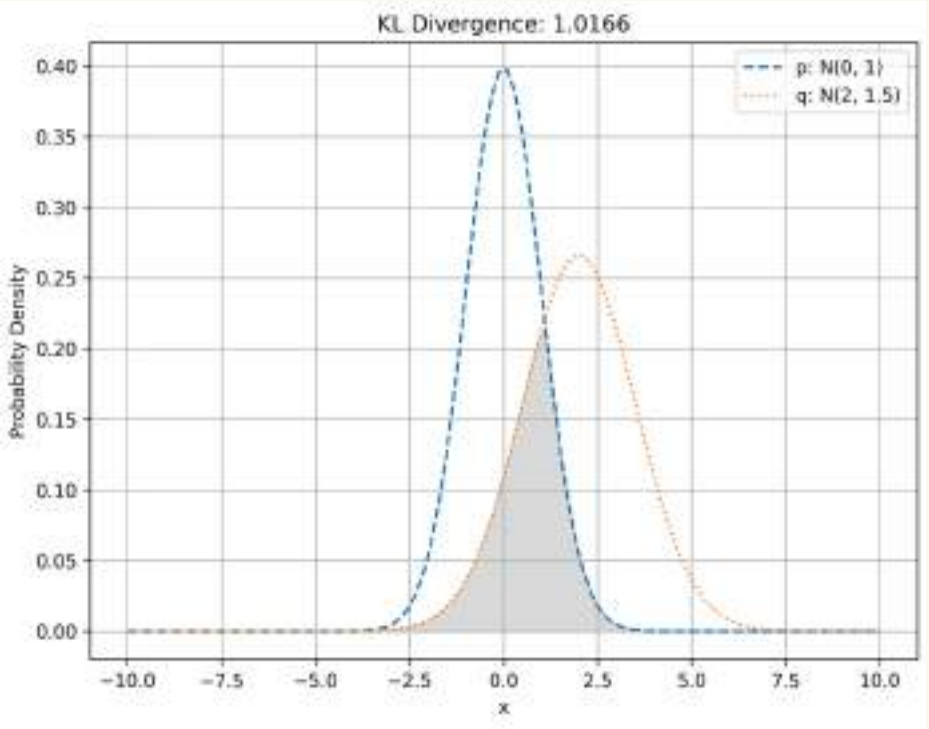
$$\underset{\theta}{\text{maximize}}\ \widehat{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\widehat{A}_t - \beta.KL\left[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)\right]\right] \quad \textbf{12.5}$$

where θ_{old} is the vector of policy parameters before the update, β is a fixed penalty coefficient, and the Kullback–Leibler divergence (KL) represents the divergence between the updated and old policies. This constraint is integrated into the objective function to avoid the risk of moving too far from the old policy.

Kullback–Leibler divergence

Kullback–Leibler (KL) divergence, also known as *relative entropy*, is a metric that quantifies the dissimilarity between two probability distributions. KL divergence between two probability distributions P and Q is defined as $\mathrm{KL}(P \,||\, Q) = \int P(x) \cdot \log(P(x) \,/\, Q(x))\, dx$, where $P(x)$ and $Q(x)$ represent the probability density functions (PDFs) of the two distributions. The integral is taken over the entire support of the random variable x (i.e., the range of values of the random variable where the PDF is nonzero). The following figure shows the KL divergence between two Gaussian distributions with different means and variances.

(continued)



KL divergence between two Gaussian distributions

The KL divergence is equal to zero if, and only if, P and Q are identical distributions.

PPO-CLIP

In *PPO-clip*, a ratio $r(\theta)$ is defined as a probability ratio between the updated policy and the old version of the policy. Given a sequence of sample actions and states, this $r(\theta)$ value will be larger than 1 if the action is more likely now than it was in the old version of the policy, and it will be somewhere between 0 and 1 if it is less likely now than it was before the last gradient step.

The central objective function to be maximized in PPO-clip takes the following form:

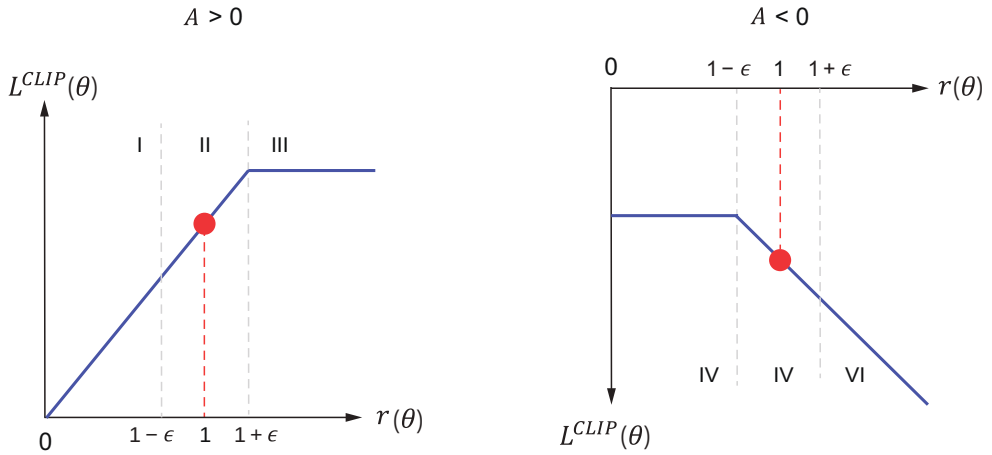
$$L^{CLIP}(\theta) = \widehat{E}_t \left[\min \left(r_t(\theta) \widehat{A}_t, clip\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \widehat{A}_t \right) \right]$$

and

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \qquad \textbf{12.6}$$

where L^{CLIP} is the clipped surrogate objective, which is an *expectation operator* computed over batches of trajectories, and epsilon ϵ is a hyperparameter (e.g., $\epsilon = 0.2$). As you can see in equation 12.6, the expectation operator is taken over the minimum of two terms. The first term represents the default objective used in normal policy gradients. It encourages the policy to favor actions that result in a high positive advantage compared to a baseline. The second term is a clipped or truncated version of the normal policy gradients. It applies a clipping operation to ensure that the update remains within a specified range, specifically between $1 - \epsilon$ and $1 + \epsilon$.

The clipped surrogate objective in PPO has different regions that define how the objective function behaves based on the advantage estimate $\hat{A}_t$ and the ratio of probabilities, as illustrated in figure 12.6.



Region	Advantage function estimate	Selected action	Return value of min	Gradient	Action likelihood?	Clip objective?
I	Positive	Better	$r_t(\theta)\hat{A}_t$	Positive	More likely	No
II	Positive	Better	$r_t(\theta)\hat{A}_t$	Positive	More likely	No
III	Positive	Better	$(1+\epsilon)\hat{A}_t$	Zero	Unchanged	Yes
IV	Negative	Worse	$(1-\epsilon)\hat{A}_t$	Zero	Unchanged	Yes
V	Negative	Worse	$r_t(\theta)\hat{A}_t$	Negative	Less likely	No
VI	Negative	Worse	$r_t(\theta)\hat{A}_t$	Negative	Less likely	No

Figure 12.6 Clipped surrogate objective in PPO

On the left side of figure 12.6, where the advantage function is positive, the objective function represents cases where the selected action has a better-than-expected effect on the outcome. It's important to observe how the objective function flattens out when the ratio (r) becomes too high. This occurs when the action is more likely under the current policy compared to the old policy. The clipping operation limits the update

to a range where the new policy does not deviate significantly from the old policy, preventing excessively large policy updates that may disrupt training stability.

On the right side of figure 12.6, where the advantage function is negative, it represents situations where the action has an estimated negative effect on the outcome. The objective function flattens out when the ratio (r) approaches zero. This corresponds to actions that are much less likely under the current policy compared to the old policy. This flattening effect prevents overdoing updates that could otherwise reduce the probabilities of these actions to zero.

12.1.6 Multi-armed bandit (MAB)

The *multi-armed bandit* (MAB) is a class of reinforcement learning problems with a single state. In MAB, an agent is faced with a set of actions or "arms" to choose from, and each action has an associated reward distribution. The agent's goal is to maximize the total reward accumulated over a series of actions. The agent does not modify its environment through its actions, does not consider state transitions, and always stays in the same single state. It focuses on selecting the most rewarding action at each time step without considering the impact on the environment's state.

Slot machine (one-armed bandit)

A *slot machine* (aka a fruit machine, pokies, or one-armed bandit) is a popular gambling device typically found in casinos. It is a mechanical or electronic gaming machine that features multiple spinning reels with various symbols on them. Players insert coins or tokens into the machine and then pull a lever (hence the term "one-armed bandit") or press a button to initiate the spinning of the reels. The objective of playing a slot machine is to align the symbols on the spinning reels in a winning combination. The term "bandit" originates from the analogy to a slot machine, where the agent pulls an arm (like pulling the lever on a slot machine) to receive varying rewards. This highlights how people perceive slot machines, especially older mechanical ones, as resembling a thief or bandit taking the player's money.

The MAB agent faces the exploration versus exploitation dilemma. To learn the best actions, it must explore various options (exploration). However, it also needs to quickly converge and swiftly focus on the most promising action (exploitation) based on its current beliefs. In supervised and unsupervised machine learning, the primary goal is to fit a prediction model (in supervised learning) or to discover patterns within the given data (in unsupervised learning), without an explicit notion of exploration, as seen in reinforcement learning, where the agent interacts with an environment to learn optimal behavior through trial and error. MAB problems provide valuable insights into learning from limited feedback and balancing exploration against discovering high-reward actions. In MAB, the agent's objective is to exploit the actions that have historically yielded high rewards while exploring to gather information about potentially more rewarding actions.

