第 3 章 基于表格的强化学习方法

通过前面章节的学习，相信读者已经大致理解了 Q 函数的意义，以及如何使用时序差分方法来优化 Q 函数。根据 Q 函数的定义，只要 Q 函数的计算精度足够高，就能在各种状态下计算最优的动作，从而玩好游戏，取得好成绩。

理解了这些理论知识之后相信读者和笔者一样跃跃欲试，让我们试一试前面的理论是否成立，通过一个实际的实验来检验。

3.1 代码运行环境说明

从本章开始，本书将进入代码实战的内容，读者搭建好运行环境以后，跟随本书的代码开始调试代码，在强化学习的过程中调试代码是必不可少的，所谓“纸上得来终觉浅，绝知此事须躬行”，实际调试代码能帮助读者更好地理解算法的运算过程，务必躬行。

本书的代码在以下环境中测试无误，读者应尽量和以下环境保持一致，避免不必要的环境调试。

表 3-1 中的 Python 和 PyTorch 的版本号并不是特别敏感，一般来讲不同版本的 Python 和 PyTorch 也可以正常运行，如果读者使用不同版本的 Python 或者 PyTorch 运行出错，则建议使用表 3-1 中的版本再试。

表 3-1 本书代码运行环境

包	版 本
Python	3.9
PyTorch	1.12.1(CPU)
Gym	0.26.2
pettingzoo	1.23.1

表 3-1 中特别注明了 PyTorch 使用 CPU 版本，其实使用 GPU 版本也是可以的，只是本书的重点是对原理层面进行讲解，实战代码主要用于理论验证，所以不是在特别复杂的游戏环境中验证，计算量不大，使用 CPU 计算也是可以接受的。考虑到 GPU 版本的 PyTorch 比较复杂，更加推荐读者使用 CPU 来运行本书的代码。

表 3-1 中的 Gym 包的版本比较敏感，推荐使用表中推荐的版本，如果不一致，则很容易抛出异常。

本书的代码主要在 Jupyter Notebook 环境中运行，建议读者使用该环境运行，可以很方便地看到游戏运行的动画。

3.2 游戏环境

3.2.1 Gym 包介绍

既然要实际地运行强化学习的程序，当然需要一个让强化学习程序活动的空间，这个空间就是要求解的游戏环境，创建虚拟游戏环境，有一个在强化学习领域非常流行的工具包，叫作 Gym，如图 3-1 所示。

