Chapter 19. Reinforcement Learning

Reinforcement learning (RL) is one of the most exciting fields of machine learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,[1] particularly in games (e.g., *TD-Gammon*, a Backgammon-playing program) and in machine control, but seldom making the headline news. However, a revolution took place in 2013, when researchers from a British startup called DeepMind[2] demonstrated a system that could learn to play just about any Atari game from scratch,[3] eventually outperforming humans[4] in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.[5] This was the first of a series of amazing feats:

- In 2016, DeepMind's AlphaGo beat Lee Sedol, a legendary professional player of the game of Go, and in 2017, it beat Ke Jie, the world champion: no program had ever come close to beating a master of this game, let alone the very best.
- In 2020, DeepMind released AlphaFold, which can predict the 3D shape of proteins with unprecedented accuracy: this is a game changer in biology, chemistry, and medicine. In fact, Demis Hassabis (founder and CEO) and John Jumper (director) were awarded the Nobel Prize in Chemistry for AlphaFold.
- In 2022, DeepMind released AlphaCode, which can generate code at a competitive programming level.
- In 2023, they released GNoME which can predict new crystal structures, including hundreds of thousands of predicted stable materials.

So how did DeepMind achieve all of this? Well, they applied the power of deep learning to the field of reinforcement learning, and it worked beyond their wildest dreams: *deep reinforcement learning* was born. Today, although DeepMind continues to lead the way, many other organizations have joined in and the whole field is boiling with new ideas, with a wide range of applications.

In this chapter I will first explain what reinforcement learning is and what it's good at, then present three of the most important families of techniques in deep reinforcement learning: policy gradients, deep Q-networks (including a discussion of Markov decision processes), and lastly actor-critic methods, including the popular PPO, which we will use to beat an Atari game. So let's get started!

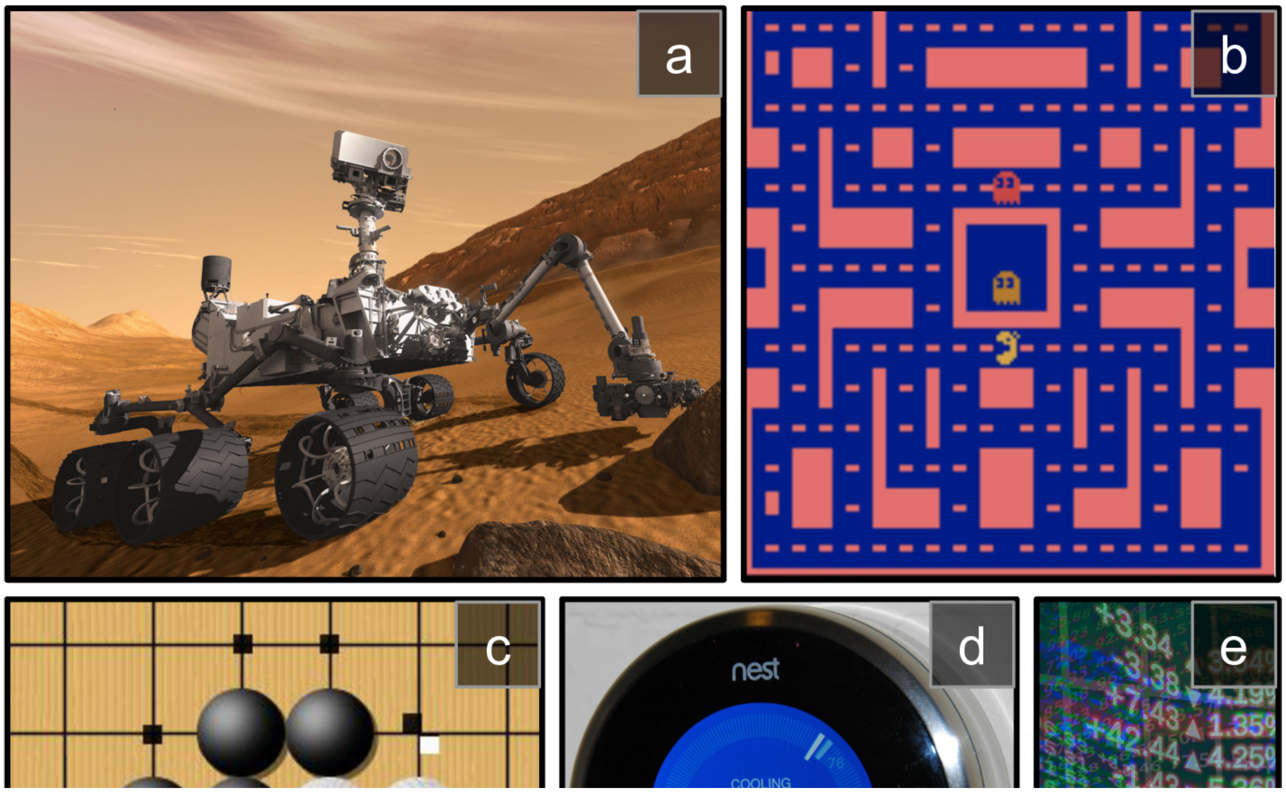
What is Reinforcement Learning?

In reinforcement learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 19-1):

- The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
- The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
- Similarly, the agent can be the program playing a board game such as Go. It only gets a reward if it wins.
- The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to which reinforcement learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.



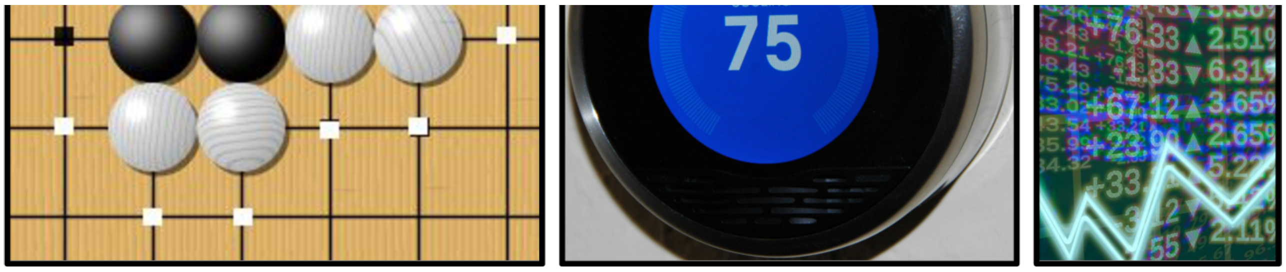


Figure 19-1. Reinforcement learning examples: (a) robotics, (b) *Ms. Pac-Man*, (c) Go player, (d) thermostat, (e) automatic trader[6]

Let's now turn to one large family of RL algorithms: *policy gradients*.

Policy Gradients

The algorithm a software agent uses to determine its actions is called its *policy*. The policy can be any algorithm you can think of, such as a neural network taking observations as inputs and outputting the action to take (see Figure 19-2).

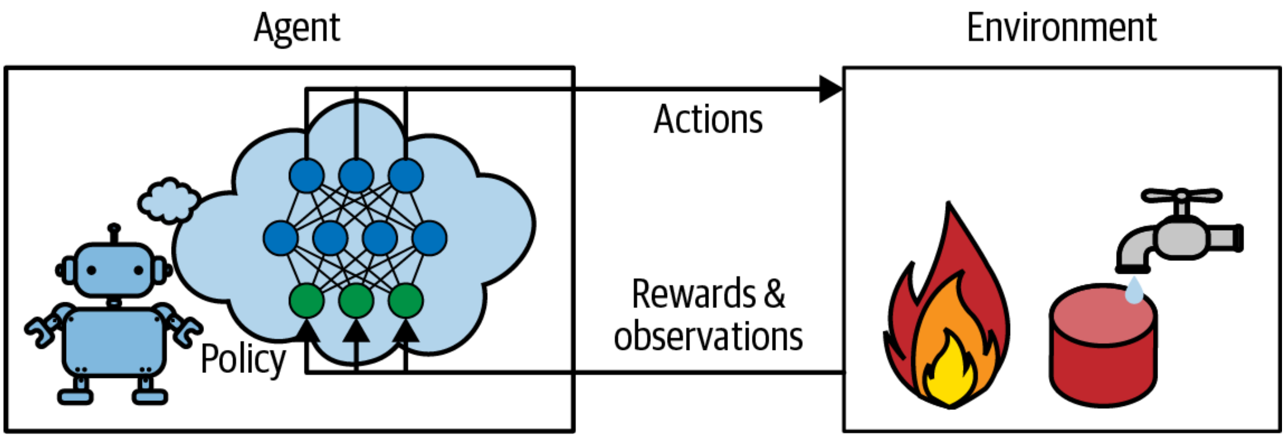


Figure 19-2. Reinforcement learning using a neural network policy

The policy does not even have to be deterministic. In fact, in some cases it does not even have to observe the environment, as long as it can get rewards! For example, consider a blind robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 – p$. The rotation angle would be a random angle between $–r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 19-3). This is an example of *policy search*, in this case using a brute-force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies[7] and make the 20 survivors produce 4 offspring each. An offspring is a copy of its parent[8] plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.[9]

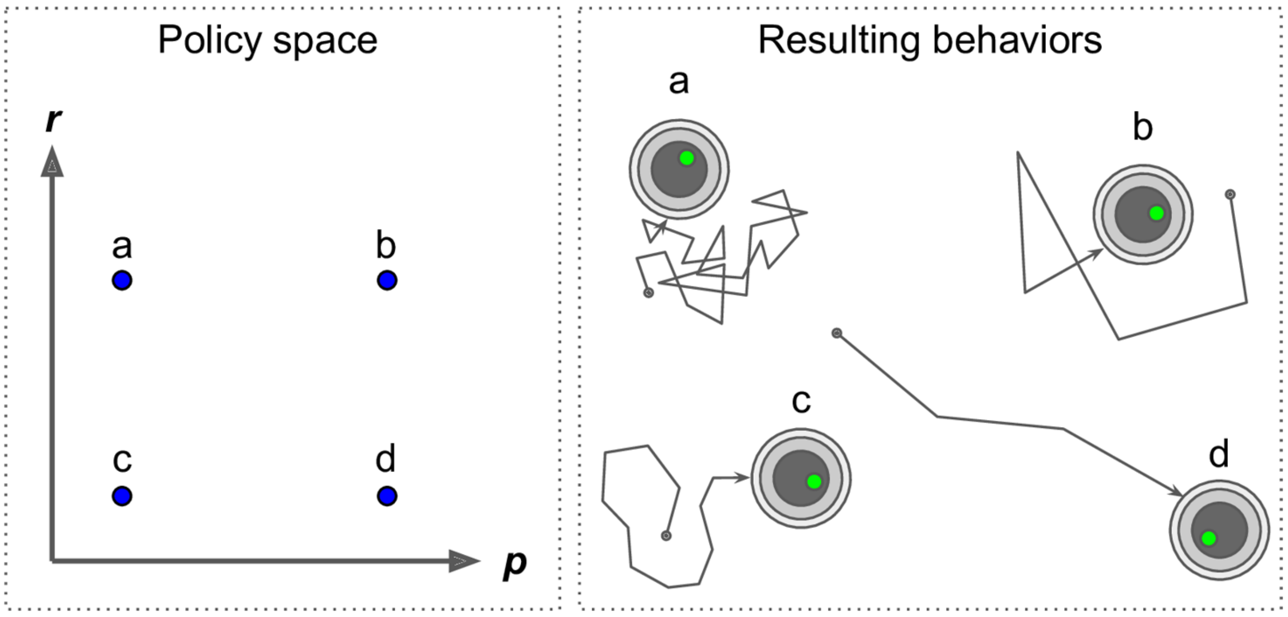


Figure 19-3. Four points in the policy space (left) and the agent's corresponding behavior (right)

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards.[10] Algorithms that follow this strategy are known as *policy gradients* (PG) algorithms. But before we can implement them, we first need to create an environment for the agent to live in—so it's time to introduce the Gymnasium library.