To understand MAB, imagine you're in a casino, and in front of you is a row of slot machines (one-armed bandits). Each slot machine represents an "arm" of the MAB. Let's assume that you are presented with three slot machines, as shown in figure 12.7. Every time you decide to play a slot machine, your situation (or state) is the same: "Which slot machine should I play next?" The environment doesn't change or provide you with different conditions; you're perpetually making decisions in this singular state. There's no context such as "If the casino is crowded, then play machine A" or "If it's raining outside, then play machine B." It's always just "Which machine should I play?" Your action is choosing a slot machine to play—you insert a coin and pull the lever. You are allowed to pull the levers of these machines for a total of 90 times, and each slot machine has its own payoff distribution, characterized by its mean and standard deviation. However, at the beginning, you are unaware of the specific details of these distributions. After pulling the lever, the machine might give you a payout (a positive reward) or nothing (a reward of zero). Over time, you're trying to discern if one machine gives a higher payout more frequently than the others.

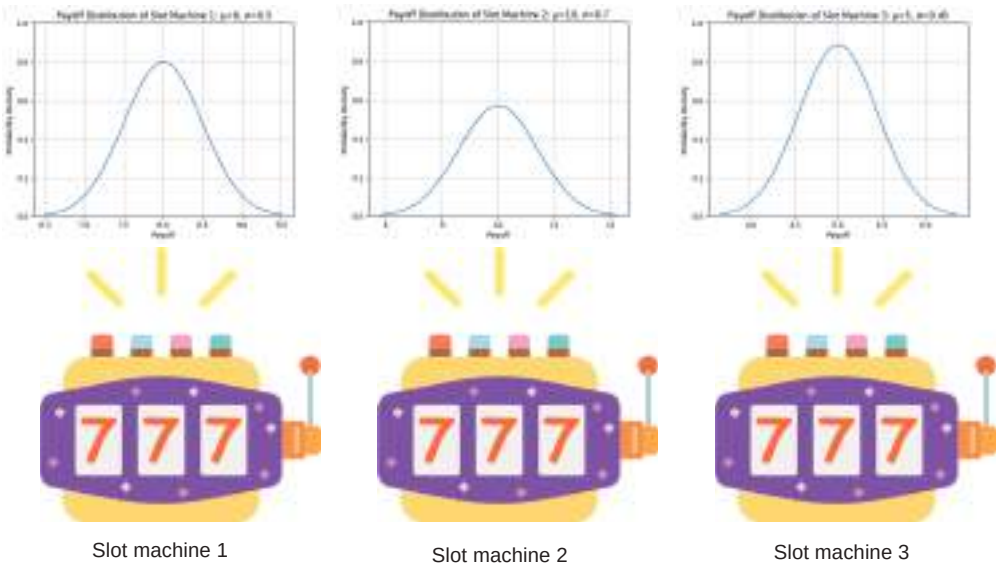


Figure 12.7 Three slot machines as a noncontextual multi-armed bandit (MAB) problem. The payoff distribution's mean and standard deviations of these three machines are (8,0.5), (10, 0.7), and (5, 0.45), respectively.

The objective in this MAB problem is to maximize the total payoff or cumulative reward obtained over the 90 trials by choosing the most rewarding slot machine. In other words, you're trying to learn the payoff distributions of the machines. The term *payoff distribution* refers to the probability distributions of rewards (or payoffs) that each machine (or arm) provides. Since you don't have prior knowledge about these payoff distributions, you need to explore the slot machines by trying different options

to gather information about their performance. The challenge is to strike a balance between exploration and exploitation. Exploration involves trying different machines to learn their payoff distributions, while exploitation involves choosing the machine that is believed to yield the highest rewards based on the available information.

You can apply various bandit algorithms or strategies to determine the best approach for selecting the slot machines and maximizing your cumulative reward over the 90 trials. Examples of these strategies include, but are not limited to, explore-only, exploit-only greedy, epsilon-greedy (ε-greedy), and upper confidence bound (UCB):

- *Explore-only strategy*—In this strategy, the agent randomly selects a slot machine to play at each trial without considering the past results. In MAB, "regret" is a common measure, calculated as the difference between the maximum possible reward and the reward obtained from each selected machine. For example, if we apply the explore-only strategy in the 90 trials, and considering the mean of the payoff distribution of each slot machine, we will get a total average return of $30 \times 8 + 30 \times 10 + 30 \times 5 = 690$. The maximum possible reward can be obtained if you use machine 2 during the 90 trials. The maximum reward in this case will be $90 \times 10 = 900$. This means that regret $\rho = 900 - 690 = 210$.
- *Exploit-only greedy strategy*—In this case, the agent tries each machine once and then selects the slot machine that has the highest estimated mean reward. For example, assume that in the first trial, the agent gets payoffs of 7, 6, and 3 from machines 1, 2, and 3 respectively. The agent will then focus on using machine 1, thinking that it is the most rewarding. This can lead the agent to get stuck due to a lack of exploration.
- *ε-greedy strategy*—Here, the agent tries to strike a balance between exploration and exploitation by randomly selecting a slot machine with a certain probability (epsilon). This is the exploration part, where the agent occasionally tries out all three machines to gather more information about them. With a probability of $1 - \varepsilon$, the agent chooses the machine that has the highest estimated reward based on past experiences. This is the exploitation part, where the agent opts for the action that appears to be the best based on data the agent has gathered so far. For example, for the 90 trials and if $\varepsilon = 10\%$, the agent will randomly select a slot machine about 9 times (10% of 90). The other 81 trials (90% of 90) would see the agent choosing the slot machine that has, based on the trials up to that point, yielded the highest average reward.
- *Upper confidence bound (UCB) strategy*—In this strategy, the agent selects a machine based on a trade-off between its estimated mean reward and the uncertainty or confidence in that estimate. It prioritizes exploring machines with high potential rewards but high uncertainty in their estimates to reduce uncertainty and maximize rewards over time. In UCB, the arm (the action A_t) is chosen at time step t using the following formula:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right] \quad \textbf{12.7}$$

where $Q_t(a)$ is the estimated value of action a at trial t. $Q_t(a)$ = sum of rewards/ $N_t(a)$. $N_t(a)$ is the number of trials where action a has been selected, prior to trial t. The first term on the right side of equation 12.7 represents the exploitation part. If you always pulled the arm with the highest $Q_t(a)$, you would always be exploiting the current knowledge without exploring other arms. The second term is the exploration part. As the number of trials t increases, the exploration term generally increases, but it's reduced by how often action a has already been selected. The multiplier c scales the influence of this exploration term.

Each strategy employs a different approach to balance exploration and exploitation, leading to different levels of regret. This regret level quantifies the cumulative loss an agent incurs due to not always choosing the optimal action. Intuitively, it measures the difference between the reward an agent could have achieved by always pulling the best arm (i.e., by selecting the optimal action) and the reward the agent actually received by following a certain strategy.

Let's look at Python implementations of the four MAB strategies. We'll start by setting the numbers of arms (actions), the payoff distributions of each slot machine, and the number of trials. We'll also define a `sample_payoff` to sample a payoff from a slot machine and calculate the maximum possible reward.

Listing 12.1 MAB strategies

```
import numpy as np

K = 3
payoff_params = [
    {"mean": 8, "std_dev": 0.5},
    {"mean": 10, "std_dev": 0.7},
    {"mean": 5, "std_dev": 0.45}
]
num_trials = 90

def sample_payoff(slot_machine):
    return np.random.normal(payoff_params[slot_machine]["mean"],
➥         payoff_params[slot_machine]["std_dev"])

max_reward = max([payoff_params[i]["mean"] for i in range(K)])
```

Set the number of slot machines (arms).

Specify payoff distribution.

Set the number of trials.

Calculate the maximum possible reward.

Function to sample a payoff from a slot machine

As a continuation of listing 12.1, we'll define a function named `explore_only()` that implements the explore-only strategy. In this function, `total_regret` is initialized, and we iterate over the specified number of trials. A slot machine is randomly selected by generating a random integer between 0 and $K-1$ (inclusive), where K represents the number of slot machines. We then sample the payoff from the selected slot machine

by calling the `sample_payoff()` function, which returns a reward value based on the payoff distribution of the selected machine. The regret is calculated by subtracting the reward obtained from the maximum possible reward (`max_reward`). Here, we consider the maximum value of the mean values as the points of maximum probability in the payoff distributions of the three machines. The average regret is returned as output:

```
def explore_only():
    total_regret = 0
    for _ in range(num_trials):
        selected_machine = np.random.randint(K)
        reward = sample_payoff(selected_machine)
        regret = max_reward - reward
        total_regret += regret
    average_regret = total_regret / num_trials
    return average_regret
```

Randomly select a slot machine.

Sample the payoff from the selected slot machine.

Calculate the regret.

Calculate the total regret.

Calculate the average regret.

The second strategy is defined in the `exploit_only_greedy()` function. This function selects the slot machine with the highest mean payoff by finding the index of the maximum mean payoff from the list of payoff means (`payoff_params[i]["mean"]`) for each machine (`i`). The `np.argmax()` function returns the index of the maximum mean payoff, representing the machine that is believed to provide the highest expected reward:

```
def exploit_only_greedy():
    total_regret = 0
    for _ in range(num_trials):
        selected_machine = np.argmax([payoff_params[i]["mean"] for i in
        range(K)])
        reward = sample_payoff(selected_machine)
        regret = max_reward - reward
        total_regret += regret
    average_regret = total_regret / num_trials
    return average_regret
```

Select the slot machine with the highest mean payoff for exploitation.