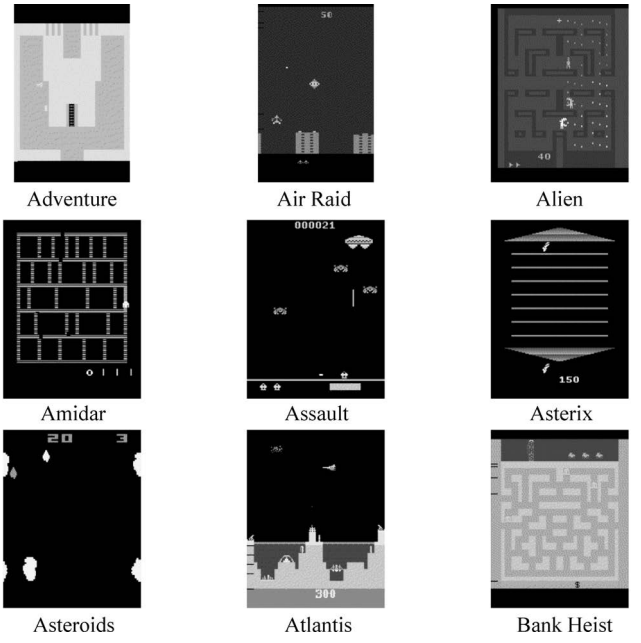


图 3-1 Gym 提供的部分游戏环境

由于 Gym 包提供了很多虚拟的游戏环境，并且提供了简单的接口，所以能方便地和这些环境交互，获取环境中的状态等，学习强化学习 Gym 包几乎是必备的。

3.2.2 定义游戏环境

本章所需要使用的游戏环境也是由 Gym 包提供的，事实上读者在前面章节中已经见过该游戏环境，也就是前面介绍过的冰湖游戏环境，使用 Gym 包获取该游戏环境十分简单，代码如下：

```
#第 3 章/定义环境
import gym


class MyWrapper(gym.Wrapper):

    def __init__(self):
        #is_slippery 控制会不会滑动
        env = gym.make('FrozenLake-v1',
                       render_mode='rgb_array',
                       is_slippery=False)

        super().__init__(env)
        self.env = env

    def reset(self):
        state, _ = self.env.reset()
        return state

    def step(self, action):
        state, reward, terminated, truncated, info = self.env.step(action)
        over = terminated or truncated

        #走一步扣一分,逼迫机器人尽快结束游戏
        if not over:
            reward = -1

        #掉坑扣 100 分
        if over and reward == 0:
            reward = -100

        return state, reward, over

    #打印游戏图像
    def show(self):
        from matplotlib import pyplot as plt
        plt.figure(figsize=(3, 3))
        plt.imshow(self.env.render())
        plt.show()


env = MyWrapper()

env.reset()

env.show()
```

以上代码的运行结果如图 1-4 所示。

虽然在前面已经大致介绍过该游戏环境，但是考虑章节的独立性，此处再简单地对该环境进行介绍。该环境的特征大致如下：

(1) 该游戏的目标是控制小人获得礼物，中途不能掉到坑里。

(2) 可以看出该游戏环境有 4 行 4 列共 16 个格子。

(3) 小人的动作空间是上、下、左、右 4 个动作。

(4) 这个游戏有两种判定结束的方式，一种是小人获得礼物，另一种是小人掉进了坑里。

观察该游戏环境，可以发现这个游戏可以被无止境地玩下去，例如反复地左右来回走，可能会出现永远不结束的玩法，为了防止这种情况的发生，在环境定义的代码中给每步走在地面的动作都设定一个较小的负反馈值，此处设定为－1 分，这样做是为了告诉机器人要尽快结束游戏，因为每走一步都会造成－1 的负反馈，这样可以强迫机器尽快结束游戏，避免无意义地来回走动，游戏结束得越快越好，尽量使用最少的步数来获得礼物。

为了告诉这个机器人不要掉到坑里，可以在掉坑时给予反馈－100 分，这样可以告诉机器人要尽量避免掉到坑里。

综上所述，这个游戏的目标是以最少的步数完成游戏，尽量以获得礼物的形式结束游戏，而不是掉到坑里。

3.2.3 游戏环境操作方法介绍

游戏的环境定义好了，也许读者想试试自己玩这个游戏，下面给出该游戏的操作方法，代码如下：

```
#第 3 章/试玩游戏
env.reset()

action = 1
next_state, reward, over = env.step(action)
print(next_state, reward, over)

env.show()
```

在上面这段代码中，首先调用了环境的 reset()函数，这个函数能把游戏复位，也就是回到最初始的状态。

然后定义了 action＝1，action 就是动作，在冰湖游戏环境中一共有 4 种动作，分别是上、下、左、右，分别使用数字 0、1、2、3 来表示。这里简单地定义了 action＝1，表示向下走一步的动作。