Introduction to the Gymnasium Library

One of the challenges of reinforcement learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment. However, this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either—adding more computing power won't make the robot move any faster—and it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you might use a library like PyBullet or MuJoCo for 3D physics simulation.

The Gymnasium library is an open source toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physics simulations, and so on), that you can use to train agents, compare them, or develop new RL algorithms. It's the successor of *OpenAI Gym*, and is now maintained by a community of researchers and developers.

Gymnasium is preinstalled on Colab, along with the Arcade Learning Environment (ALE) library `ale_py`, which is an emulator for Atari 2600 games and is required for all the Atari environments, as well as the Box2D library, required for several environments with 2D physics. If you are coding on your own machine instead of Colab, and you followed the installation instructions at *https://homl.info/installp*, then you should be good to go.

Let's start by importing Gymnasium and making an environment:

```python
import gymnasium as gym

env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=1000)
```

Here, we've created a CartPole environment (version 1). This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 19-4)—a classic control task. I'll explain `render_mode` and `max_episode_steps` shortly.

TIP

The `gym.envs.registry` dictionary contains the names and specifications of all the available environments. You can print a nice list with `gym.pprint_registry()`. The Atari environments will only be available once we start the ALE emulator.

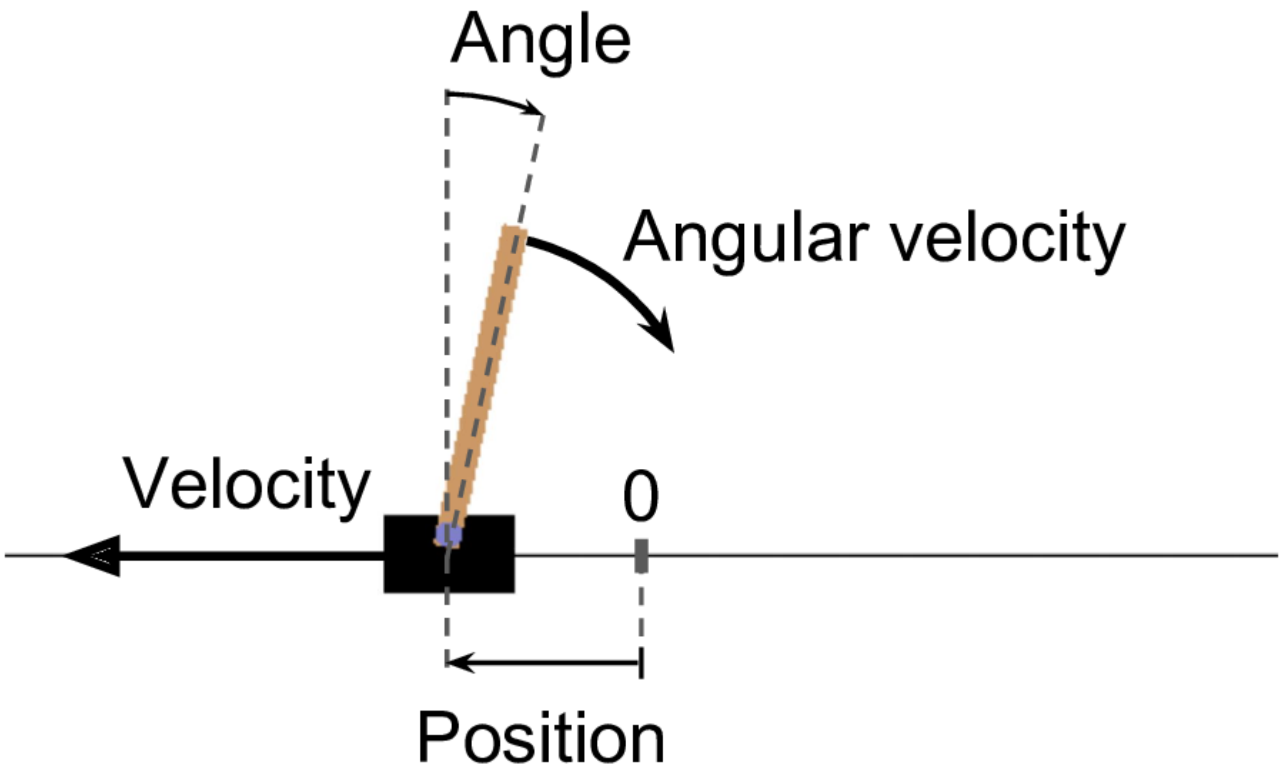


Figure 19-4. The CartPole environment

After the environment is created, you must initialize it using the `reset()` method, optionally specifying a random seed. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a NumPy array containing four floats representing the cart's horizontal position (`0.0` = center), its velocity (positive means right), the angle of the pole (`0.0` = vertical), and its angular velocity (positive means clockwise). The `reset()` method also returns a dictionary that may contain extra environment-specific information. This can be useful for debugging, and sometimes for training. For example, in many Atari environments, it contains the number of lives left. However, in the CartPole environment, this dictionary is empty.

```
>>> obs, info = env.reset(seed=42)
>>> obs
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
>>> info
{}
```

Let's call the `render()` method to render this environment as an image. Since we set `render_mode="rgb_array"` when creating the environment, the image will be returned as a NumPy array (you can then use Matplotlib's `imshow()` function to display this image):

```
>>> img = env.render()
>>> img.shape  # height, width, channels (3 = Red, Green, Blue)
(400, 600, 3)
```

Now let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left or right. Other environments may have additional discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, truncated, info = env.step(action)
>>> obs
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
>>> reward, done, truncated, info
(1.0, False, False, {})
```

The `step()` method executes the desired action and returns five values:

`obs`

This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

`reward`

In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running for as long as possible. An *episode* is one run of the environment until the game is over or interrupted.

`done`

This value will be `True` when the episode is over. This will happen when the pole tilts too much, or goes off the screen. After that, the environment must be reset before it can be used again.

`truncated`

This value will be `True` when an episode is interrupted early, typically by an environment wrapper that imposes a maximum number of steps per episode (see Gymnasium's documentation for more details on environment wrappers). By default, the environment specification for CartPole sets the maximum number of steps to 500, but we changed this to 1,000 when we created the environment. Some RL algorithms treat truncated episodes differently from episodes finished normally (i.e., when `done` is `True`), but in this chapter we will treat them identically.

`info`

This environment-specific dictionary may provide extra information, just like the one returned by the `reset()` method.

TIP

Once you have finished using an environment—possibly after many episodes—you should call its `close()` method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1  # go left if leaning left, otherwise go right

totals = []
for episode in range(500):
    total_rewards = 0
    obs, info = env.reset(seed=episode)
    while True:  # no risk of infinite loop: will be truncated after 1000 steps
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        total_rewards += reward
        if done or truncated:
            break

    totals.append(total_rewards)
    totals.append(total_rewards)
```

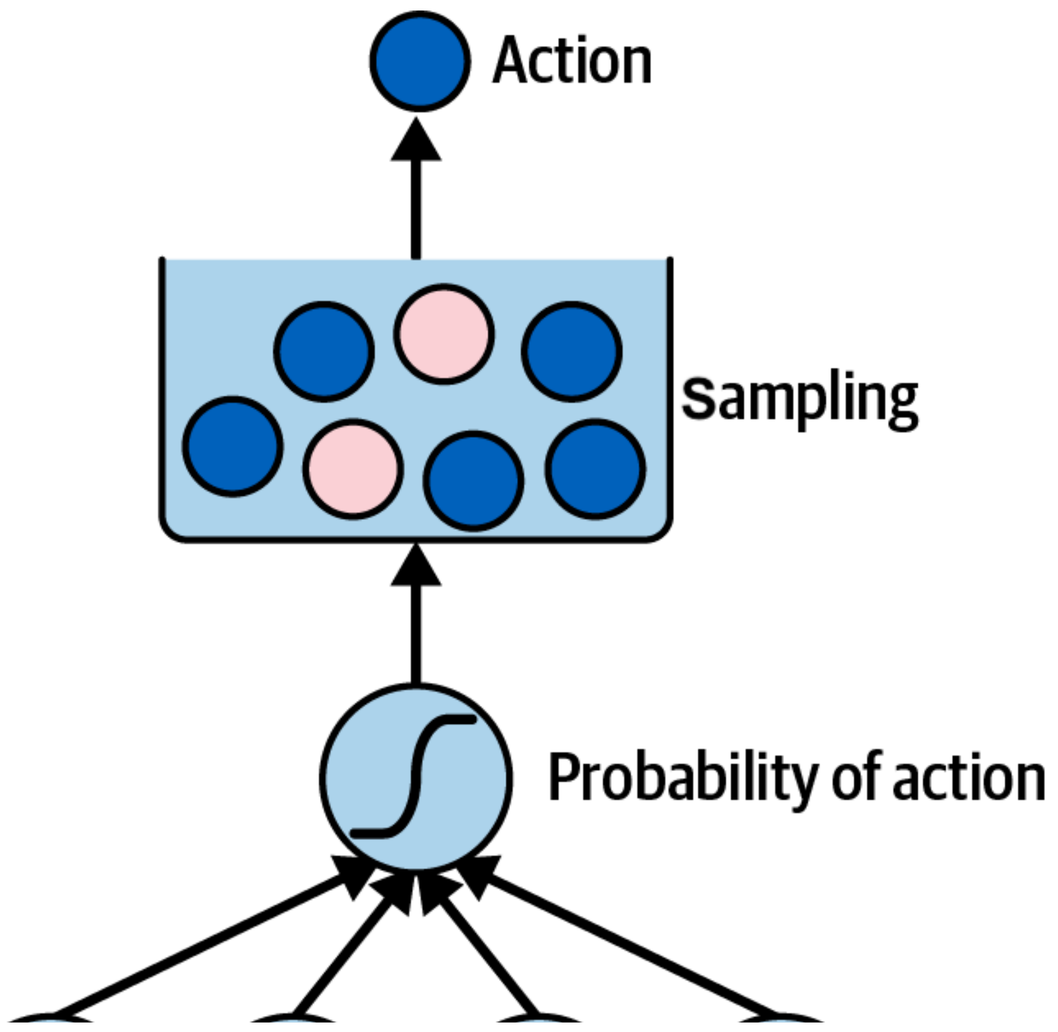
This code is self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), min(totals), max(totals)
(41.698, 8.389445512070509, 24.0, 63.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 63 consecutive steps. Not great. If you look at the simulation in this chapter's notebook, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. A neural network can do better!