Sample the payoff from the selected slot machine.

Calculate the regret.

Calculate the total regret.

Calculate the average regret.

The following `epsilon_greedy(epsilon)` function implements the epsilon-greedy strategy. This function checks if a randomly generated number between 0 and 1 is less than the epsilon value. If it is, the algorithm performs exploration by randomly selecting a slot machine for exploration. If this condition is not satisfied, the algorithm performs exploitation by selecting the slot machine with the highest mean payoff:

```
def epsilon_greedy(epsilon):
    total_regret = 0
    for _ in range(num_trials):
        if np.random.random() < epsilon:
            selected_machine = np.random.randint(K)
        else:
            selected_machine = np.argmax([payoff_params[i]["mean"]
            ➥ for i in range(K)])
        reward = sample_payoff(selected_machine)
        regret = max_reward - reward
        total_regret += regret
    average_regret = total_regret / num_trials
    return average_regret
```

Randomly select a slot machine for exploration.

Select the slot machine with the highest mean payoff for exploitation.

Sample the payoff from the selected slot machine.

The following `ucb(c)` function implements the upper confidence bound (UCB) strategy. This function starts by initializing an array to keep track of the number of plays for each slot machine. The function also initializes an array to accumulate the sum of rewards obtained from each slot machine and initializes a variable to accumulate the total regret. The code includes a loop that plays each slot machine once to gather initial rewards and update the counts and sum of rewards:

```
def ucb(c):
    num_plays = np.zeros(K)
    sum_rewards = np.zeros(K)
    total_regret = 0

    for i in range(K):
        reward = sample_payoff(i)
        num_plays[i] += 1
        sum_rewards[i] += reward

    for t in range(K, num_trials):
        ucb_values = sum_rewards / num_plays + c * np.sqrt(np.log(t) / num_
plays)
        selected_machine = np.argmax(ucb_values)
        reward = sample_payoff(selected_machine)
        num_plays[selected_machine] += 1
        sum_rewards[selected_machine] += reward
        optimal_reward = max_reward
        regret = optimal_reward - reward
        total_regret += regret

    average_regret = total_regret / num_trials
    return average_regret
```

Initialize an array to keep track of the number of plays for each slot machine.

Initialize an array to accumulate the sum of rewards obtained from each slot machine.

Initialize the total regret of each slot machine.

Play each slot machine once to initialize.

Continue playing with the UCB strategy.

Calculate the UCB values.

The following code snippet is used to run the strategies, calculate average regrets, and print the results:

```
avg_regret_explore = explore_only()
avg_regret_exploit = exploit_only_greedy()
avg_regret_epsilon_greedy = epsilon_greedy(0.1)
avg_regret_ucb = ucb(2)
```

Set the epsilon value for the epsilon-greedy strategy, and run the strategy.

Set the value of the exploration parameter c for the UCB strategy, and run the strategy.

```
print(f"Average Regret - Explore only Strategy: {round(avg_regret_
explore,4)}")
print(f"Average Regret - Exploit only Greedy Strategy:
➥    {round(avg_regret_exploit,4)}")
print(f"Average Regret - Epsilon-greedy Strategy:
➥    {round(avg_regret_epsilon_greedy,4)}")
print(f"Average Regret - UCB Strategy: {round(avg_regret_ucb,4)}")
```

Print the results.

Given the random sampling included in the code, your results will vary, but running the code will produce results something like these:

```
Average Regret - Explore only Strategy: 2.246
Average Regret - Exploit only Greedy Strategy: 0.048
Average Regret - Epsilon-greedy Strategy: 0.3466
Average Regret - UCB Strategy: 0.0378
```

MAB algorithms and concepts find applications in various real-world scenarios where decision-making under uncertainty and exploration–exploitation trade-offs are involved. Examples of real-world applications of MABs include, but are not limited to, resource allocation (dynamically allocating resources to different options to maximize performance), online advertising (dynamically allocating ad impressions to different options and learning which ads yield the highest click-through rate, which is the probability that a user clicks on an ad), design of experiments and clinical trials (optimizing the allocation of patients to different treatment options), content recommendation (personalizing content recommendations for users), and website optimization (optimizing different design options).

As you saw in figure 12.1, MABs can be classified into noncontextual and contextual MABs. In contrast to the previously explained noncontextual MABs, a contextual multi-armed bandit (CMAB) uses the contextual information contained in the environment. In CMAB, the learner repeatedly observes a context, selects an action, and receives feedback in the form of a reward or loss specific to the chosen action. CMAB algorithms use supplementary information, known as *side information* or *context*, to make informed decisions in real-world scenarios. For example, in a truck selection problem, the shared context is the type of delivery route (city or interstate). Section 12.5 shows how to use CMAB to solve this problem as an example of combinatorial actions.

Contextual multi-armed bandit applications

Contextual multi-armed bandits (CMABs) have found applications in areas like personalized recommendations and online advertising, where the context can be information about a user. For example, Amazon shows how to develop and deploy a CMAB workflow on SageMaker using a built-in Vowpal Wabbit (VW) container to train and deploy contextual bandit models (http://mng.bz/y8rE). These CMABs can be used to personalize content for a user, such as content layout, ads, search, product recommendations, etc. Moreover, a scalable algorithmic decision-making platform called WayLift was developed based on CMAB using VW for optimizing marketing decisions (http://mng.bz/MZom).

12.2 Optimization with reinforcement learning

RL can be used for combinatorial optimization problems by framing the problem as a Markov decision process (MDP) and applying RL algorithms to find an optimal policy that leads to the best possible solution. In reinforcement learning, an agent learns to make sequential decisions in an environment to maximize a notion of cumulative reward. This process involves finding an optimal policy that maps states to actions in order to maximize the expected long-term reward.

The convergence of reinforcement learning and optimization has recently become an active area of research, drawing significant attention from the academic and industrial communities. Researchers are actively exploring ways to apply the strengths of RL to tackle complex optimization problems efficiently and effectively. For example, the

generic end-to-end pipeline presented in section 11.7 can be used to tackle combinatorial optimization problems such as TSP, the vehicle routing problem (VRP), the satisfiability problem (SAT), maximum cut (MaxCut), maximal independent set (MIS), etc. This pipeline includes training the model with supervised or reinforcement learning. Listing 11.4 showed how to solve TSP with an ML model pretrained using a supervised or reinforcement learning approach, as described by Joshi, Laurent, and Bresson [3].

An end-to-end framework for solving the VRP using reinforcement learning is presented in a paper by Nazari et al. [4]. The capacitated vehicle routing problem (CVRP) was also handled by reinforcement learning in a paper by Delarue, Anderson, and Tjandraatmadja [5]. Another framework called RLOR is described in a paper by Wan, Li, and Wang[6] as a flexible framework of deep reinforcement learning for routing problems such as CVRP and TSP. A distributed model-free RL algorithm called DeepPool is described in a paper by Alabbasi, Ghosh, and Aggarwal [7] as learning optimal dispatch policies for ride-sharing applications by interacting with the environment. DeepFreight is another model-free RL algorithm for the freight delivery problem described in a paper by Jiayu et al. [8]. It decomposes the problem into two closely collaborative components: truck-dispatch and package-matching. The key objectives of the freight delivery system are to maximize the number of served requests within a certain time limit and to minimize the total fuel consumption of the fleet during this process. MOVI is another model-free approach for a large-scale taxi dispatch problem, described by Oda and Joe-Wong [9]. RL is also used to optimize traffic signal control (TSC) as a way to mitigate congestion. In a paper by Ruan et al. [10] (of which I was a co-author), a model of a real-world intersection with real traffic data collected in Hangzhou, China, is simulated with different RL-based traffic signal controllers. We also proposed a multi-agent reinforcement learning model to provide both macroscopic and microscopic control in mixed traffic scenarios [11]. The experimental results show that the proposed approach demonstrates superior performance compared with other baselines in terms of several metrics, such as throughput, average speed, and safety.

A framework for learning optimization algorithms, known as "Learning to Optimize," is described by Li and Malik [12]. The problem was formulated as an RL problem, in which any optimization algorithm can be represented as a policy. Guided policy search is used, and autonomous optimizers are trained for different classes of convex and non-convex objective functions. These autonomous optimizers converge faster or reach better optima than hand-engineered optimizers. This is somewhat similar to the amortized optimization concept described in section 11.6, but using reinforcement learning.

RL-based dispatching is described by Toll et al. for a four-elevator system in a 10-floor building [13]. The elevator dispatching problem is a combinatorial optimization problem that involves efficiently dispatching multiple elevators in a multi-floor building to serve incoming requests from passengers. As explained in section 1.4.2, the number of possible states in this case is 10^4 (elevator positions) $\times$ 2^{40} (elevator buttons) $\times$ 2^{18} (hall call buttons) = 2.88 x 10^{21} different states.