接着调用环境的 step()函数，表明要在环境中执行一个动作，该函数接受一个动作作为参数，这里输入前面定义好的 action 即可。

step()函数有 3 个返回值，分别是 next_state、reward、over，下面分别说明这 3 个值的含义。

(1) next_state：执行动作之后环境的状态发生了改变，next_state 即改变后的状态，也就是下一个时刻的 state。

(2) reward：执行动作之后环境会给予一个反馈，以告诉机器人这一步动作有多好，或者有多不好，这个反馈即 reward。

(3) over：执行动作可能会导致游戏结束，例如掉进坑里，或者获得礼物，变量 over 表明到此时此刻为止，游戏是否已经结束，所以 over 为布尔值。

这样就完成了一步动作的调用，下面是输出的内容：

```
4 -1 False
```

从上面的输出可以看出，在游戏环境的初始状态下执行 1 这个动作，将导致状态变化为 4，环境给予了反馈－1，这一步动作的执行并没有导致游戏结束。

最后调用了环境的 show()函数，这会打印游戏的图像，输出如图 3-2 所示。

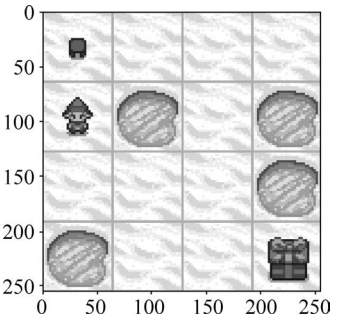


图 3-2　执行动作 1 之后的冰湖游戏环境

小人确实往下走了一格，所以动作执行是成功的。重复上述步骤就可以手动玩这个游戏了。

3.3　定义 Q 表

如本章标题所写，本章要使用基于表格的方式来求解强化学习问题，对于图 3-2 所示的冰湖游戏环境，由于复杂度比较低，所以可以使用表格来求解，后续会有基于神经网络的方式来求解，用于应对更复杂的游戏环境。

既然要基于表格求解，此处先把要求解的表格定义出来，根据前两章节的学习，了解到在强化学习任务中，主要的任务往往就是求解 Q 表，而 Q 表的定义，是指在所有状态下做出所有动作的预估分数，如表 3-2 所示。

表 3-2 冰湖环境的 Q 表

行	列	上	下	左	右
第 1 行	第 1 列	0	0	0	0
第 1 行	第 2 列	0	0	0	0
第 1 行	第 3 列	0	0	0	0
第 1 行	第 4 列	0	0	0	0
第 2 行	第 1 列	0	0	0	0
第 2 行	第 2 列	0	0	0	0
第 2 行	第 3 列	0	0	0	0
第 2 行	第 4 列	0	0	0	0
第 3 行	第 1 列	0	0	0	0
第 3 行	第 2 列	0	0	0	0
第 3 行	第 3 列	0	0	0	0
第 3 行	第 4 列	0	0	0	0
第 4 行	第 1 列	0	0	0	0
第 4 行	第 2 列	0	0	0	0
第 4 行	第 3 列	0	0	0	0
第 4 行	第 4 列	0	0	0	0

如表 3-2 所示，Q 表评估了在各种状态下做各个动作的分数，此时此刻 Q 表还是空的，没有填充内容，在后续的训练过程中将逐渐修正该 Q 表，最终机器人的行动就是根据该 Q 表决定的，所以 Q 表的质量决定了算法的性能。

定义 Q 表的代码如下：

```
#第 2 章/初始化 Q 表
import numpy as np

#定义了每种状态下每个动作的价值
Q = np.zeros((16, 4))

Q
```

运行结果如下：

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
```

```
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

Q 表定义了每种状态下执行每个动作的价值，从图 3-2 可以看出，游戏环境共有 $4\times4=16$ 种状态，在每种状态当中都可以做上、下、左、右 4 个动作，所以 Q 表是 16×4 的矩阵，初始化时让值全部为 0 即可。

此时的 Q 表还是初始化状态，最终目标是要在 Q 表当中计算出 Q 函数的值。

3.4　强化学习的一般过程

强化学习算法有很多种，但是万变不离其宗，大多数算法遵循以下一般过程，如图 3-3 所示。

在强化学习算法的整个工作过程中，一般要经历这样一个循环：

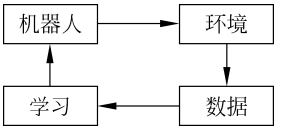


图 3-3　强化学习的一般过程

(1) 机器人和环境交互。

(2) 环境状态改变，产生反馈等数据。

(3) 机器人从环境产生的这些数据中学习、进步。

(4) 学习后的机器人使用不同的策略和环境交互。

(5) 由于机器人的策略发生了改变，所以环境也会产生不同的数据。

(6) 机器人再次从这些数据中学习、进步。

(7) 重复以上步骤，直至机器人停止进步，这意味着机器人的能力已经达到极限。

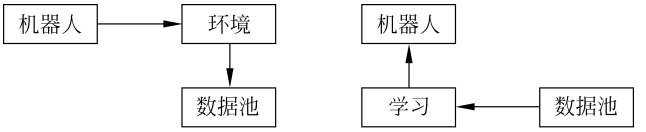
3.4.1　数据池的必要性

通过以上步骤的梳理可以发现，在整个训练过程中机器人要反复多次地和环境交互，这样才能产生数据，进而学习、进步。

在 Gym 这样的轻量式的环境中这样做也许问题不大，因为环境的交互速度很快，不会占用大量的计算资源，但是在复杂的环境中，和环境交互的成本可能很高，例如在某些驾驶模拟环境中，“左转方向盘”这个动作的执行可能就需要 1s 时间，1s 已经是一个让人无法忍受的漫长时间了，因为在整个训练过程中这样的动作可能要执行成千上万次，累计起来这样的成千上万个 1s，总和时间可能会漫长到以年计算。显然我们不愿意在这种事情上消耗如此漫长的宝贵时间。