Neural Network Policies

Let's create a neural network policy. This neural network will take an observation as input, and it will output the action to be executed, just like the policy we hardcoded earlier. More precisely, it will estimate a probability for each action, then it will select an action randomly, according to the estimated probabilities (see Figure 19-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability *p* of action 1 (right), and of course the probability of action 0 (left) will be 1 – *p*. For example, if it outputs 0.7, then we will pick action 1 with 70% probability, or action 0 with 30% probability (this is a *Bernoulli distribution* with *p* = 0.7).



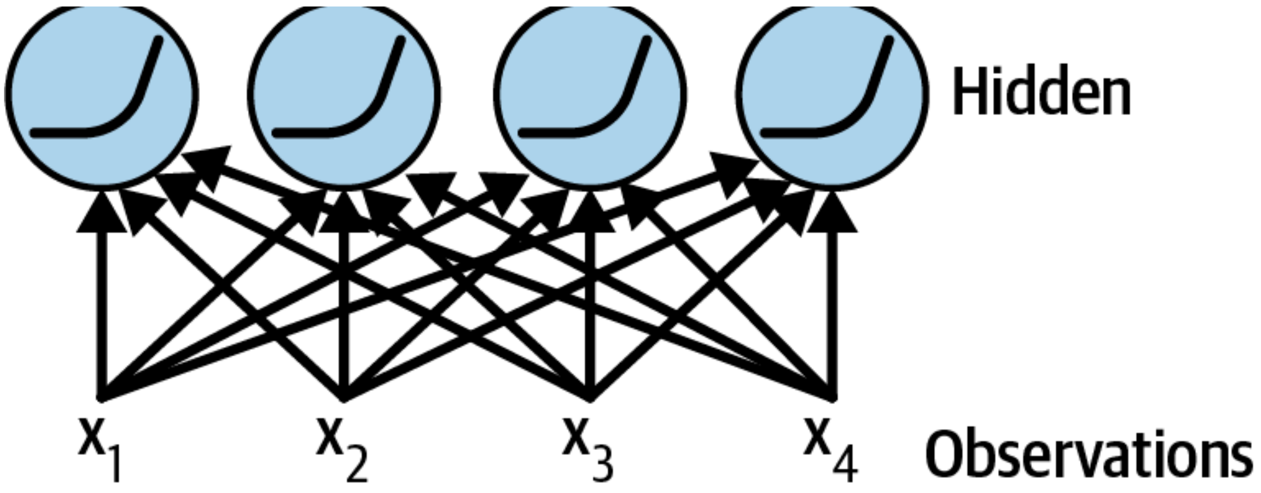


Figure 19-5. Neural network policy

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried. This *exploration/exploitation dilemma* is central in reinforcement learning.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Let's use PyTorch to implement a basic neural network policy for CartPole:

```python
import torch
import torch.nn as nn

class PolicyNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(4, 5), nn.ReLU(), nn.Linear(5, 1))

    def forward(self, state):
        return self.net(state)
```

Our policy network is a tiny MLP, since it's a fairly simple task. The number of inputs is the size of the environment's state: in the case of CartPole, it is just the size of a single observation, which is 4. We have just one hidden layer with 5 units (no need for more in this case). Finally, we want to output a single probability, so we have a single output neuron. If there were more than two possible actions, there would be one output neuron per action instead. For performance and numerical stability, we don't add a sigmoid function at the end, so the network will actually output logits rather than probabilities.

Next let's define a function that will use this policy network to choose an action:

```
def choose_action(model, obs):
    state = torch.as_tensor(obs)
    logit = model(state)
    dist = torch.distributions.Bernoulli(logits=logit)
    action = dist.sample()
    log_prob = dist.log_prob(action)
    return int(action.item()), log_prob
```

The function takes a single observation, converts it to a tensor, and passes it to the policy network to get the logit for action 1 (right). It then creates a `Bernoulli` probability distribution with this logit, and it samples an action from it: this distribution will output 1 (right) with probability *p* = exp(logit) / (1 + exp(logit)), and 0 (left) with probability 1 - *p*. If there were more than two possible actions, you would use a `Categorical` distribution instead. Lastly, we compute the log probability of the sampled action (i.e., either log(*p*) or log(1 - *p*)): this log probability will be needed later for training.

TIP

If the action space is continuous, you can use a Gaussian distribution instead of a Bernoulli or categorical distribution: instead of predicting logits, the policy network must predict the mean and standard deviation (or the log of the standard deviation) of the distribution. The log of the standard deviation is often clipped to ensure the distribution is neither too wide nor too narrow.

OK, we now have a neural network policy that can take an environment state (in this case a single observation) and choose an action. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for a total of 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward tirely responsible. This is called the *credit assignment problem*. When the agent gets a reward (or a penalty), it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount factor*, γ (gamma), at each step. This sum of discounted rewards is called the action's *return*. Consider the example in Figure 19-6. If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$.

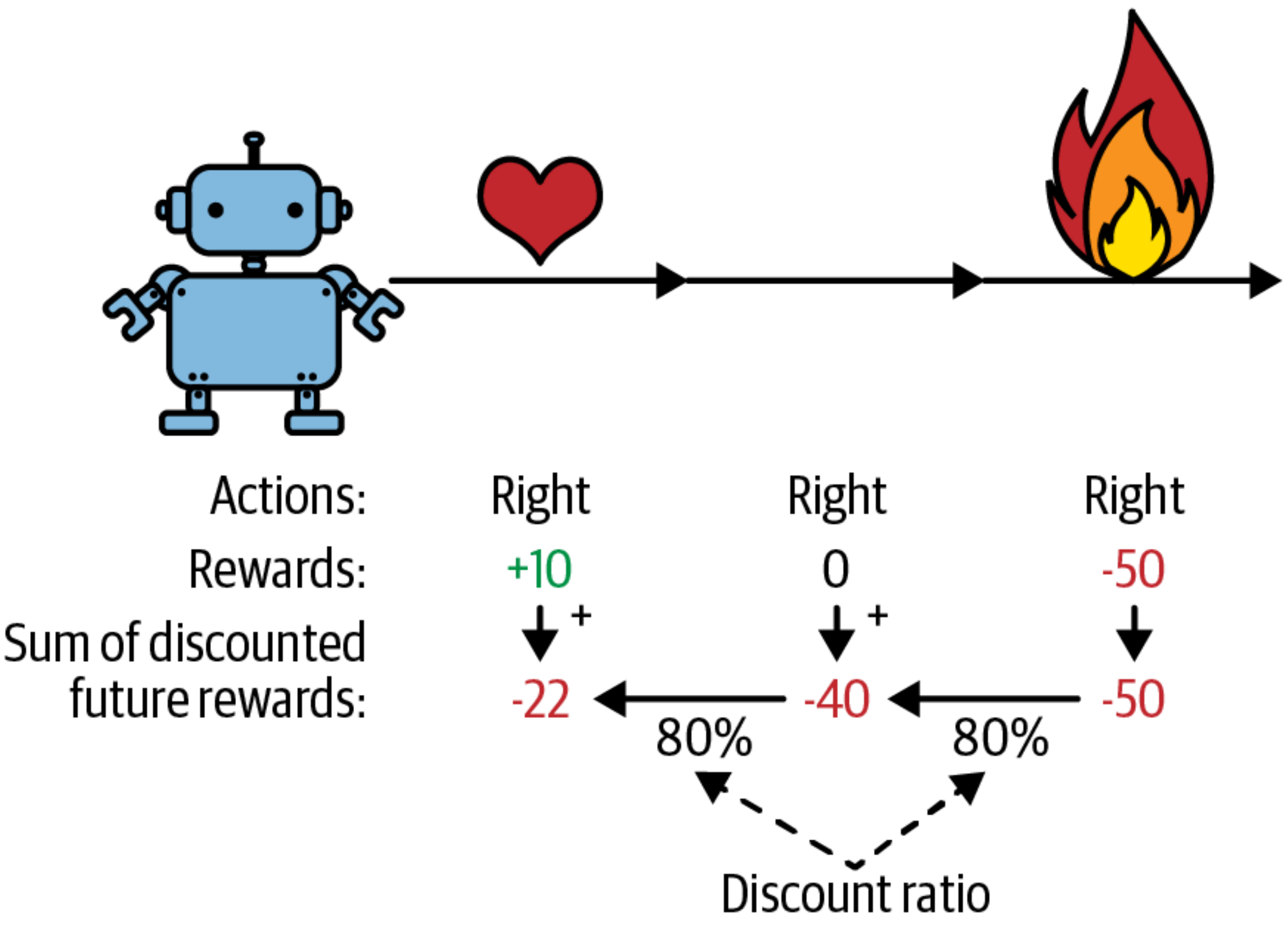


Figure 19-6. Computing an action's return: the sum of discounted future rewards

The following function computes the returns given the rewards and the discount factor:

```
def compute_returns(rewards, discount_factor):
    returns = rewards[:]  # copy the rewards
    for step in range(len(returns) - 1, 0, -1):
        returns[step - 1] += returns[step] * discount_factor

    return torch.tensor(returns)
```

This function produces the expected result:

```
>>> compute_returns([10, 0, -50], discount_factor=0.8)
tensor([-22., -40., -50.])
```

If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

Now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Solving the CartPole using Policy Gradients

As discussed earlier, policy gradient algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular PG algorithm, called *REINFORCE* (or *Monte Carlo PG*), was introduced back in 1992 by Ronald Williams.[11] It has many variants, with various tweaks, but the general principle is this:

1. First, let the neural network policy play the game for an episode, and record the rewards and estimated log probabilities.
2. Then compute each action's return, using the function defined in the previous section.
3. If an action's return is positive, it means that the action was probably good, and you want to make this action even more likely to be chosen in the future. Conversely, if an action's return is negative, you want to make this action *less* likely. To achieve this, you can minimize the REINFORCE loss defined in Equation 19-1: this will maximize the expected discounted rewards.

Equation 19-1. REINFORCE loss

In this equation, $\pi_\theta(a_t | s_t)$ is the policy network's estimated probability for action a_t given state s_t (where t is the timestep), and r_t is the observed return of this action.

Let's use PyTorch to implement this algorithm. First, we need a function to let the policy network play an episode, and record the rewards and log probabilities:

```
def run_episode(model, env, seed=None):
    log_probs, rewards = [], []
    obs, info = env.reset(seed=seed)
    while True:  # the environment will truncate the episode if it is too long
        action, log_prob = choose_action(model, obs)
        obs, reward, done, truncated, _info = env.step(action)
        log_probs.append(log_prob)
```

```
            rewards.append(reward)
            if done or truncated:
                return log_probs, rewards
```

The function first resets the environment to start a new episode. For reproducibility, we pass a seed to the `reset()` method. Then comes the game loop: at each iteration, we pass the current environment state (i.e., the last observation) to the `choose_action()` method we defined earlier. It returns the chosen action and its log probability. We then call the environment's `step()` method to execute the action. This returns a new observation (a NumPy array), a reward, two booleans indicating whether the game is over or truncated, and an info dict (which we can safely ignore in the case of CartPole). We record the log probabilities and rewards in two lists, which we return when the episode is over.