Stable-Baselines3 (SB3) provides reliable implementations of reinforcement learning algorithms in PyTorch for several OpenAI Gym-compatible and custom RL environments. To install Stable Baselines3 with pip, execute `pip install stable-baselines3`. Examples of RL algorithm implementations in SB3 include advantage actor-critic (A2C), soft actor-critic (SAC), deep deterministic policy gradient (DDPG), deep Q network (DQN), hindsight experience replay (HER), twin delayed DDPG (TD3), and proximal policy optimization (PPO). Environments and projects available in SB3 include the following:

- *mobile-env*—A Gymnasium environment for autonomous coordination in wireless mobile networks. It allows the simulation of various scenarios involving moving users in a cellular network with multiple base stations. This environment is used in section 12.4.
- *gym-electric-motor*—An OpenAI Gym environment for the simulation and control of electric drivetrains. This environment is used in exercise 6 for this chapter (see appendix C).
- *highway-env*—An environment for decision-making in autonomous driving in different scenarios, such as highway, merge, roundabout, parking, intersection, and racetrack.
- *Generalized state dependent exploration (gSDE) for deep reinforcement learning in robotics*—An exploration method to train RL agents directly on real robots.
- *RL Reach*—A platform for running reproducible RL experiments for customizable robotic reaching tasks.
- *RL Baselines3 Zoo*—A framework to train, evaluate RL agents, tune hyperparameters, plot results, and record videos.
- *Furuta Pendulum Robot*—A project to build and train a rotary inverted pendulum, also known as a Furuta pendulum.
- *UAV_Navigation_DRL_AirSim*—A platform for training UAV navigation policies in complex unknown environments.
- *tactile-gym*—RL environments focused on using a simulated tactile sensor as the primary source of observations.
- *SUMO-RL*—An interface to instantiate RL environments with Simulation of Urban MObility (SUMO) for traffic signal control.
- *PyBullet Gym*—Environments for single and multi-agent reinforcement learning of quadcopter control.

The following sections provide examples of how you can use RL methods to handle control problems with combinatorial actions.

12.3 Balancing CartPole using A2C and PPO

Let's consider a classic control task where the goal is to balance a pole on top of a cart by moving the cart left or right, as shown in figure 12.8. This task can be considered an optimization problem where the objective is to maximize the cumulative reward by finding an optimal policy that balances the pole on the cart for as long as possible. The agent needs to learn how to make decisions (take actions) that maximize the reward signal. The agent explores different actions in different states and learns which actions lead to higher rewards over time. By iteratively updating its policy based on observed rewards, the agent aims to optimize its decision-making process and find the best actions for each state.

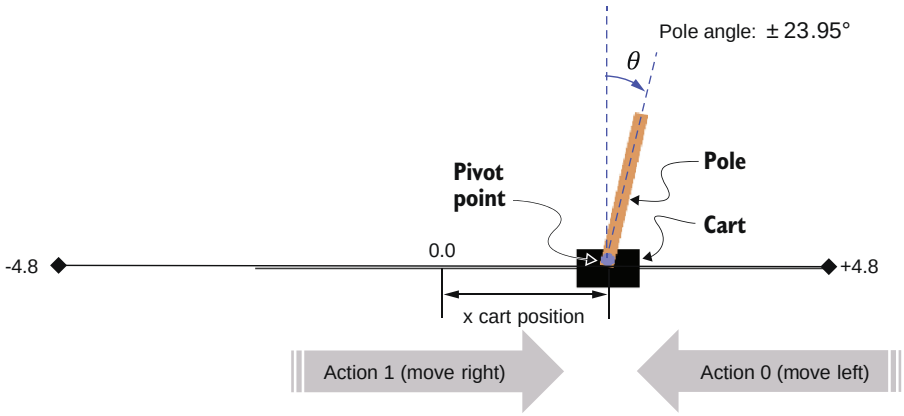


Figure 12.8 CartPole balancing problem

The state of this environment is described by four variables:

- Cart position (continuous)—This represents the position of the cart along the *x*-axis. The value ranges from –4.8 to 4.8.
- Cart velocity (continuous)—This represents the velocity of the cart along the *x*-axis. The value ranges from –inf to inf.
- Pole angle (continuous)—This represents the angle of the pole from the vertical position. The value ranges from –0.418 to 0.418 radians or –23.95° to 23.95° degrees.
- Pole angular velocity (continuous)—This represents the angular velocity of the pole. The value ranges from –inf to inf.

The action space in the CartPole environment is discrete and consists of two possible actions:

- Action 0—Move the cart to the left.
- Action 1—Move the cart to the right.

The agent receives a reward of +1 for every time step when the pole remains upright. The episode ends if one of the following conditions is met:

- The pole angle is more than ±12 degrees from the vertical.
- The cart position is more than ±2.4 units from the center.
- The episode reaches a maximum time step limit (typically 200 steps).

In the CartPole environment, the objective is to balance the pole on the cart for as long as possible, maximizing the cumulative reward. Let's look at the code for learning the optimal policy to balance the CartPole using the advantage actor-critic (A2C) algorithm discussed in section 12.1.4.

As shown in listing 12.2, we start by importing the necessary libraries:

- `gym` is the OpenAI Gym library, used for working with reinforcement learning environments.
- `torch` is the PyTorch library used for building and training neural networks.
- `torch.nn` is a module providing the tools for defining neural networks.
- `torch.nn.functional` contains various activation and loss functions.
- `torch.optim` contains optimization algorithms for training neural networks.
- `tqdm` provides a progress bar for tracking the training progress.
- `seaborn` is used for visualization.

Listing 12.2 Balancing CartPole using the A2C algorithm

```
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from tqdm import tqdm
import pandas as pd
import seaborn as sns
```

Next, we create the actor and critic networks using PyTorch. The `Actor` class is a subclass of `nn.Module` in PyTorch, representing the policy network, and the `__init__` method defines the architecture of the actor network. Three fully connected layers (`fc1`, `fc2`, `fc3`) are used. The `forward` method performs a forward pass through the network, applying activation functions (ReLU) and returning the action probabilities using the softmax function:

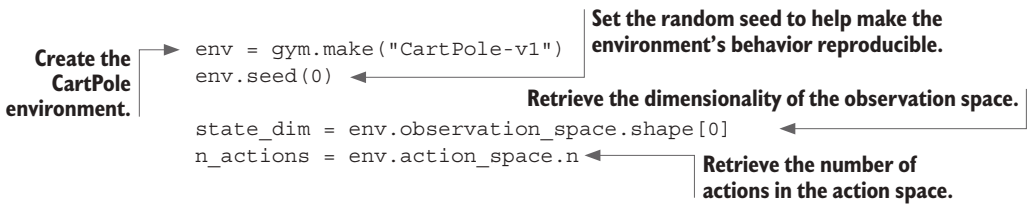
```
class Actor(nn.Module):

    def __init__(self, state_dim,  action_dim):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
```

```
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, action_dim)

    def forward(self, state):
        x1 = F.relu(self.fc1(state))
        x2 = F.relu(self.fc2(x1))
        action_probs = F.softmax(self.fc3(x2), dim=-1)
        return action_probs
```

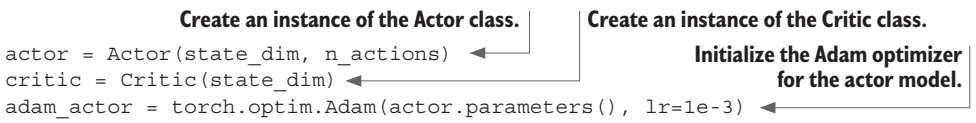
The `Critic` class is also a subclass of `nn.Module`, representing the value network. The `__init__` method defines the architecture of the critic network, similar to the actor network. The forward method performs a forward pass through the network, applying activation functions (ReLU) and returning the predicted value:

```
class Critic(nn.Module):

    def __init__(self, state_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, state):
        x1 = F.relu(self.fc1(state))
        x2 = F.relu(self.fc2(x1))
        value = self.fc3(x2)
        return value
```

As a continuation, the following code snippet is used to create an instance of the CartPole environment using the OpenAI Gym library and to retrieve important information about the environment:



`state_dim` represents the state space or the observation space and in this example has a value of 4 (the four states being cart position, cart velocity, pole angle, and pole angular velocity). `n_actions` represents the dimensionality of the action space or the number of actions, which in this example is 2 (push left and push right). We can now initialize the actor and critic models as well as the Adam optimizer for the actor and critic models using learning rate `lr=1e-3`. The discount factor, gamma, determines the importance of future rewards compared to immediate rewards in reinforcement learning algorithms:



```
adam_critic = torch.optim.Adam(critic.parameters(), lr=1e-3)
gamma = 0.99
```

Set the discount rate.

Initialize the Adam optimizer for the critic model.

After this initialization, we can start the training process, where an agent interacts with the environment for a specified number of episodes. During the training, the agent computes the advantage function, updates the actor and critic models, and keeps track of the training statistics. The following code snippet is used to initialize the training process:

Set the total number of episodes to run.

Create an empty list to store the total rewards obtained in each episode.

```
num_episodes=500
episode_rewards = []
stats={'actor loss':[], 'critic loss':[], 'return':[]}
pbar = tqdm(total=num_episodes, ncols=80, bar_format='{l_bar}{bar}| {n_fmt}/
➥ {total_fmt}')
```

Initialize the tqdm progress bar.

Create a dictionary to store the training statistics, including the actor loss, critic loss, and total return for each episode.

The training loop iterates over the specified number of episodes. In each episode, the environment is reset to its initial state, and the random seeds are initialized to ensure that the sequence of random numbers generated by the environment remains consistent across different runs of the code:

```
for episode in range(num_episodes):
    done = False
    total_reward = 0
    state = env.reset()
    env.seed(0)
```

The agent then interacts with the environment by taking actions based on its policy, accumulating rewards, and updating its parameters to improve its performance in balancing the pole on the cart until the episode is complete. The actor network is used to determine action probabilities given the current state, and a categorical distribution is created using these probabilities. An action is then stochastically sampled from this distribution and executed in the environment. The resulting next state, reward, done flag, and additional information are received from the environment, completing one step of the agent-environment interaction loop:

Obtain action probabilities given the current state.