从上面的说明可以看出，和环境每次交互产生的数据都可能是非常珍贵的，如果仅用于一次学习就丢弃，显然是一种巨大的浪费，如果能把这些数据收集起来，反复地从这些数据中学习，从而减少和环境交互的次数，则将是很有帮助的。

因此，提出了数据池的概念，在有数据池的强化学习算法中，学习的一般过程如图 3-4 所示。



(a) 收集数据　　(b) 从数据中学习

图 3-4　有数据池的学习

对比图 3-3，从图 3-4 可以看出，强化学习的循环分成了两部分，分别是和环境交互产生数据，以及从积累的数据中学习两个步骤。

通过如图 3-4 的修改，数据的利用率提高了，理论上一条数据可以被反复地学习无限多次。看起来很美好，但是读者可以思考一下，这样的改造是否会存在隐患呢？

3.4.2　异策略和同策略

下面论述该过程中存在的问题，让我们分步来叙述，第 1 步如图 3-5 所示。

图 3-5　第 1 步有数据池的学习

在第 1 步，机器人和环境交互产生数据，并归入数据池，这看起来没有什么问题。接下来看第 2 步的情况，如图 3-6 所示。

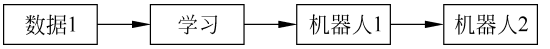


图 3-6　第 2 步有数据池的学习

在第 2 步，机器人从数据中学习，此时机器人的参数发生改变，也就是说，此时的机器人和第 1 步中的机器人已经不是同一个机器人了，它们的行为是不同的。接下来看第 3 步时的情况，如图 3-7 所示。

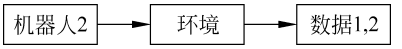


图 3-7　第 3 步有数据池的学习

在第 3 步，学习后的机器人继续和环境交互，产生的数据继续归入数据池，此时的数据池里有两个机器人和环境交互产生的数据，分别是第 1 步的机器人和第 2 步的机器人，这些数据混杂在一起，共同对后续机器人的学习产生影响，接下来看第 4 步的情况，如图 3-8 所示。

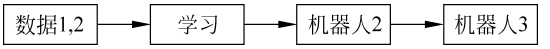


图 3-8　第 4 步有数据池的学习

在第 4 步，问题产生了，机器人要从数据池中积累的数据中学习，但是这些数据并不都来自上一步且由自己产生的，有些算法无法从不是自己产生的数据中学习，而有些算法则可以。