We can finally write the training function:

```
def train_reinforce(model, optimizer, env, n_episodes, discount_factor):
    for episode in range(n_episodes):
        seed = torch.randint(0, 2**32, size=()).item()
        log_probs, rewards = run_episode(model, env, seed=seed)
        returns = compute_returns(rewards, discount_factor)
        std_returns = (returns - returns.mean()) / (returns.std() + 1e-7)
        losses = [-logp * rt for logp, rt in zip(log_probs, std_returns)]
        loss = torch.cat(losses).sum()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print(f"\rEpisode {episode + 1}, Reward: {sum(rewards):.2f}", end=" ")
```

That's nice and short, isn't it? At each training iteration, the function runs an episode and gets the log probabilities and rewards.[12] Then it computes the return for each action. Next, it standardizes the returns (i.e., it subtracts the mean return and divides by the standard deviation, plus a small value to avoid division by zero). This standardization step is optional but it's a common and recommended tweak to the REINFORCE algorithm, as it stabilizes training. Next, the function computes the REINFORCE loss using Equation 19-1, and it performs an optimizer step to minimize the loss.

That's it, we're ready to build and train a policy network!

```
torch.manual_seed(42)
model = PolicyNetwork()
optimizer = torch.optim.NAdam(model.parameters(), lr=0.05)
train_reinforce(model, optimizer, env, n_episodes=400, discount_factor=0.95)
```

Training will take less than a minute. If you run an episode using this policy network, you will see that it perfectly balances the pole. Success!

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that its return estimates are extremely noisy, especially when good actions are mixed with bad ones. However, it is the foundation of more powerful algorithms, such as *actor-critic* algorithms (which we will discuss at the end of this chapter).

TIP

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

Moreover, the REINFORCE algorithm is quite unstable: the agent may improve for a while during training, then forget everything catastrophically, learn again, forget, learn, etc. It's a roller coaster. This is in large part due to the fact that the training samples are not independent and identically distributed (IID): indeed, the training samples consist of whatever states the agent is capable of reaching right now. As the agent progresses, it explores different parts of the environment, and it can forget everything about other parts. For example, once it learns to properly hold the pole upright, it will no longer see non-vertical poles, and it will totally forget how to handle them. And this issue gets much worse with more complex environments.

NOTE

Reinforcement learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.[13] As the researcher Andrej Karpathy put it, "[Supervised learning] wants to work. [...] RL must be forced to work". You will need time, patience, perseverance, and perhaps a bit of luck too. This is a major reason RL is not as widely adopted as regular deep learning.

So we will now look at another popular family of algorithms: *value-based methods*.

Value-Based Methods

Whereas PG algorithms directly try to optimize the policy to increase rewards, value-based methods are less direct: the agent learns to estimate the value of each state (i.e., the expected return), or the value of each action in a given state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first discuss *Markov decision processes* (MDPs).

Markov Decision Processes

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state *s* to a state *s*' is fixed, and it depends only on the pair (*s*, *s*'), not on past states. This is why we say that the system has no memory.

Figure 19-7 shows an example of a Markov chain with four states.

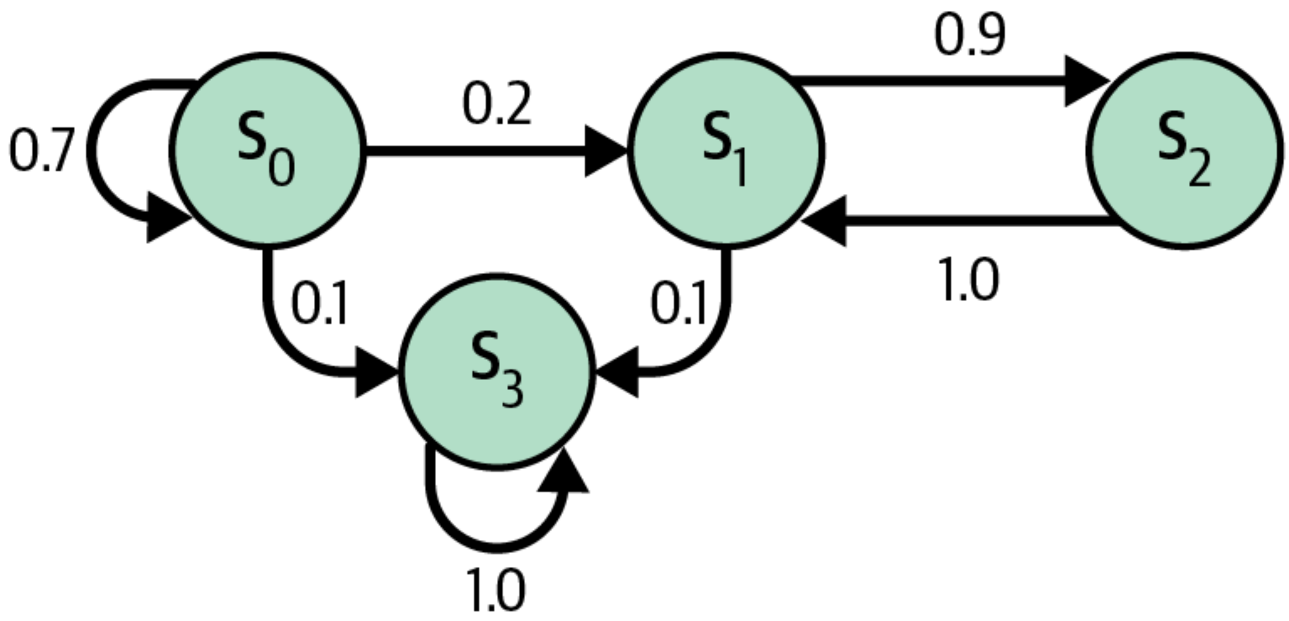


Figure 19-7. Example of a Markov chain

Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back, because no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever, since there's no way out: this is called a *terminal state*. Markov chains can have very different dynamics, and they are frequently used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by Richard Bellman.[14] They resemble Markov chains, but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

For example, the MDP represented in Figure 19-8 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

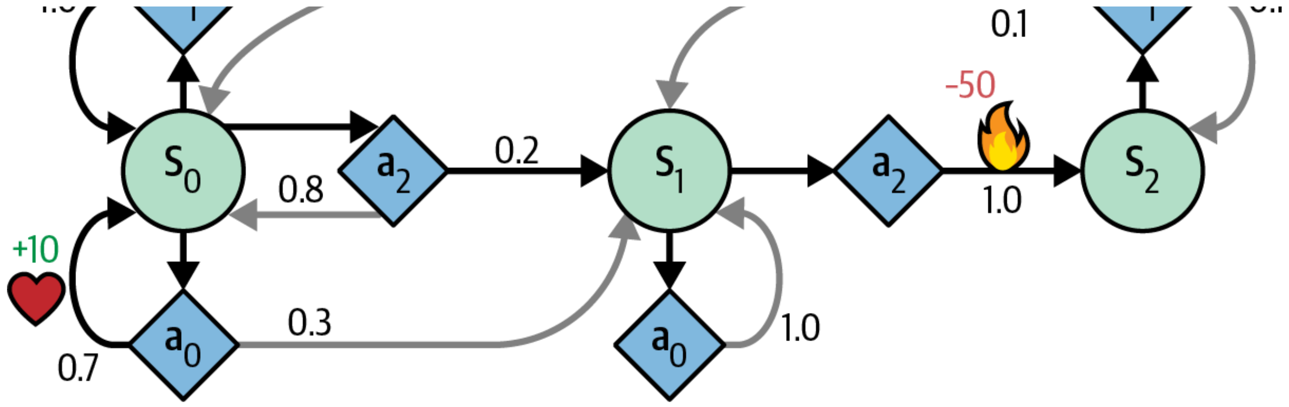


Figure 19-8. Example of a Markov decision process

If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10 and remaining in state s_0. It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_2. It can choose to stay put by repeatedly choosing action a_0, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_2 it has no choice but to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state *s*, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches the state, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman optimality equation* applies (see Equation 19-2). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 19-2. Bellman optimality equation

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a. For example, in Figure 19-8, $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a. For example, in Figure 19-8, $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: first initialize all the state value estimates to zero, and then iteratively update them using the *value iteration* algorithm (see Equation 19-3). A remarkable result is update them using the *value iteration* algorithm (see Equation 19-3). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 19-3. Value iteration algorithm

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

NOTE

This algorithm is an example of *dynamic programming*, which breaks down a complex problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-values* (quality values). The optimal Q-value of the state-action pair (s, a), noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Let's look at how it works. Once again, you start by initializing all the Q-value estimates to zero, then you update them using the *Q-value iteration* algorithm (see Equation 19-4).

Equation 19-4. Q-value iteration algorithm

Once you have the optimal Q-values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-value for that state. The fancy math notation for this is .

Let's apply this algorithm to the MDP represented in Figure 19-8. First, we need to define the MDP:

```
transition_probabilities = [  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [  # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]
possible_actions = [[0, 1, 2], [0, 2], [1]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

For example, to know the transition probability of going from s_2 to s_0 after playing action a_1, we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2, we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-values to zero (except for the impossible actions, for which we set the Q-values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf)  # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

Now let's run the Q-value iteration algorithm. It applies Equation 19-4 repeatedly, to all Q-values, for every state and every possible action:

```
gamma = 0.90  # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
                for sp in range(3)])
```

That's it! The resulting Q-values look like this:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

For example, when the agent is in state s_0 and it chooses action a_1, the expected sum of discounted future rewards is approximately 17.0.

For each state, we can find the action that has the highest Q-value:

```
>>> Q_values.argmax(axis=1)  # optimal action for each state
array([0, 0, 1])
```

This gives us the optimal policy for this MDP when using a discount factor of 0.90: in state s_0 choose action a_0, in state s_1 choose action a_0 (i.e., stay put), and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *temporal difference (TD) learning* algorithm is very similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 19-5).

Equation 19-5. TD learning algorithm

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation ,
which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So, the first line of Equation 19-5 can be rewritten like this: .

TIP

TD learning has many similarities with stochastic gradient descent, including the fact that it handles one sample at a time. Moreover, just like SGD, it can only truly converge if you gradually reduce the learning rate; otherwise, it will keep bouncing around the optimum Q-values.

For each state s, this algorithm keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later, assuming it acts optimally.

Q-Learning

Similarly, the Q-learning algorithm is an adaptation of the Q-value iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 19-6). Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values. Once it has accurate Q-value estimates (or close enough), then the optimal policy is to choose the action that has the highest Q-value (i.e., the greedy policy).

Equation 19-6. Q-learning algorithm

For each state-action pair (*s*, *a*), this algorithm keeps track of a running average of the rewards *r* the agent gets upon leaving the state *s* with action *a*, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-value estimates for the next state *s*', since we assume that the target policy will act optimally from then on.

Let's implement the Q-learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-values just like earlier, we are ready to run the Q-learning algorithm with learning rate decay (using power scheduling, introduced in Chapter 11):