Create a categorical distribution.

```
while not done:
    probs = actor(torch.from_numpy(state).float())
    dist = torch.distributions.Categorical(probs=probs)
    action = dist.sample()

    next_state, reward, done, info = env.step(action.detach().data.numpy())
```

Sample an action from the categorical distribution.

Pass the action to the environment.

The advantage function is then computed using the rewards, next state value, and current state value.

```
advantage = reward + (1-
➥ done)*gamma*critic(torch.from_numpy(next_state).float()) -
➥     critic(torch.from_numpy(state).float())

total_reward += reward
state = next_state
```

Compute the advantage function.

Update the total reward.

Move to the next state.

This advantage function is used to update the critic model. The critic loss is calculated as the mean squared error of the advantage. The critic parameters are updated using the Adam optimizer:

```
critic_loss = advantage.pow(2).mean()
adam_critic.zero_grad()
critic_loss.backward()
adam_critic.step()
```

The actor loss is calculated as the negative log probability of the chosen action multiplied by the advantage. The actor parameters are updated using the Adam optimizer:

```
actor_loss = -dist.log_prob(action)*advantage.detach()

adam_actor.zero_grad()
actor_loss.backward()
adam_actor.step()
```

The total reward for the episode is then appended to the `episode_rewards` list:

```
episode_rewards.append(total_reward)
```

The actor loss, critic loss, and total reward for the episode are added to the `stats` dictionary. The statistics are printed for each episode:

```
    stats['actor loss'].append(actor_loss)
    stats['critic loss'].append(critic_loss)
    stats['return'].append(total_reward)
    print('Actor loss= ', round(stats['actor loss'][episode].item(), 4), 'Critic
       ➥ loss= ', round(stats['critic loss'][episode].item(), 4), 'Return= ',
       ➥ stats['return'][episode])

    pbar.set_description(f"Episode {episode + 1}")
    pbar.set_postfix({"Reward": episode_rewards})
    pbar.update(1)
pbar.close()
```

Print the tracking statistics.

Update the tqdm progress bar.

Close the tqdm progress bar.

Let's now visualize the learning process by plotting the episode rewards obtained during each training episode using a scatter plot with a trend line:

```
data = pd.DataFrame({"Episode": range(1, num_episodes + 1), "Reward":
➥     episode_rewards})

plt.figure(figsize=(12,6))
sns.set(style="whitegrid")
```

Set the size of the figure to be displayed.

Create a DataFrame for episode rewards.

Set the style of the seaborn plots to have a white grid background.

```
sns.regplot(data=data, x="Episode", y="Reward", scatter_kws={"alpha": 0.5})
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.title("Episode Rewards with Trend Line")
plt.show()
```

Create the scatter plot with a trend line.

When you run this code, you'll get a scatter plot and trend line something like the one in figure 12.9.

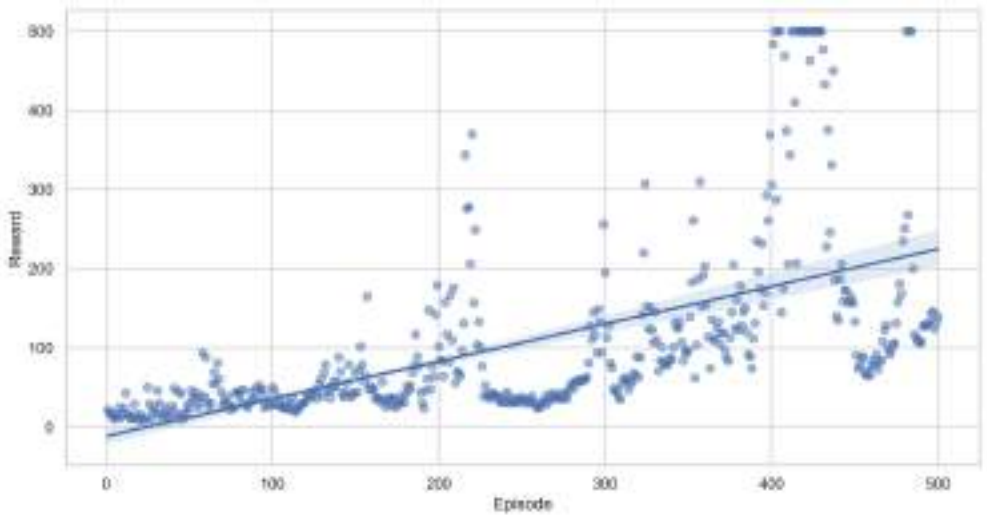
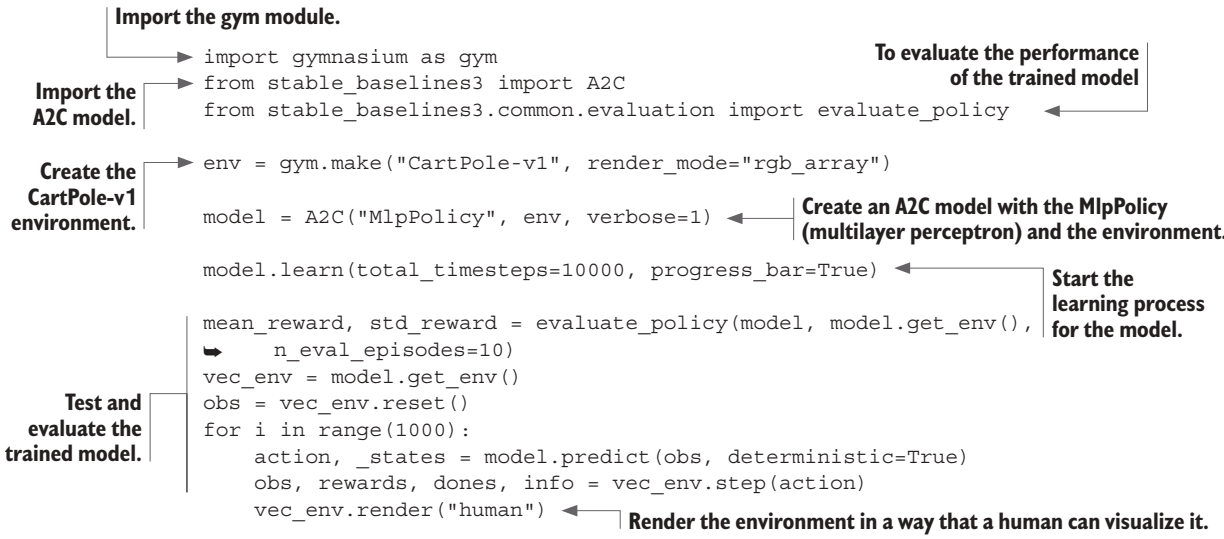


Figure 12.9 Episode rewards with a trend line

As you can see, there are fluctuations in the rewards during the learning process, as depicted by the scatter plot showing increasing and decreasing rewards in different episodes. However, the overall trend or pattern in the rewards over the episodes is improving. Fluctuations in the rewards during the learning process are expected and considered normal behavior in reinforcement learning. Initially, the agent may explore different actions, which can lead to both successful and unsuccessful episodes, resulting in varying rewards. As the training progresses, the agent refines its policy and tends to exploit more promising actions, leading to more consistent rewards.

Stable Baselines3 (SB3) provides more abstract and reliable implementations of different reinforcement learning algorithms based on PyTorch. As a continuation of listing 12.2, the following code snippet shows the steps for handling the CartPole environment using the A2C implementation in SB3. The A2C agent uses `MlpPolicy` as a specific type of policy network, which in turn uses a multilayer perceptron (MLP) architecture. The created agent will interact with the environment for a total of 10,000 timesteps to learn the optimal policy:

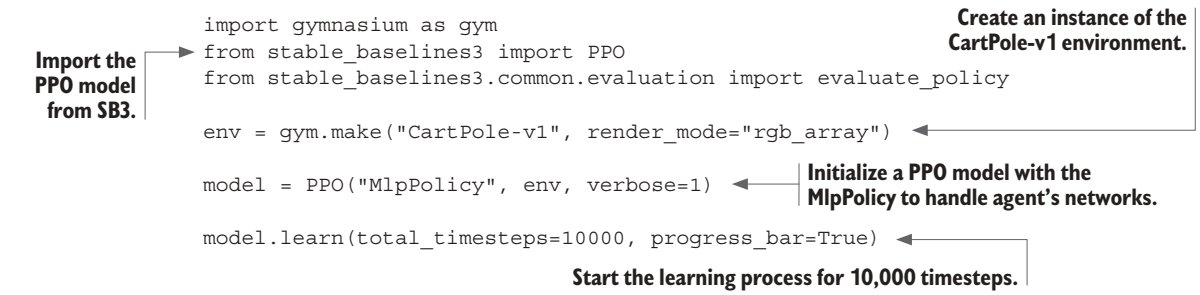



```
import gymnasium as gym
from stable_baselines3 import A2C
from stable_baselines3.common.evaluation import evaluate_policy

env = gym.make("CartPole-v1", render_mode="rgb_array")

model = A2C("MlpPolicy", env, verbose=1)

model.learn(total_timesteps=10000, progress_bar=True)

mean_reward, std_reward = evaluate_policy(model, model.get_env(),
➥     n_eval_episodes=10)
vec_env = model.get_env()
obs = vec_env.reset()
for i in range(1000):
    action, _states = model.predict(obs, deterministic=True)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

This code snippet sets up the environment, trains an A2C model, evaluates its performance for 1,000 steps, and then visualizes the model's actions in the environment.

We can use PPO instead of A2C to balance the CartPole. The next listing is quite similar to the previous one, but it uses PPO instead of A2C.

Listing 12.3 Balancing CartPole using the PPO algorithm