自此出现了强化学习算法的一个分水岭，只能从自己产生的数据中学习的算法被称为同策略算法，而可以从不是自己产生的数据中学习的算法则被称为异策略算法。

为了理解这两种算法的差异性，下面举一个感性的例子来说明，如图 3-9 所示。

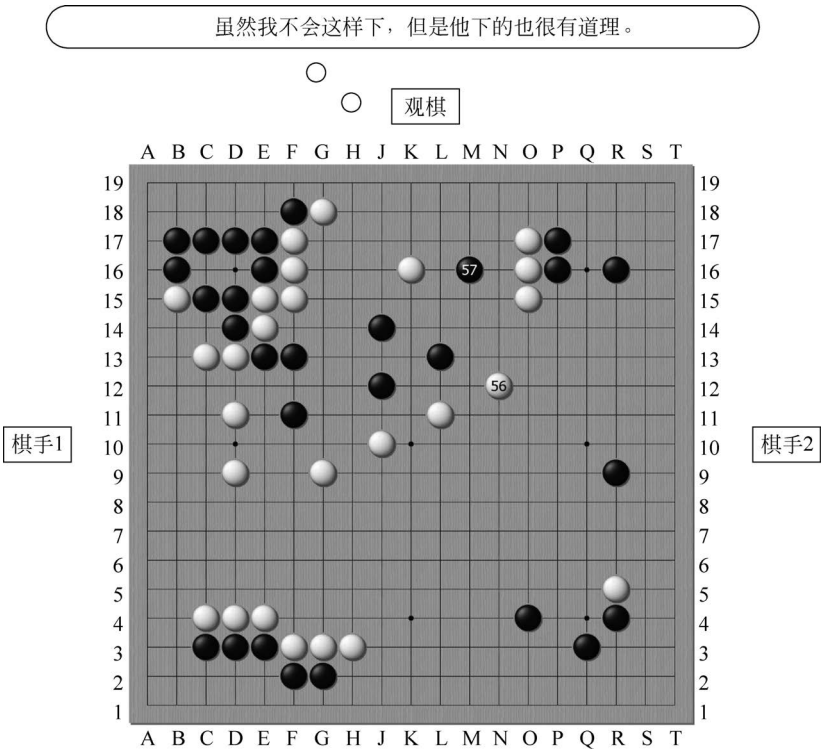


图 3-9　异策略算法示例

异策略算法像是从观棋中学习的棋手，通过大量阅读别人的走法来总结经验教训，这样它即使不去下棋，也能获得进步，这有点像是“熟读唐诗三百首，不会写诗也会吟”。通过不断地学习他人玩出来的数据，机器人自身也可以得到进步。

如果异策略是通过大量地从别人下棋的过程来学习下棋，则同策略就是真实地在下棋，通过不断地和对手切磋来精进自己的棋艺，它通过复盘自己的棋谱来总结经验教训，从而获得进步，如图 3-10 所示。

异策略算法和同策略算法都有各自的应用场景和各自擅长处理的问题，本章后续会分别介绍一个异策略算法和一个同策略算法。

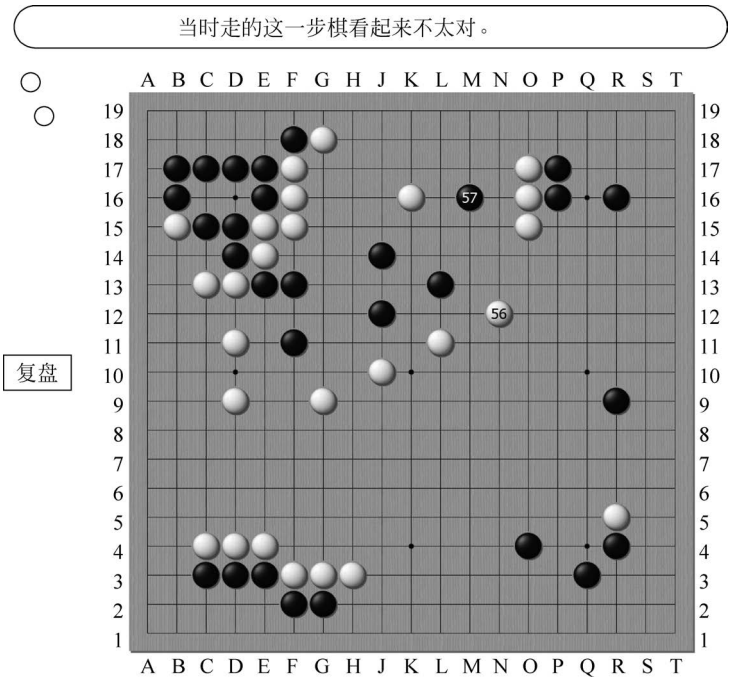


图 3-10 同策略算法示例

3.5 定义 play 函数和数据池

3.5.1 定义 play 函数

综上所述，在整个算法的训练过程中，可能需要玩很多局游戏，并且需要把玩游戏的过程记录成数据，为了便于处理这些工作，定义一个 play()函数，代码如下：

```
#第 3 章/定义 play 函数
from IPython import display
import random


#玩一局游戏并记录数据
def play(show=False):
    data = []
    reward_sum = 0

    state = env.reset()
    over = False
    while not over:
        action = Q[state].argmax()
        if random.random() < 0.1:
```

```
            action = env.action_space.sample()

        next_state, reward, over = env.step(action)

        data.append((state, action, reward, next_state, over))
        reward_sum += reward

        state = next_state

        if show:
            display.clear_output(wait=True)
            env.show()

    return data, reward_sum


play()[-1]
```

play()函数每调用一次就会玩一局游戏，并且把游戏的数据记录下来，play()函数在具体的实现过程中可以重点注意一下代码中加粗的部分，该部分是根据 state 计算一个 action 的过程。