```
alpha0 = 0.05  # initial learning rate
decay = 0.005  # learning rate decay
gamma = 0.90  # discount factor
state = 0  # initial state
```

```
for iteration in range(10_000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max()  # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

This algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in Figure 19-9, the Q-value iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

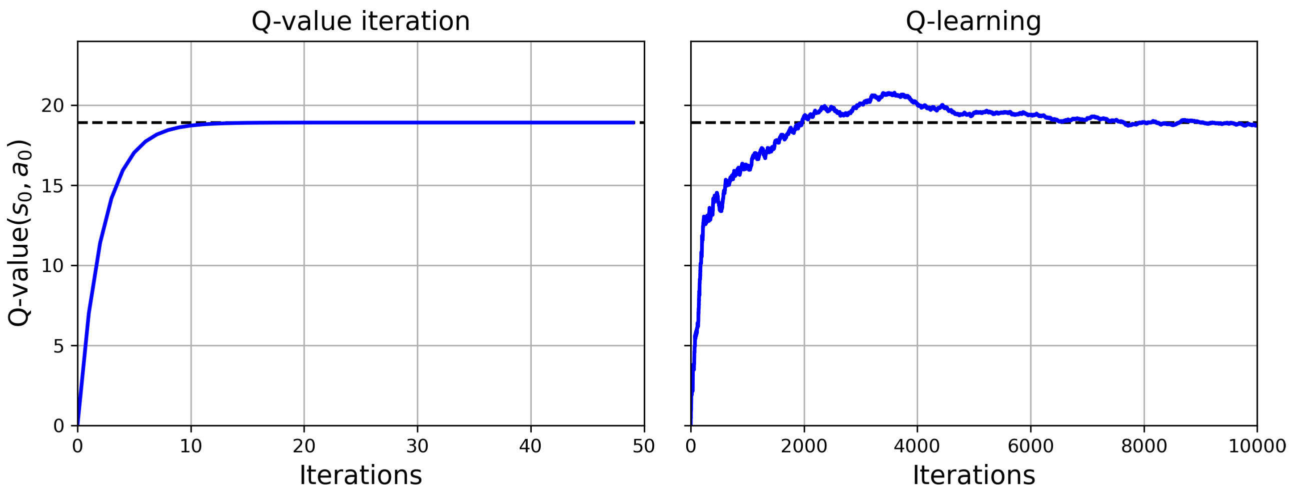


Figure 19-9. Learning curve of the Q-value iteration algorithm versus the Q-learning algorithm

The Q-learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training. For example, in the code we just ran, the policy being executed (the exploration policy) was completely random, while the policy being trained was never used. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value. Conversely, the REINFORCE algorithm is *on-policy*: it explores the world using the policy being trained. It is somewhat surprising that Q-learning is capable of learning the optimal policy by just watching an agent act randomly. Imagine learning to play golf when your teacher is a blindfolded monkey. Can we do better?

Exploration policies

Of course, Q-learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy* (ε is epsilon): at each step it acts randomly with probability *ε*, or greedily with probability 1–*ε* (i.e., choosing the action with the highest Q-value). The advantage of the *ε*-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-value estimates, as shown in Equation 19-7.

Equation 19-7. Q-learning using an exploration function

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-learning to train an agent to play *Ms. Pac-Man* (see Figure 19-1). There are about 150 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our galaxy, so there's absolutely no way you can keep track of an estimate for every single Q-value.

The solution is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector $\boldsymbol{\theta}$). This is called *approximate Q-learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., the distances of the closest ghosts, their directions, and so on) to estimate Q-values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN that is used to estimate Q-values is called a *deep Q-network* (DQN), and using a DQN for approximate Q-learning is called *deep Q-learning*.

Now, how can we train a DQN? Well, consider the approximate Q-value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know we want this approximate Q-value to be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally from then on. To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s', for all possible actions a'. We get an approximate future Q-value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-value $y(s, a)$ for the state-action pair (s, a), as shown counted value estimate, we get a target Q-value $y(s, a)$ for the state-action pair (s, a), as shown in Equation 19-8.

Equation 19-8. Target Q-value

With this target Q-value, we can run a training step using any gradient descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-value $Q_{\boldsymbol{\theta}}(s, a)$ and the target Q-value $y(s, a)$, or the Huber loss to reduce the algorithm's sensitivity to large errors. And that's the deep Q-learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a deep Q-network. In theory, we need a neural net that takes a state-action pair as input, and outputs an approximate Q-value. However, in practice it's much more efficient to use a neural net that takes only a state as input, and outputs one approximate Q-value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
class DQN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(4, 32), nn.ReLU(),
                                 nn.Linear(32, 32), nn.ReLU(),
                                 nn.Linear(32, 2))

    def forward(self, state):
        return self.net(state)
```

Our DQN is very similar to our earlier policy network, except it outputs a Q-value for each action instead of logits. Now let's define a function to choose an action based on this DQN:

```
def choose_dqn_action(model, obs, epsilon=0.0):
        if torch.rand(()) < epsilon:  # epsilon greedy policy
            return torch.randint(2, size=()).item()
        else:
            state = torch.as_tensor(obs)
            Q_values = model(state)
            return Q_values.argmax().item()  # optimal according to the DQN
```

This function takes an environment state (a single observation) and passes it to the neural net to predict the Q-values, then it simply returns the action with the largest predicted Q-value (`argmax()`). To ensure that the agent explores the environment, we use an *ε*-greedy policy, meaning we choose a random action with probability *ε*.

WARNING

DQNs generally don't work with continuous action spaces, unless you can discretize the space (which only works if it's tiny) or combine them with policy gradients. This is because the DQN agent must find the action with the highest Q-value at each step: in a continuous action space, this requires running an optimization algorithm on the Q-value function at each step, which is not practical.

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which stabilizes training by making the data distribution more consistent. Each experience will be represented as a tuple with six elements: a state *s*, the action *a* that the agent took, the resulting reward *r*, the next state *s′* it reached, a boolean indicating whether the episode ended at that point (`done`), and finally another boolean indicating whether the episode was truncated at that point. We will also need a function to sample a random batch of experiences from the replay buffer. It will return a tuple containing six tensors: one for each field.

```
def sample_experiences(replay_buffer, batch_size):
    indices = torch.randint(len(replay_buffer), size=[batch_size])
    batch = [replay_buffer[index] for index in indices.tolist()]
    return [to_tensor([exp[index] for exp in batch]) for index in range(6)]

def to_tensor(data):
    array = np.stack(data)
    dtype = torch.float32 if array.dtype == np.float64 else None
    return torch.as_tensor(array, dtype=dtype)
```

The `sample_experiences()` function takes a replay buffer and a batch size, and it randomly samples the desired number of experience tuples from the buffer. Then, for each of the six fields in the experience tuples, it extracts that field from each experience in the batch, and converts that list to a tensor using the `to_tensor()` function. Lastly, it returns the list of six tensors. The tensors all have shape [*batch size*] except for the observation tensors which have shape [*batch size*, 4].

The `to_tensor()` function takes a Python list containing observations (i.e., 64-bit NumPy arrays of shape [4]), or actions (integers), or rewards (floats), or booleans (done or truncated), and it returns a tensor of the appropriate PyTorch type. Note that the 64-bit NumPy arrays containing the observations are converted to 32-bit tensors.

The replay buffer can be any data structure that supports appending and indexing, and can limit the size to avoid blowing up memory during training. For simplicity, we will use a Python *deque*, from the standard `collections` package. This is a double-ended queue, in which elements can be efficiently appended or popped (i.e., removed) on both ends. If you set a size limit, and that limit is reached, appending an element to one end of the queue automatically pops an item from the other side: this means that each new experience replaces the oldest experience, which is exactly what we want.

TIP

Appending and popping items on the ends of a deque is very fast, but random access can be slow when the queue gets very long (e.g., 100,000 items or more). If you need a very large replay buffer, you should use a circular buffer instead (see the notebook for an implementation), or check out DeepMind's Reverb library.

Let's also create a function that will play a full episode using our DQN, and store the resulting experiences in the replay buffer. We'll run in eval mode with `torch.no_grad()` since we don't need gradients for now. For logging purposes, we'll also make the function sum up all the rewards in the episode and return the result:

```python
def play_and_record_episode(model, env, replay_buffer, epsilon, seed=None):
    obs, _info = env.reset(seed=seed)
    total_rewards = 0
    model.eval()
    with torch.no_grad():
        while True:
            action = choose_dqn_action(model, obs, epsilon)
            next_obs, reward, done, truncated, _info = env.step(action)
            experience = (obs, action, reward, next_obs, done, truncated)
            replay_buffer.append(experience)
            total_rewards += reward
            if done or truncated:
                return total_rewards
            obs = next_obs
```

Next, let's create a function that will sample a batch of experiences from the replay buffer and train the DQN by performing a single gradient descent step on this batch:

```python
def dqn_training_step(model, optimizer, criterion, replay_buffer, batch_size,
                      discount_factor):
    experiences = sample_experiences(replay_buffer, batch_size)
    state, action, reward, next_state, done, truncated = experiences
    with torch.inference_mode():
        next_Q_value = model(next_state)

    max_next_Q_value, _ = next_Q_value.max(dim=1)
    running = (~(done | truncated)).float()  # 0 if s' is over, 1 if running
    target_Q_value = reward + running * discount_factor * max_next_Q_value
    all_Q_values = model(state)
    Q_value = all_Q_values.gather(dim=1, index=action.unsqueeze(1))
    loss = criterion(Q_value, target_Q_value.unsqueeze(1))
    optimizer.zero_grad()
    loss.backward()
    loss.backward()
    optimizer.step()
```

TIP

The `torch.inference_mode()` context is like `torch.no_grad()`, plus models run in eval mode within the context, and new tensors cannot be used in backpropagation.

Here's what's happening in this code:

- The function starts by sampling a batch of experiences from the replay buffer.
- Then it uses the DQN to compute the target Q-value for each experience in the batch. For this, the code implements Equation 19-8: the DQN is used in inference mode to evaluate all the Q-values for the next state *s*', then we keep only the max Q-value since we assume that the agent will play optimally from now on, and we multiply this max Q-value with the discount factor. If the episode was over (done or truncated), then the discounted max Q-value is multiplied by zero, since we cannot expect any more rewards. Otherwise, it's multiplied by 1 (i.e., unchanged). Lastly, we add the experience's reward. All of this is performed simultaneously for all experiences in the batch.
- Next, the function uses the model again (in training mode this time) to compute all the Q-values for the current state *s*, and it uses the `gather()` method to extract just the Q-value that corresponds to the action that was actually chosen. Again, this is done simultanesouly for all experiences in the batch.
- Lastly, we compute the loss, which is typically the MSE between the target Q-values and the predicted Q-values, and we perform an optimizer step to minimize the loss.

Phew! This was the hardest part. Now we can write the main training function, and run it:

```
from collections import deque

def train_dqn(model, env, replay_buffer, optimizer, criterion, n_episodes=800,
              warmup=30, batch_size=32, discount_factor=0.95):
    totals = []
    for episode in range(n_episodes):
        epsilon = max(1 - episode / 500, 0.01)
        seed = torch.randint(0, 2**32, size=()).item()
        total_rewards = play_and_record_episode(model, env, replay_buffer,
                                                epsilon, seed=seed)
        print(f"\rEpisode: {episode + 1}, Rewards: {total_rewards}", end=" ")
        totals.append(total_rewards)
        if episode >= warmup:
            dqn_training_step(model, optimizer, criterion, replay_buffer,
                              batch_size, discount_factor)
    return totals

torch.manual_seed(42)
dqn = DQN()
optimizer = torch.optim.NAdam(dqn.parameters(), lr=0.03)
optimizer = torch.optim.NAdam(dqn.parameters(), lr=0.03)
mse = nn.MSELoss()
replay_buffer = deque(maxlen=100_000)
totals = train_dqn(dqn, env, replay_buffer, optimizer, mse)
```

The training algorithm runs for 800 episodes. At each training iteration, we make the DQN play one full episode using the `play_and_record_episode()` function, then we run one training step using the `dqn_training_step()` function. Note that we only start training after several warmup episodes, to ensure that the replay buffer contains plenty of experiences. We also linearly decrease the epsilon value for the ε-greedy policy, from 1.0 down to 0.01 after 500 episodes (then it remains at 0.01): this way, the agent's behavior will gradually become less random, focusing more on exploitation and less on exploration. The function also records the total rewards for each episode, and returns these totals: they are plotted in Figure 19-10.

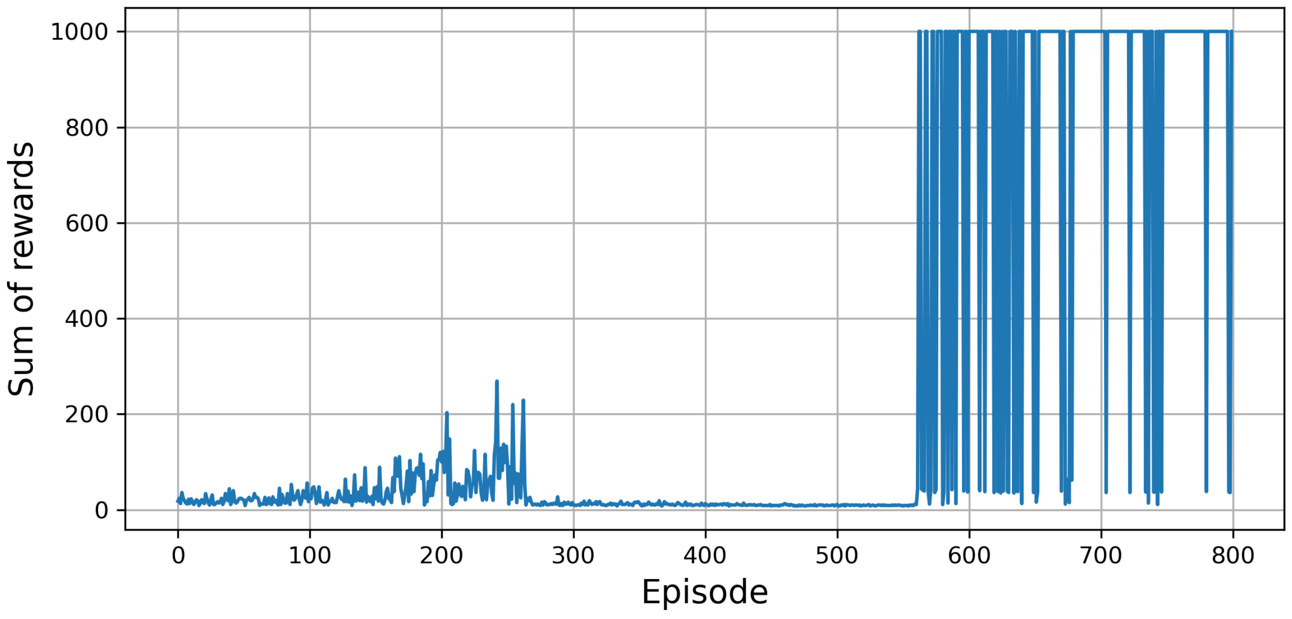


Figure 19-10. Learning curve of the deep Q-learning algorithm

NOTE

Why not plot the loss? Well, it's a poor indicator of the model's performance, it's preferable to plot the total rewards for each episode. Indeed, the loss might go down while the agent performs worse (e.g., if the agent gets stuck in one small region of the environment and the DQN starts overfitting it). Conversely, the loss could go up while the agent performs better (e.g., if the DQN was underestimating the target Q-values and it starts correctly increasing them).

The good news is that the algorithm worked: the trained agent perfectly balances the pole on the cart and reaches the maximum total reward of 1,000. The bad news is that training is completely unstable. In fact, it's even less stable than REINFORCE: I had to tweak the hyperparameters quite a bit before stumbling upon this successful training run. As you can see, the agent managed to reach a reward of 200 points after roughly 200 episodes, which isn't bad, but soon after it forgot everything and performed terribly until episode ~550, when it quickly cracked the problem.

So why is this DQN implementation unstable? Could it be the data distribution? Well, the replay buffer is quite large, so the data distribution is certainly much more stable than with the REINFORCE algorithm. So what's happening? Well, in this basic deep Q-learning implementation, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. Luckily, there are ways to improve this: let's see how.

DQN Improvements

In their 2013 paper, DeepMind researchers proposed a way to stabilize DQN training by using two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model, and its weights are copied from the online model at regular intervals (e.g., every 10,000 steps in their Atari models). This makes the Q-value targets much more stable, so the feedback loop is dampened, and its effects are much less severe. They combined this major improvement with several other tweaks: a very large replay buffer, a tiny learning rate, very long training time (50 million steps), a very slowly decreasing epsilon (over 1 million steps), and a powerful neural net (a CNN).

Then, in a 2015 paper,[15] DeepMind researchers tweaked their DQN algorithm again, increasing its performance and somewhat stabilizing training. They called this variant *double DQN*. The update was based on the observation that the target network is prone to overestimating Q-values. Indeed, suppose all actions are equally good: the Q-values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others, by pure chance. The target model will always select the largest Q-value, which will be slightly greater than the mean Q-value, most likely overestimating the true Q-value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, the researchers proposed using the online model instead of the target model when selecting the best action for the next state, and using the target model only to estimate the Q-value of this best action.

Another important improvement was the introduction of *prioritized experience replay* (PER), which was proposed in a 2015 paper[16] by DeepMind researchers (once again!). Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently?

More specifically, experiences are considered "important" if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, a, s') is very surprising, and thus probably worth learning from.[17] When an experience is recorded in the replay buffer, its priority is set to a very large value, to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience's priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^ζ, where ζ is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There's one catch, though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or else the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience's training weight as $w = (n\ P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value will depend on the task, but if you increase one, you will usually want to increase the other as well.

One last noteworthy DQN variant is the *dueling DQN* algorithm (DDQN, not to be confused with double DQN, although both techniques can easily be combined): it was introduced in yet another 2015 paper[18] by DeepMind researchers. To understand how it works, we must first note that the Q-value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s and $A(s, a)$ is the *advantage* of taking the action a in state s, compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages. The rest of the algorithm is just the same as earlier.

These techniques can be combined in various ways, as DeepMind demonstrated in a 2017 paper:[19] the paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

Speaking of combining different methods, why not combine policy gradients with value-based methods, to get the best of both world? This is the core idea behind actor-critic algorithms. Let's discuss them now.

Actor-Critic Algorithms

Actor-critics are a family of RL algorithms that combine policy gradients with value-based methods. An actor-critic is composed of a policy (the actor) and a value network (the critic), which are trained simultaneously. The actor relies on the critic to estimate the value (or advantage) of actions or states, guiding its policy updates. Since the critic can use a large replay buffer, it stabilizes training and increases data efficiency. It's a bit like an athlete (the actor) learning with the help of a coach (the critic).

Moreover, actor-critic methods support stochastic policies and continuous action spaces, just like policy gradients. So we do get the best of both worlds.

Let’s implement a basic actor-critic:

```
class ActorCritic(nn.Module):
    def __init__(self):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(4, 32), nn.ReLU(),
                                  nn.Linear(32, 32), nn.ReLU())
        self.actor_head = nn.Linear(32, 1)  # outputs action logits
        self.critic_head = nn.Linear(32, 1)  # outputs state values

    def forward(self, state):
        features = self.body(state)
        return self.actor_head(features), self.critic_head(features)
```

In the constructor, we build the actor and critic networks. In this implementation, they share the same lower layers (called the *body*): this is common practice, as it reduces the total number of parameters and thereby increases data efficiency, but it also makes training a bit less stable since it couples the actor and critic more closely (another dog-chasing-its-tail situation). The actor network takes a batch of environment states and outputs an action logit for each state (that’s the logit for action 1, just like for REINFORCE). The critic network estimates the value of each given state. The `forward()` method takes a batch of states and runs them through both networks (with a shared body), and returns the action logits and state values.

Now let’s write a function to choose an action. It’s identical to the `choose_action()` function we wrote earlier for the REINFORCE policy network, except that it also returns the state value estimated by the critic network. This will be needed for training:

```
def choose_action_and_evaluate(model, obs):
    state = torch.as_tensor(obs)
    logit, state_value = model(state)
    dist = torch.distributions.Bernoulli(logits=logit)
    action = dist.sample()
    log_prob = dist.log_prob(action)
    return int(action.item()), log_prob, state_value
```

Great! Now let’s see how to train our actor-critic. We’ll start by defining a function that will perform one training step:

```
def ac_training_step(optimizer, criterion, state_value, target_value, log_prob,
                     critic_weight):
    td_error = target_value - state_value
    actor_loss = -log_prob * td_error.detach()
    critic_loss = criterion(state_value, target_value)
    loss = actor_loss + critic_weight * critic_loss
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

First, we compute the TD error, which is the difference between the target value $y = r + \gamma V(s')$, and the state value V(s). The actor's loss is the same as in REINFORCE, except that we multiply the log probability by the TD error instead of the (standardized) return. In other words, we encourage actions that performed better than the value network expected. As for the critic's loss, it encourages the critic's value estimates V(s) to match the target values y (e.g., using the MSE). Lastly, the overall loss is a weighted sum of the actor's loss and the critic's loss. To stabilize training, it's generally a good idea to give less weight to the critic's loss. Then we perform an optimizer step to minimize the loss. Oh, and note that we call `td_error.detach()` because we don't gradient descent to affect the critic network via the actor's loss.

We'll also need a function to compute the target value:

```
def get_target_value(model, next_obs, reward, done, truncated, discount_factor):
    with torch.inference_mode():
        _, _, next_state_value = choose_action_and_evaluate(model, next_obs)

    running = 0.0 if (done or truncated) else 1.0
    target_value = reward + running * discount_factor * next_state_value
    return target_value
```

This code first evaluates V(s') using the `choose_action_and_evaluate()` function (we ignore the chosen action and its log probability). We run this in inference mode because we are computing the target: we don't want gradient descent to affect it. Next we simply evaluate the target $y = r + \gamma V(s')$. If the episode is over, then $y = r$.