```
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.evaluation import evaluate_policy

env = gym.make("CartPole-v1", render_mode="rgb_array")

model = PPO("MlpPolicy", env, verbose=1)

model.learn(total_timesteps=10000, progress_bar=True)
```

During the training, the code renders logger output in the following format:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 61.8        |
|    ep_rew_mean          | 61.8        |
| time/                   |             |
|    fps                  | 362         |
|    iterations           | 5           |
|    time_elapsed         | 28          |
|    total_timesteps      | 10240       |
| train/                  |             |
|    approx_kl            | 0.0064375857 |
|    clip_fraction        | 0.051       |
|    clip_range           | 0.2         |
```

```
|    entropy_loss          | -0.61        |
|    explained_variance    | 0.245        |
|    learning_rate         | 0.0003       |
|    loss                  | 26.1         |
|    n_updates             | 40           |
|    policy_gradient_loss | -0.0141      |
|    value_loss            | 65.2         |
------------------------------------------
```

The logger output presents the following information:

- `rollout/`
  - `ep_len_mean`—The mean episode length during rollouts
  - `ep_rew_mean`—The mean episodic training reward during rollouts
- `time/`
  - `fps`—The number of frames per seconds achieved during training, indicating the computational efficiency of the algorithm
  - `iterations`—the number of completed training iterations
  - `time_elapsed`—The time elapsed in seconds since the beginning of training
  - `total_timesteps`—The total number of timesteps (steps in the environments) the agent has experienced during training
- `train/`
  - `approx_kl`—The approximate Kullback-Leibler (KL) divergence between the old and new policy distributions, measuring the extent of policy changes during training
  - `clip_fraction`—The mean fraction of surrogate loss that was clipped (above the `clip_range` threshold) for PPO.
  - `clip_range`—The current value of the clipping factor for the surrogate loss of PPO
  - `entropy_loss`—The mean value of the entropy loss (negative of the average policy entropy)
  - `explained_variance`—The fraction of the return variance explained by the value function
  - `learning_rate`—The current learning rate value
  - `loss`—The current total loss value
  - `n_updates`—The number of gradient updates applied so far
  - `policy_gradient_loss`—The current value of the policy gradient loss
  - `value_loss`—The current value for the value function loss for on-policy algorithms

We usually keep an eye on the reward and loss values. As a continuation, the following code snippet shows how to evaluate the policy and render the environment state:

```
mean_reward, std_reward = evaluate_policy(model, model.get_env(),
➥     n_eval_episodes=10)
vec_env = model.get_env()
obs = vec_env.reset()
for i in range(1000):
    action, _states = model.predict(obs, deterministic=True)
    obs, rewards, dones, info = vec_env.step(action)
    vec_env.render("human")
```

Retrieve the vectorized environment associated with the model.

Evaluate the policy of the trained model.

Reset the environment to its initial state and get the initial observation.

Test the trained agent.

As shown in the preceding code, after training the model, we evaluate the policy of the trained model over 10 episodes and return the mean and standard deviation of the rewards. We then allow the agent to interact with the environment for 1,000 steps and render the environment. The output will be an animated version of figure 12.8 showing the behavior of the CartPole learning the balancing policy.

12.4 Autonomous coordination in mobile networks using PPO

Schneider et al. described the mobile-env environment as an open platform for reinforcement learning in wireless mobile networks [14]. This environment enables the representation of users moving within a designated area and potentially connecting to one or multiple base stations. It supports both multi-agent and centralized reinforcement learning policies.

In the mobile-env environment, we have a mobile network formed by a number of base stations or cells (BSs) and user equipment (UE), as illustrated in figure 12.10. Our objective is to decide what connections should be established among the UEs and BSs in order to maximize the quality of experience (QoE) globally. For individual UEs, a higher QoE is achieved by establishing connections with as many BSs as possible, resulting in higher data rates. However, since BSs distribute resources among connected UEs (e.g., scheduling physical resource blocks), UEs end up competing for limited resources, leading to conflicting goals.

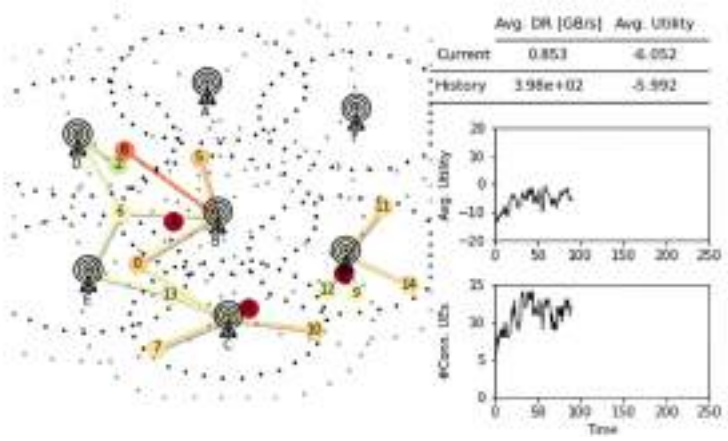


Figure 12.10 The mobile-env environment with a number of base stations and user equipment

To achieve maximum QoE globally, the policy must consider two crucial factors:

- The data rate (DR in GB/s) of each connection is determined by the channel's quality (e.g., the signal-to-noise ratio) between the UE and BS.
- The QoE of individual UEs does not necessarily increase linearly with higher data rates.

Let's create a mobile network environment and train a PPO agent to learn the coordination policy. We'll start in listing 12.4 by importing `gymnasium`, which is a maintained fork of OpenAI's Gym library. `mobile_env` is imported to create an environment related to mobile networks. `IPython.display` enables the use of interactive display features within IPython or Jupyter Notebook environments. We'll create a small instance of `mobile_env` that contains three base stations and five users.

Listing 12.4 Mobile network coordination using PPO

```
import gymnasium
import matplotlib.pyplot as plt
import mobile_env
from IPython import display

from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy

env = gymnasium.make("mobile-small-central-v0", render_mode="rgb_array")
print(f"\nSmall environment with {env.NUM_USERS} users and {env.NUM_STATIONS}
➥     cells.")
```

Print the number of users and the number of base stations.

Create an instance of the environment.

Next, we'll create an instance of the PPO agent using the multilayer perceptron (MLP) policy (`MlpPolicy`) with the mobile environment (`env`) as the environment. The training progress will be logged to the `results_sb` directory for TensorBoard visualization. You can install and set up `tensorboard logdir` as follows:

```
pip install tensorboard
tensorboard --logdir .
```

The agent is trained for a total of 30,000 time steps:

```
model = PPO(MlpPolicy, env, tensorboard_log='results_sb', verbose=1)
model.learn(total_timesteps=30000, progress_bar=True)
```

The following code snippet can be used to render the environment state after training. During the episode, the trained model is used to predict the action to be taken based on the current observation. The action is then executed in the environment, and the environment responds with the next observation (`obs`), the reward received (`reward`), a Boolean flag indicating whether the episode is terminated (`terminated`), a flag indicating whether the episode was terminated due to the episode time limit (`truncated`), and environment information (`info`):

```
obs, info = env.reset()
```

Reset the environment, returning the initial observation and environment information.

```
done = False
while not done:
    action, _ = model.predict(obs)
    obs, reward, terminated, truncated, info = env.step(action)
    done = terminated or truncated

    plt.imshow(env.render())
    display.display(plt.gcf())
    display.clear_output(wait=True)
```

Flag to track if the episode is complete.

Predict the action to be taken based on the current observation.

Execute the action and get the environment response.

Update the flag.

Render the environment state.

The output is an updated version of figure 12.10 showing three stations, five users, and the dynamically changing connections between them. The average data rate and average utility of the established connections are also rendered.

Instead of having a single RL agent centrally control cell selection for all users, an alternative approach is to adopt a multi-agent RL. In this setup, multiple RL agents work in parallel, with each agent being responsible for the cell selection of a specific user. For instance, in the mobile-small-ma-v0 environment, we will use five RL agents, each catering to the cell selection needs of a single user. This approach allows for more distributed and decentralized control, enhancing the scalability and efficiency of the system. We'll use Ray and Ray RLlib in this example. Ray is an open source unified framework for scaling AI and Python applications like machine learning. Ray RLlib is an open source library for RL, offering support for production-level, highly distributed RL workloads while maintaining unified and simple APIs for a large variety of industry applications. To install RLlib with pip, execute `pip install -U "ray[rllib]"`.

As a continuation of listing 12.4, we can import the Ray libraries to learn the optimal coordination policy following a multi-agent approach:

```
import ray
from ray.tune.registry import register_env
import ray.air
from ray.rllib.algorithms.ppo import PPOConfig
from ray.rllib.policy.policy import PolicySpec
from ray.tune.stopper import MaximumIterationStopper
from ray.rllib.algorithms.algorithm import Algorithm
from mobile_env.wrappers.multi_agent import RLlibMAWrapper
```

The following function will create and return a wrapped environment suitable for RLlib's multi-agent setup. Here we create a small instance of `mobile_env`:

```
def register(config):
    env = gymnasium.make("mobile-small-ma-v0")
    return RLlibMAWrapper(env)
```

We'll now initialize Ray using the following function.

```
ray.init(
    num_cpus=2,
    include_dashboard=False,
    ignore_reinit_error=True,
    log_to_driver=False,
)
```

Specify the number of CPUs.

Disable the Ray web-based dashboard.

Ignore the reinitialization error if Ray is already initialized.