要计算一个 action 需要有一个 state，在 play()函数的实现中，该计算过程就是简单地在 Q 表当中查询该 state 下分数最高的 action，最后取该动作执行即可。

值得注意的是，不是所有算法都会使用这种方法计算 action。此处演示的是 QLearning(Q 学习)算法采取的计算方法。

因为不希望机器人太过于死板，一般会给动作增加一定的随机性，例如有 10%的概率采取随机的动作，以上就是 play()函数的定义。该函数在后续的章节中会反复使用，读者务必熟悉该函数的实现过程。

也许读者会好奇调用一次 play()函数，它能收集到什么样的数据，下面是一局游戏的示例，如表 3-3 所示。

表 3-3　play()函数收集到的一局游戏的数据

state	action	reward	next state	over
0	0	−1	0	False
0	0	−1	0	False
0	0	−1	0	False
0	1	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False

续表

state	action	reward	next state	over
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
4	0	−1	4	False
…				

篇幅所限，此处不给出游戏环境的动画了，但从表 3-3 也能看出来，机器人几乎是在无意义地乱走。由于篇幅原因，表 3-3 的内容并没有给全，但最后一条数据的 over 字段一定是 True，只有该字段为 True，才能说明一局游戏游玩结束了。

3.5.2 定义数据池

有了 play()函数以后，可以很方便地采集玩游戏过程中得到的数据，为了便于管理这些数据，还需要定义一个数据池，数据池可以让机器人从过去的自己的那些数据中反复学习，提高数据的利用率，做到温故而知新。定义数据池的代码如下：

```
#第 3 章/定义数据池
class Pool:

    def __init__(self):
        self.pool = []

    def __len__(self):
        return len(self.pool)

    def __getitem__(self, i):
        return self.pool[i]

    #更新动作池
    def update(self):
        #每次更新不少于 N 条新数据
        old_len = len(self.pool)
        while len(pool) - old_len < 200:
            self.pool.extend(play()[0])

        #只保留最新的 N 条数据
        self.pool = self.pool[-1_0000:]

    #获取一批数据样本
```

```
    def sample(self):
        return random.choice(self.pool)


pool = Pool()
pool.update()

len(pool), pool[0]
```

数据池的功能是收集数据、采样数据。数据池收集数据的方式就是调用前面定义的play()函数。

数据池的采样功能是从数据池中随机地抽取一条数据。由于本章的任务比较简单，所以此处不涉及批采样的功能，每次仅采样一条数据即可，在后续章节中会要求数据池每次采样都采样 N 条数据。

下面是一条本章定义的数据池采样到的数据样例，如表 3-4 所示。

表 3-4　数据池采样到的一条数据

state	action	reward	next state	over
4	0	−1	4	False

从表 3-4 可以看出，采样的结果是一步动作的结果，数据中包括 5 个字段，分别是 state、action、reward、next state、over。后续将会在这 5 个字段的基础上研发强化学习算法。

3.6　使用时序差分方法更新 Q 表

回顾上面的准备工作，到此处为止，已经准备好了以下工具组件：

(1) 冰湖游戏环境。

(2) Q 表，一共有 $4\times4=16$ 种状态，每种状态可以做出上、下、左、右 4 个动作，所以 Q 表是一个 16×4 的矩阵，初始化为全 0。

(3) play()函数，每调用一次会根据 Q 表中记录的策略玩一局游戏，并收集一局游戏的数据。

(4) 数据池，它负责调用 play()函数，并收集 play()函数返回的数据，数据池还具有数据采样的功能，每次采样获得 state、action、reward、next state、over 各一条。

回顾以上这些工作组件，可以看出重点在于 Q 表记录的数据，它决定了机器人能不能在冰湖这个游戏环境中表现得好，所以需要对 Q 表的数据进行优化，QLearning 算法正是完成这项工作的一个算法。