With that, we have all we need to write a function that will run a whole episode and train the actor-critic at each step (we also compute the total rewards and return it when the episode is over):

```
def run_episode_and_train(model, optimizer, criterion, env, discount_factor,
                          critic_weight, seed=None):
    obs, _info = env.reset(seed=seed)
    total_rewards = 0
    while True:
        action, log_prob, state_value = choose_action_and_evaluate(model, obs)
        next_obs, reward, done, truncated, _info = env.step(action)
        target_value = get_target_value(model, next_obs, reward, done,
                                        truncated, discount_factor)
        ac_training_step(optimizer, criterion, state_value, target_value,
                         log_prob, critic_weight)
        total_rewards += reward
        if done or truncated:
            return total_rewards
        obs = next_obs
```

And lastly we can write our main training function, which just calls the `run_episode_and_train()` function many times, and returns the total rewards for each episode:

```
def train_actor_critic(model, optimizer, criterion, env, n_episodes=400,
                       discount_factor=0.95, critic_weight=0.3):
    totals = []
    model.train()
    for episode in range(n_episodes):
        seed = torch.randint(0, 2**32, size=()).item()
        total_rewards = run_episode_and_train(model, optimizer, criterion, env,
                                              discount_factor, critic_weight,
                                              seed=seed)
        totals.append(total_rewards)
        print(f"\rEpisode: {episode + 1}, Rewards: {total_rewards}", end=" ")

    return totals
```

Let's run it!

```
torch.manual_seed(42)
ac_model = ActorCritic()
optimizer = torch.optim.NAdam(ac_model.parameters(), lr=1.1e-3)
criterion = nn.MSELoss()
totals = train_actor_critic(ac_model, optimizer, criterion, env)
```

And it works! We get a very stable CartPole which collects the maximum rewards. That said, this implementation is still very sensitive to the choice of hyperparameters and random seeds, and training is still very unstable. Luckily, researchers have come up with various techniques that can stabilize the actor-critic. Here are some of the most popular:

Asynchronous advantage actor-critic (A3C)[20]

This is an important actor-critic variant introduced by DeepMind researchers in 2016 where multiple agents learn in parallel, exploring different copies of the environment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the state values, or even the Q-values, the critic estimates the *advantage* of each action (hence the second A in the name), just like in the Dueling DQN.

Advantage actor-critic (A2C)

A2C is a variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

Soft actor-critic (SAC)[21]

SAC is an actor-critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training, and makes it less likely to repeatedly execute the same action when the critic produces imperfect estimates. This algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly).

Proximal policy optimization (PPO)[22]

This algorithm by John Schulman and other OpenAI researchers is based on A2C, but it clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *trust region policy optimization*[23] (TRPO) algorithm, also by OpenAI. OpenAI made the news in April 2019 with its AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*.

The last two algorithms, SAC and PPO, are among the most widely used RL algorithms today, and several libraries provide easy to use and highly optimized implementations. For example, let's use the popular Stable-Baselines3 library to train a PPO agent on the Breakout Atari game.

TIP

Which RL algorithm should you use? PPO is a great general-purpose RL algorithm—a good bet if you're not sure. SAC is the most sample efficient for continuous action tasks, making it ideal for robotics. DQN remains strong for discrete tasks such as Atari games or board games.

Mastering Atari Breakout using Stable-Baseline3's PPO Implementation

Since Stable-Baselines3 (SB3) is not installed by default on Colab, we must first run `%pip install -q stable_baselines3`, which will take a couple of minutes. However, if you are running the code on your own machine and you followed the installation instructions at *https://homl.info/installp*, then it's already installed.

Next, we must create an ALE interface: it will run the Atari 2600 emulator and allow Gymnasium to interface with it (the Atari games will appear in the list of available environments):

```
import ale_py

ale = ale_py.ALEInterface()
```

Atari games were stored on read-only memory (ROM) cartridges. These ROMs can now be dowloaded freely and used for research and educational purposes. On Colab, they are preinstalled, but if you followed the installation instructions at *https://homl.info/installp*, then you must download them now and place them in the `ale_py.roms` folder. For this, you can use the AutoROM library (which is already installed, if you followed the installation instructions), like this:

```
from pathlib import Path
from AutoROM import main as autorom

autorom(accept_license=True, source_file=None,  # you accept the Atari license
        install_dir=Path(ale_py.roms.__path__[0]), quiet=False)
```

NOTE

The pip package for `gymnasium[atari]` automatically downloads the ROMs, so there's no need to use AutoROM. However, the installation instructions at *https://homl.info/installp* use Anaconda, and unfortunately the gymnasium-atari package does not currently download the ROMs automatically.

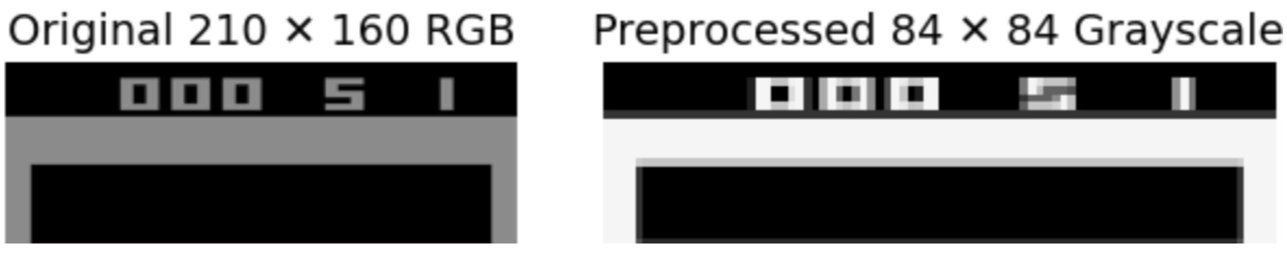
Now that we have SB3, the ALE interface, and the ROMs, we are ready to create the Breakout environment. But instead of creating it using Gymnasium directly, we will use SB3's `make_atari_env()` function: it creates a wrapper environment containing multiple Breakout environments that will run in parallel. Each observation from the wrapper environment will contain one observation for each Breakout environment. Similarly, the wrapper environment's `step()` function will take an array containing one action for each Breakout environment. Lastly, the wrapper environment will take care of preprocessing the images, converting them from 210 × 160 RGB images to 84 × 84 grayscale images. Very convenient! So let's create an SB3 environment containing 4 Breakout environments, and reset it to get an observation:

```
from stable_baselines3.common.env_util import make_atari_env

envs = make_atari_env("BreakoutNoFrameskip-v4", n_envs=4)
obs = envs.reset()  # a 4 × 84 × 84 × 1 NumPy array (note: no info dict)
```

TIP

The `env.get_images()` method returns the original images, before preprocessing (see Figure 19-11).



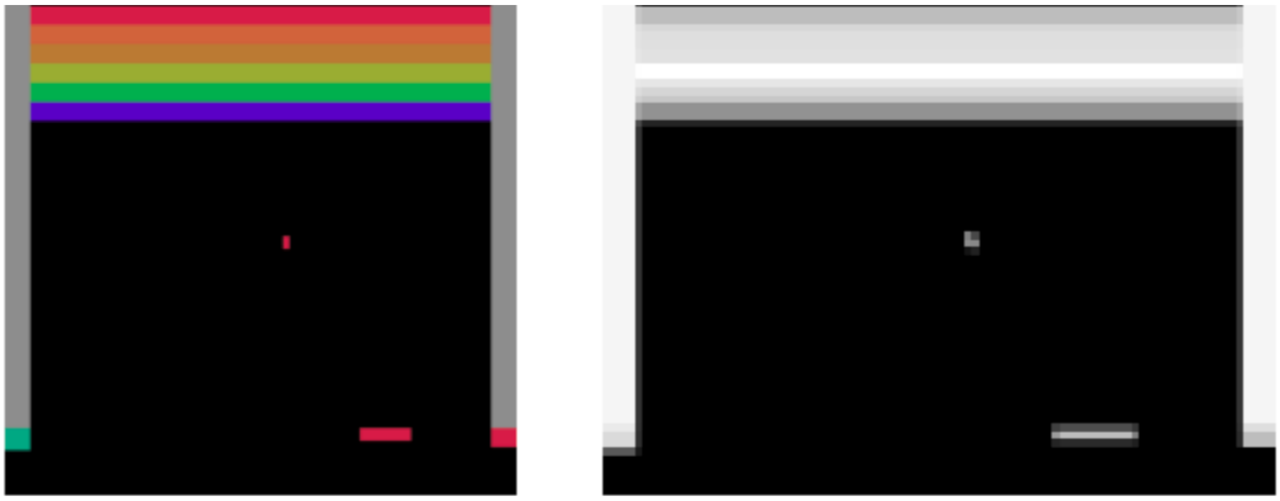


Figure 19-11. A Breakout frame before (left) and after (right) preprocessing

The ALE interface runs at 60 frames per second, which is quite fast, so consecutive frames look very similar, which wastes computation. To avoid this, the default Breakout environment repeats each action 4 times and returns only the final observation: this is called *frame skipping*. However, instead of skipping the frames, it's preferable to stack them into a single 4-channel image, and use that as the observation. For this, we must first avoid frame skipping: this is why we used the "BreakoutNoFrameskip-v4" environment rather than "Breakout-v4".[24] Then, we must wrap the environment in a `VecFrameStack`: this wrapper environment will repeat each action several times (4 in our case) and stack the resulting frames along the channels dimension (i.e., the last one):

```
from stable_baselines3.common.vec_env import VecFrameStack

envs_stacked = VecFrameStack(envs, n_stack=4)
obs = envs_stacked.reset()  # returns a 4 × 84 × 84 × 4 NumPy array
```

Now let's create a PPO model with some good hyperparameters:

```
from stable_baselines3 import PPO

ppo_model = PPO("CnnPolicy", envs_stacked, device=device, learning_rate=2.5e-4,
                batch_size=256, n_steps=256, n_epochs=4, clip_range=0.1,
                vf_coef=0.5, ent_coef=0.01, gamma=0.99, verbose=0)
```

That's a lot of arguments! Let's see what they do:

- The first argument is the policy network. Since we specified "CnnPolicy", SB3 will build a good CNN for us, based on the chosen algorithm (PPO in this case) and the observation space. If you're curious, take a look at `ppo_model.policy` to see the CNN's architecture: it's a deep CNN with an actor head (for the action logits) and a critic head (for the state values). There are four possible actions: left, right, fire (to launch the ball), and no-op (do nothing). If you prefer to use a custom neural net, you must create a subclass of the `ActorCriticPolicy` class, located in the `stable_baselines3.common.policies` module. See SB3's documentation for more details.

- `env`, `device`, `learning_rate` and `batch_size` are self-explanatory.
- `n_steps` is the number of environment steps to run (per environment) before each policy update.
- `n_epochs` is the number of training steps to run on each batch during optimization.
- `clip_range` limits the magnitude of the policy updates, to avoid large changes that might cause catastrophic forgetting.
- `vf_coef` is the weight of the value function loss in the total loss (similar to our actor-critic's `critic_weight` hyperparameter).
- `ent_coef` is the weight of the entropy term that encourages exploration.
- `gamma` is the discount rate.
- `verbose` is the logging verbosity (0 = silent, 1 = info, 2 = debug).