Ignore the forwarding logs.

We can now configure an RLlib training setup to use the proximal policy optimization (PPO) algorithm on `mobile-env`'s small scenario in a multi-agent environment:

```
config = (
    PPOConfig()
    .environment(env="mobile-small-ma-v0")
    .multi_agent(
        policies={"shared_policy": PolicySpec()},
        policy_mapping_fn=lambda agent_id, episode, worker, **kwargs:
        ➥ "shared_policy",
    )
    .resources(num_cpus_per_worker=1)
    .rollouts(num_rollout_workers=1)
)
```

Set the environment.

Configure all agents to share the same policy.

Specify that each worker should use one CPU core.

Indicate that there should be one worker dedicated to performing rollouts.

The following code snippet configures and initiates a training session using the RLlib framework. It sets up a tuner (trainer) for the PPO algorithm and executes the training:

```
tuner = ray.tune.Tuner(
    "PPO",
    run_config=ray.air.RunConfig(
        storage_path="./results_rllib",
        stop=MaximumIterationStopper(max_iter=10),
        checkpoint_config=ray.air.CheckpointConfig(checkpoint_at_end=True),
    ),
    param_space=config,
)

result_grid = tuner.fit()
```

Specify PPO.

Specify where the training results and checkpoints (saved model states) will be stored.

Define the stopping condition for the training.

Configure how checkpoints are saved.

Specify the training parameters.

Start the training process.

After training, we can load the best trained agent from the results:

```
best_result = result_grid.get_best_result(metric="episode_reward_mean",
➥ mode="max")
ppo = Algorithm.from_checkpoint(best_result.checkpoint)
```

Extract the best training result from the result_grid based on the metric of average episode reward.

Load the agent from the best checkpoint (model state) obtained in the training.

Lastly, we can evaluate a trained model on a given environment and render the results:

```
env = gymnasium.make("mobile-small-ma-v0", render_mode="rgb_array")
obs, info = env.reset()
done = False

while not done:
    action = {}
    for agent_id, agent_obs in obs.items():
        action[agent_id] = ppo.compute_single_action(agent_obs,
            ➥ policy_id="shared_policy")
```

Reset the environment to its initial state and fetch the initial observation and additional info.

Initialize the environment.

Initiate a loop to run one episode with the trained model.

Initialize an empty dictionary to hold actions for each agent in the multi-agent environment.

Iterate through each agent's observations.

```
obs, reward, terminated, truncated, info = env.step(action)
done = terminated or truncated

plt.imshow(env.render())
display.display(plt.gcf())
display.clear_output(wait=True)
```

Determine if the episode has ended. An episode ends if it is terminated or if truncated is True.

Return the new observations, rewards, termination flags, truncation flags, and additional information.

Visualize the current state of the environment.

Running this code produces an animated rendering of the environment shown in figure 12.11.

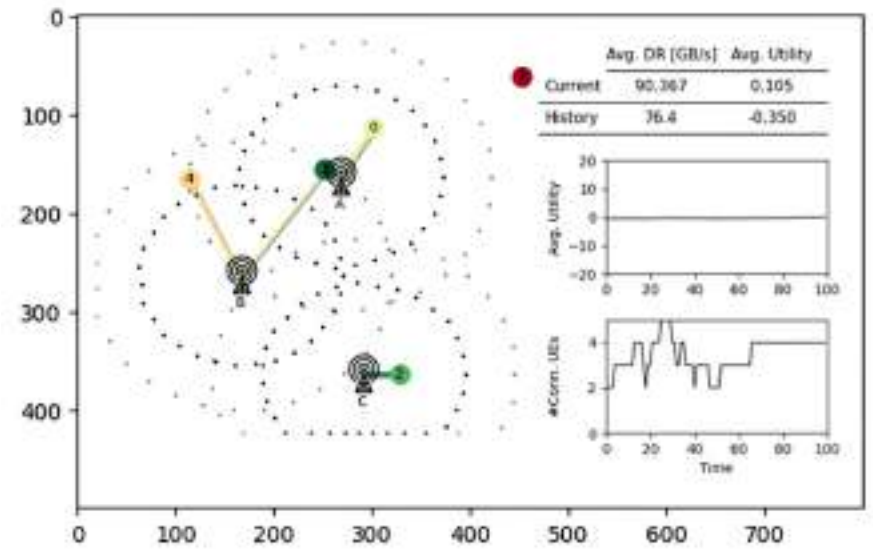


Figure 12.11 Coordination of the mobile environment using a multi-agent PPO

This rendering shows the established connection with the five user units and three base stations and the obtained average data rates and utilities. For more information about the decentralized multi-agent version of the PPO-based coordinator, see Schneider et al.'s article "mobile-env: An open platform for reinforcement learning in wireless mobile networks" and the associated GitHub repo [14].

12.5 Solving the truck selection problem using contextual bandits

Let's consider a scenario where a delivery service provider is planning to assign trucks for different delivery routes, as illustrated in figure 12.12. The goal is to maximize the efficiency of the fleet according to the type of delivery route. The delivery routes are categorized as city deliveries or interstate deliveries, and the company has to select the optimal type of truck according to the following decision variables: size, engine type, and tire type. The available options for each decision variable are as follows:

- Size—Small, medium, or large
- Engine type—Petrol, diesel, or electric
- Tire type—All-season, snow, or off-road

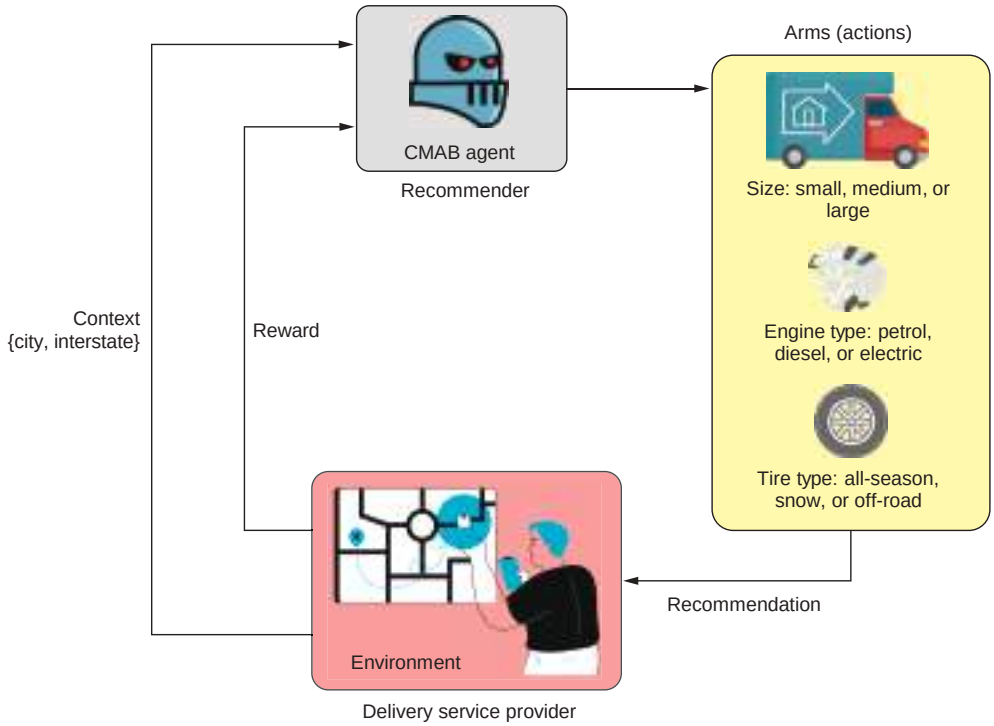


Figure 12.12 A CMAB-based recommender system to select delivery truck size, engine type, and tire type based on a specified context

The reward is based on how suitable the truck selection is for a given delivery route. The reward function receives as arguments the delivery route type and the selected actions for each variable. In order to reflect real-world conditions and add complexity to the problem, we add noise to the reward value, representing uncertainties in weather, road conditions, and so on. The objective is to select the best combination from the available choices in such a way as to maximize the total reward. This means choosing the most suitable truck size, engine type, and tire type for each type of delivery route.

A contextual multi-armed bandit (CMAB) can be used to handle this problem. Vowpal Wabbit (VW), an open source ML library developed originally at Yahoo and currently at Microsoft, supports a wide range of machine-learning algorithms, including CMAB. You can install VW using `pip install vowpalwabbit`.

We'll use CMAB to find the optimal values of the decision variables that maximize the reward based on the given context. The next listing starts by importing the necessary libraries and defining variables for the contextual bandit problem.

Listing 12.5 Contextual bandit for delivery truck selection

```
import vowpalwabbit
import torch
import matplotlib.pyplot as plt
import pandas as pd
import random
import numpy as np
from tqdm import tqdm

shared_contexts = ['city', 'interstate']

size_types = ['small', 'medium', 'large']
engine_types = ['petrol', 'diesel', 'electric']
tire_types = ['all_season', 'snow', 'performance', 'all_terrain']
```

Set the shared context.

Set the action options or the arms.

We then define the following reward function that takes as inputs the shared context and indices representing the chosen size, engine, and tire options. It returns the reward value as an output:

```
def reward_function(shared_context, size_index, engine_index, tire_index):
    size_value = [0.8, 1.0, 0.9]
    engine_value = [0.7, 0.9, 1.0]
    tire_value = [0.9, 0.8, 1.0, 0.95]

    reward = (
        size_value[size_index]
        * engine_value[engine_index]
        * tire_value[tire_index]
    )

    noise_scale = 0.05
    noise_value = np.random.normal(loc=0, scale=noise_scale)
    reward += noise_value

    return reward
```

Higher value indicates better fuel efficiency.