And now let's start training. The code below will train the model for 30 million steps. This will take many hours and will probably be too long for a Colab session (unless you get a paid subscription), so the notebook also includes code to download the trained model if you prefer. Whenever you train a model for a long time, it's important to save checkpoints at regular intervals (e.g., every 100,000 calls to the `step()` method), to avoid having to start from scratch in case of a crash or a power outage. For this, we can create a checkpoint callback and pass it to the `learn()` method:

```
from stable_baselines3.common.callbacks import CheckpointCallback

cb = CheckpointCallback(save_freq=100_000, save_path="my_ppo_breakout.ckpt")
ppo_model.learn(total_timesteps=30_000_000, progress_bar=True, callback=cb)
ppo_model.save("my_ppo_breakout")  # save the final model
```

NOTE

The `save_freq` argument counts calls to the `step()` method. Since there are 4 environments running in parallel, 50,000 calls correspond to 200,000 total timesteps.

To see the progress during training, one option is to load the latest checkpoint in another notebook, and try it out. A simpler option is to use TensorBoard to visualize the learning curves, especially the mean reward per episode. For this, you must first activate the TensorBoard extension in Colab or Jupyter by running `%load_ext tensorboard` (this is done at the start of this chapter's notebook). Next, you must start the TensorBoard server, point it to a log directory, and choose the TCP port it will listen on. The following "magic" command (i.e., starting with a %) will do that and also open up the TensorBoard client interface directly inside Colab or Jupyter:

```
tensorboard_logdir = "my_ppo_breakout_tensorboard"  # path to the log directory
%tensorboard --logdir={tensorboard_logdir} --port 6006
```

Next, you must tell the PPO model where to save its TensorBoard logs. This is done when creating the model:

```
ppo_model = PPO("CnnPolicy", [...], tensorboard_log=tensorboard_logdir)
```

And that's it. Once you start training, you will see the learning curves change every 30 seconds or so in the TensorBoard interface (or click the refresh button). The most important metric to track is the `rollout/ep_rew_mean` which is the mean reward per episode: it should slowly ramp up, even though it will sometimes go down a bit. After 1 million total steps it will typically reach around 20: that's not a very good agent. But if you let training run for 10 million steps, it should reach human level. And after 50 million steps, it will generally be superhuman.

Congratulations, you know how to train a superhuman AI! You can try it out like this:

```
ppo_model = PPO.load("my_ppo_agent_breakout")  # or load the best checkpoint
eval_env = make_atari_env("BreakoutNoFrameskip-v4", n_envs=1, seed=42)
eval_stacked = VecFrameStack(eval_env, n_stack=4)
frames = []
obs = eval_stacked.reset()
for _ in range(5000):  # some limit in case the agent never loses
    frames.append(eval_stacked.render())
    action, _ = ppo_model.predict(obs, deterministic=True) # for reproducibility
    obs, reward, done, info = eval_stacked.step(action)
    if done[0]:  # note: there's no `truncated`
        break

eval_stacked.close()
```

This will capture all the frames during one episode. You can render them as an animation using Matplotlib (see the notebook for an example). If you trained the agent for long enough (or used the pretrained model), you will see that the agent plays pretty well, and even found the strategy of digging tunnels on the sides and sending the ball through them: that's one of the best strategies in this game!

Overview of Some Popular RL Algorithms

Before we close this chapter, let's take a brief look at a few other popular algorithms:

AlphaGo[25]

AlphaGo uses a variant of *Monte Carlo tree search* (MCTS) based on deep neural networks to beat human champions at the game of Go. MCTS was invented in 1949 by Nicholas Metropolis and Stanislaw Ulam. It selects the best move after running many simulations, repeatedly exploring the search tree starting from the current position, and spending more time on the most promising branches. When it reaches a node that it hasn't visited before, it plays randomly until the game ends, and updates its estimates for each visited node (excluding the random moves), increasing or decreasing each estimate depending on the final outcome.

AlphaGo is based on the same principle, but it uses a policy network to select moves, rather than playing randomly. This policy net is trained using policy gradients. The original algorithm involved three additional neural networks, and was more complicated, but it was simplified in the AlphaGo Zero paper,[26] which uses a single neural network to both select moves and evaluate game states. The AlphaZero paper[27] generalized this algorithm, making it capable of tackling not only the game of Go, but also chess and shogi (Japanese chess). Lastly, the MuZero paper[28] continued to improve upon this algorithm, outperforming the previous iterations even though the agent starts out without even knowing the rules of the game!

NOTE

The rules of the game of Go were hardcoded into AlphaGo. In contrast, MuZero gradually learns a model of the environment: given a state *s* and an action *a*, it learns to predict the reward *r* and the probability of reaching state *s'*. Having a model of the environment (hardcoded or learned) allows these algorithms to plan ahead (in this case using MCTS). For this reason, both of these algorithms belong to the broad class of *model-based RL* algorithms. In contrast, policy gradients, value-based methods, and actor-critic methods are all *model-free RL* algorithms: they have a policy model and/or a value model, but not an *environment* model.

Curiosity-based exploration[29]

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards, and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades.

How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, it finds it boring to lose because the game starts over, so it learns to avoid it.

Open-ended learning (OEL)

The objective of OEL is to train agents capable of endlessly learning new and interesting tasks, typically generated procedurally. We're not there yet, but there has been some amazing progress over the last few years. For example, a 2019 paper[30] by a team of researchers from Uber AI introduced the *POET algorithm*, which generates multiple simulated 2D environments with bumps and holes, and trains one agent per environment: the agent's goal is to walk as fast as possible while avoiding the obstacles.

The algorithm starts out with simple environments, but they gradually get harder over time: this is called *curriculum learning*. Moreover, although each agent is only trained within one environment, it must regularly compete against other agents, across all environments. In each environment, the winner is copied over and it replaces the agent that was there before. This way, knowledge is regularly transferred across environments, and the most adaptable agents are selected.

In the end, the agents are much better walkers than agents trained on a single task, and they can tackle much harder environments. Of course, this principle can be applied to other environments and tasks as well. If you're interested in OEL, make sure to check out the Enhanced POET paper,[31] as well as DeepMind's 2021 paper[32] on this topic.

TIP

If you'd like to learn more about reinforcement learning, check out the book *Reinforcement Learning* by Phil Winder (O'Reilly).

We covered many topics in this chapter: we learned about policy gradient methods, we implemented the REINFORCE algorithm to solve the CartPole problem using Gymnasium, we explored Markov chains and Markov decision processes, which led us to value-based methods, and we implemented a deep Q-Learning model, then we discussed actor-critic methods, and we used the Stable-Baselines3 library to implement a PPO model that beat the Atari game Breakout. Lastly, we took a peek at some of the other areas of RL, including model-based RL and more. Reinforcement learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity: there is a whole world to explore!

Exercises

1. How would you define reinforcement learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a reinforcement learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay buffer?
7. What is an off-policy RL algorithm? What are the benefits?
8. What is a model-based RL algorithm? Can you give some examples?

8. What is a model-based RL algorithm? Can you give some examples?
9. Use policy gradients to solve Gymnasium's LunarLander-v2 environment.
10. Solve the BipedalWalker-v3 environment using the RL algorithm of your choice.
11. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install PyTorch on the Pi, and go wild! Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using deep learning: for example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, or through the *ageron/handson-mlp* GitHub project.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamt of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy to build, it will be immensely rewarding.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

—Aurélien Geron, August 9, 2025

[1] For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press).

[2] DeepMind was bought by Google for over $500 million in 2014.

[3] Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv preprint

arXiv:1312.5602 (2013).

[4] Volodymyr Mnih et al., “Human-Level Control Through Deep Reinforcement Learning”, *Nature* 518 (2015): 529–533.

[5] Check out the videos of DeepMind’s system learning to play *Space Invaders*, *Breakout*, and other video games at *https://homl.info/dqn3*.

[6] Images (a), (d), and (e) are in the public domain. Image (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Image (c) is reproduced from Wikipedia; it was created by user Stevertigo and released under Creative Commons BY-SA 2.0.

[7] It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the “gene pool”.

[8] If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring’s genome (in this case a set of policy parameters) is randomly composed of parts of its parents’ genomes.

[9] One interesting example of a genetic algorithm used for reinforcement learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm. Also check out *evolutionary policy optimization* (EPO), proposed in 2025, where a master agent learns stably and efficiently from the experiences of a population of agents.

[10] This is called *gradient ascent*. It’s just like gradient descent, but in the opposite direction: maximizing instead of minimizing.

[11] Ronald J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Leaning”, *Machine Learning* 8 (1992) : 229–256.

[12] We generate a new random seed for each episode using `torch.randint()`: this ensures that each episode is different, yet the whole training process is reproducible if we set PyTorch’s random seed before calling `train_reinforce()`.

[13] A great 2018 post by Alex Irpan nicely lays out RL’s biggest difficulties and limitations.

[14] Richard Bellman, “A Markovian Decision Process”, *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.

[15] Hado van Hasselt et al., “Deep Reinforcement Learning with Double Q-Learning”, *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.

[16] Tom Schaul et al., “Prioritized Experience Replay”, arXiv preprint arXiv:1511.05952 (2015).

[17] It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience’s importance (see the PER paper for some examples).

[18] Ziyu Wang et al., “Dueling Network Architectures for Deep Reinforcement Learning”, arXiv preprint arXiv:1511.06581 (2015).

[19] Matteo Hessel et al., “Rainbow: Combining Improvements in Deep Reinforcement Learning”, arXiv pre-

print arXiv:1710.02298 (2017): 3215–3222.

20 Volodymyr Mnih et al., “Asynchronous Methods for Deep Reinforcement Learning”, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.

21 Tuomas Haarnoja et al., “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”, *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.

22 John Schulman et al., “Proximal Policy Optimization Algorithms”, arXiv preprint arXiv:1707.06347 (2017).

23 John Schulman et al., “Trust Region Policy Optimization”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

24 The “v4” suffix is the version number, it’s unrelated to frame skipping or the number of parallel environments.

25 David Silver et al., “Mastering the Game of Go with Deep Neural Networks and Tree Search”, *Nature* 529 (2016): 484–489.

26 David Silver et al., “Mastering the Game of Go Without Human Knowledge”, *Nature* 550 (2017): 354–359.

27 David Silver et al., “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”, arXiv preprint arXiv:1712.01815.

28 Julian Schrittwieser et al., “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”, arXiv preprint arXiv:1911.08265 (2019).

29 Deepak Pathak et al., “Curiosity-Driven Exploration by Self-Supervised Prediction”, *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

30 Rui Wang et al., “Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions”, arXiv preprint arXiv:1901.01753 (2019).

31 Rui Wang et al., “Enhanced POET: Open-Ended Reinforcement Learning Through Unbounded Invention of Learning Challenges and Their Solutions”, arXiv preprint arXiv:2003.08536 (2020).

32 Open-Ended Learning Team et al., “Open-Ended Learning Leads to Generally Capable Agents”, arXiv preprint arXiv:2107.12808 (2021).