Higher value indicates better performance.

Higher value indicates better comfort.

Initial reward is based on the selected options.

Add noise to the reward representing uncertainties in weather, road conditions, and so on.

Return the reward.

As a continuation, the following `generate_combinations` function generates combinations of actions and examples for the contextual bandit problem. This function takes four inputs: the shared context and three lists representing size, engine, and tire options for the actions. The function returns the list of examples and the list of descriptions once all combinations have been processed. Nested loops are used to iterate over each combination of size, engine, and tire options. The `enumerate` function is used to simultaneously retrieve the index (`i`, `j`, `k`) and the corresponding options (size, engine, tire):

```
def generate_combinations(shared_context, size_types, engine_types, tire_
types):
    examples = [f"shared |User {shared_context}"]
    descriptions = []
    for i, size in enumerate(size_types):
        for j, engine in enumerate(engine_types):
            for k, tire in enumerate(tire_types):
                examples.append(f"|Action truck_size={size} engine={engine}
                    ➥ tire={tire}")
                descriptions.append((i, j, k))
    return examples, descriptions
```

We now need to sample from a probability mass function (PMF) representing truck actions. The `sample_truck_pmf` function samples an index from a given PMF and returns the index along with its probability. The indices are used to retrieve the corresponding size, engine, and tire indices from the indices list:

```
def sample_truck_pmf(pmf):
    pmf_tensor = torch.tensor(pmf)
    index = torch.multinomial(pmf_tensor, 1).item()
    chosen_prob = pmf[index]

    return index, chosen_prob
```

Convert the pmf to a Torch tensor.

Perform a multinomial sampling.

Capture the probability of the selected action.

Return the sampled index and its corresponding probability.

Now we need to create a VW workspace for training the contextual bandit model:

```
cb_vw = vowpalwabbit.Workspace(
    "--cb_explore_adf --epsilon 0.2 --interactions AA AU AAU -l 0.05
--power_t 0",
    quiet=True,
)
```

The workspace is defined using the following parameters:

- `--cb_explore_adf`—This parameter specifies the exploration algorithm for contextual bandit learning using the action-dependent features (ADF). In many real-world applications, there may be features associated with each action (or arm), and these are the *action-dependent features*. This enables the model to explore different actions based on the observed context.
- `--epsilon 0.2`—This parameter sets the exploration rate or epsilon value to 0.2. It determines the probability of the model exploring a random action instead of selecting the action with the highest predicted reward. A higher epsilon value encourages more exploration.
- `--interactions AA AU AAU`—This parameter creates feature interactions between namespaces in VW. These interactions help the model capture more complex relationships between features.
- `-l 0.05`—This parameter sets the learning rate to 0.05. It determines the rate at which the model's internal parameters are updated during the learning process.

A higher learning rate makes the model converge faster, but if you adjust the learning rate too high, you risk over-fitting and end up worse on average.

- `--power_t 0`—This argument sets the power value to 0. It affects the learning rate decay over time. A power value of 0 indicates a constant learning rate.
- `quiet=True`—This argument sets the `quiet` mode to `True`, suppressing the display of unnecessary information or progress updates during the training process. It helps keep the output concise and clean.

The following code snippet is used to train the CMAB model:

```
num_iterations = 2500
cb_rewards = []
with tqdm(total=num_iterations, desc="Training") as pbar:
    for _ in range(num_iterations):
        shared_context = random.choice(shared_contexts)
        examples, indices = generate_combinations(
            shared_context, size_types, engine_types, tire_types
        )
        cb_prediction = cb_vw.predict(examples)
        chosen_index, prob = sample_truck_pmf(cb_prediction)
        size_index, engine_index, tire_index = indices[chosen_index]
        reward = reward_function(shared_context, size_index, engine_index,
            ➥ tire_index)
        cb_rewards.append(reward)
        examples[chosen_index + 1] = f"0:{-1*reward}:{prob} {examples[chosen_
            ➥ index + 1]}"
        cb_vw.learn(examples)
        pbar.set_postfix({'Reward': reward})
        pbar.update(1)
cb_vw.finish()
```

Obtain the chosen index and its corresponding probability.

Obtain the model's predictions for the chosen actions.

Select a random shared context.

Generate examples and indices.

Obtain and append the reward.

Retrieve the corresponding size, engine, and tire indices.

Update the examples corresponding to the chosen index.

Learn and update the model's internal parameters based on the observed rewards.

Finalize the VW workspace.

After training, we can use the following function to test the trained CMAB model. This code evaluates the trained model by generating examples, making predictions, sampling actions, and calculating the expected reward based on a given shared context:

```
def test_model(shared_context, size_types, engine_types, tire_types):
    examples, indices = generate_combinations(shared_context, size_types,
        ➥ engine_types, tire_types)
    cb_prediction = cb_vw.predict(examples)
    chosen_index, prob = sample_truck_pmf(cb_prediction)
    chosen_action = examples[chosen_index]
    size_index, engine_index, tire_index = indices[chosen_index]
    expected_reward = reward_function(shared_context, size_index,
        ➥ engine_index, tire_index)
    print("Chosen Action:", chosen_action)
    print("Expected Reward:", expected_reward)

test_shared_context = 'city'
test_model(test_shared_context, size_types, engine_types, tire_types)
```

The code will produce output something like the following:

```
Chosen Action: Action truck_size=medium engine=electric tire=snow
Expected Reward: 1.012
```

This output represents the chosen action based on the given shared context during testing. In this case, the chosen action specifies a truck with a medium size, an electric engine, and a snow tire. The obtained reward is 1.012. Note that the maximum reward with noise ≈ 1.0 + 0.15 = 1.15. Given that the maximum values from `size_value`, `engine_value`, and `tire_value` are 1, and considering that the noise is a random value with a standard deviation of 0.05, a value of +3 standard deviations (3 × 0.05 = 0.15) would cover about 99.7% of cases in a normal distribution.

12.6 Journey's end: A final reflection

In this book, we have embarked on a comprehensive journey through a diverse landscape of search and optimization algorithms. We first explored deterministic search algorithms that tirelessly traverse problem spaces, seeking optimal solutions through both blind and informed methods. Then we climbed the peaks and valleys of trajectory-based algorithms, witnessing the power of simulated annealing and the ingenious designs of tabu search for escaping local optima. Continuing on our path, we ventured into the realm of evolutionary computing algorithms, witnessing the power of genetic algorithms and their variants in solving complex continuous and discrete optimization problems. Along the way, we embarked on a fascinating journey with swarm intelligence algorithms, starting with particle swarm optimization and offering a glimpse into other algorithms such as the ant colony optimization and artificial bee colony algorithms. Finally, we embraced the realm of machine learning–based methods, where supervised, unsupervised, and reinforcement learning algorithms are used to handle combinatorial optimization problems. Each algorithm covered in this book carries its own set of strengths and weaknesses. Remember, the choice of technique is determined by the task at hand, the characteristics of the problem, and the available resources.

I hope that the knowledge you've gained from this book empowers you to solve real-world problems and embrace the boundless potential of search and optimization in different domains. The fascinating world of search and optimization algorithms continues to expand and evolve. It is up to us to harness this knowledge, to further our capabilities, to solve the problems of today, and to shape the future.

Summary

- Reinforcement learning (RL) can be formulated as an optimization problem wherein the agent aims to learn and/or refine its policy to maximize the expected cumulative reward within a specific environment.
- Reinforcement learning problems can be classified into two main categories: Markov decision processes (MDPs) and multi-armed bandit (MAB) problems. MDP problems involve environments where the agent's actions impact the

environment and its future states. MAB problems focus on maximizing cumulative rewards from a set of independent choices (often referred to as "arms") that can be made repeatedly over time. MABs don't consider the impact of choices on future options, unlike MDPs.

- In MDP-based problems, reinforcement learning uses MDP as a foundational mathematical framework to model decision-making problems under uncertainty. MDP is used to describe an environment for RL where an agent learns to make decisions by performing actions in an environment to achieve a goal.
- RL is classified into model-based and model-free RL, based on the presence or absence of a model of the environment. The model refers to an internal representation or understanding of how the environment behaves—specifically, the transition dynamics and reward function.
- Based on how RL algorithms learn and update their policy from collected experiences, RL algorithms can be classified into off-policy and on-policy RL.
- Advantage actor-critic (A2C) and proximal policy optimization (PPO) are model-free on-policy RL methods.
- By using a clipped objective function, PPO strikes a balance between encouraging exploration and maintaining stability during policy updates. The clipping operation restricts the update to a bounded range, preventing large policy changes that could be detrimental to performance. This mechanism ensures that the policy update remains within a reasonable and controlled distance from the previous policy, promoting smoother and more stable learning.
- Unlike MDPs, Multi-armed bandits (MABs) don't consider the impact of choices on future states, and the agent does not need to worry about transitioning between states because there is only one state. Explore-only, exploit-only greedy, ε-greedy, and upper confidence bound (UCB) are examples of MAB strategies for determining the best approach for selecting the actions to maximize the cumulative reward over the time.
- Contextual multi-armed bandits (CMABs) are an extension of MAB where the decision-making is influenced by additional contextual information about each choice or environment.
- Reinforcement learning can be applied to solve various combinatorial optimization problems, including the traveling salesman problem, traffic signal control, elevator dispatching, optimal dispatch policies for ride-sharing, the freight delivery problem, personalized recommendations, CartPole balancing, coordinating autonomous vehicles in mobile networks, and truck selection.