要理解 QLearning 算法的原理，首先回顾一下 Q 函数的定义，如式(3-1)所示。

$$Q(s_t,a_t)=E\left[R_t+\text{gamma}\cdot R_{t+1}+\text{gamma}^2\cdot R_{t+2}+\cdots+\text{gamma}^{n-t}\cdot R_n\right] \quad (3\text{-}1)$$

从式(3-1)可以看出，Q 函数计算的是在 s_t 状态下，执行 a_t 动作，后续可以得到折扣回

报的和的期望，关于这一点读者如果不清楚，则可以回顾第 2 章的内容。

如果对式(3-1)进行变形，则可以得到式(3-2)

$$Q(s_t,a_t)=R_t+E[\text{gamma}\cdot R_{t+1}+\text{gamma}^2\cdot R_{t+2}+\cdots+\text{gamma}^{n-t}\cdot R_n] \tag{3-2}$$

式(3-2)只是把式(3-1)中的 R_t 提到了期望的外面，因为在实际优化过程中 R_t 是个可以获得的实际数据，不需要估计，关于这一点可以回顾数据池采样的结果，其中包括了 reward 数据，这个数据是可以实际获取的，不需要估计，所以不需要包括在期望函数中。

下面对式(3-2)进一步地进行化简，得到式(3-3)

$$Q(s_t,a_t)=R_t+\text{gamma}\cdot Q(s_{t+1},a_{t+1}) \tag{3-3}$$

根据 Q 函数的定义，式(3-3)显然是成立的。此时可以发现一个有趣的现象，从数学定义上，式(3-3)等号左右两边是相等关系，但注意到 Q 函数本身是一个带期望的估计函数，它的估计是有误差的，所以式(3-3)等号两边的估计误差可以不一致，此时等号的两边显然就不相等了，这时，误差产生了。

根据第 2 章讲解的时序差分理论，读者应该已经注意到应该以谁为准修正 Q 函数的误差了。

注意到式(3-3)等号两边都是 Q 函数估计的结果，但是等号的左边完全是估计值，没有一丁点儿的事实成分，而等号的右边包括了一步的事实数据 R_t，所以很显然，等号右边的估计值更可靠。

根据时序差分理论，应该以等号右边为准修正 Q 函数的估计值。使用时序差分方法修正 Q 函数的方法如式(3-4)所示。

$$\begin{cases}\text{value}=Q(s_t,a_t)\\ \text{target}=R_t+\text{gamma}\cdot Q(s_{t+1},a_{t+1})\\ \text{td}=\text{target}-\text{value}\\ Q(s_t,a_t)=Q(s_t,a_t)\cdot\text{alpha}\cdot\text{td}\end{cases} \tag{3-4}$$

式(3-4)中的 alpha 类似于学习率，以上方法就被称为时序差分方法。

3.7 QLearning 算法

做完以上准备工作后，终于可以进入本书的第 1 个实战的强化学习算法了，即 QLearning(Q 学习)算法，QLearning 算法可以说是强化学习算法当中最简单、最基础的算法，也是最好理解的算法，所以本书选择以该算法作为强化学习体系的切入例子，向读者介绍强化学习算法的一般过程。

QLearning 算法是一种异策略算法，也就是说 QLearning 算法通过围观他人的棋局来精进自己的棋艺，所以 QLearning 算法有一个很明显的优点，也就是对数据的利用率高，可以从非自身产生的数据中学习，可以使用数据池提高数据的利用率。

回顾式(3-3)可以发现一处矛盾，在等号的右边需要变量 a_{t+1}，这对 QLearning 算法是困难的，因为 QLearning 算法是异策略算法，它获得的数据可能不是自身产生的，所以在

s_{t+1} 这种状态下，它不一定会做出 a_{t+1} 这个动作，所以计算 a_{t+1} 的 Q 值对它来讲是缺乏意义的，甚至是一种误导。

QLearning 算法是如何解决上述矛盾的呢？事实上 QLearning 算法在这里采用了一种笔者认为简单的方法，即求所有动作中 Q 值最大的。如式(3-5)所示。

$$Q(s_t,a_t)=R_t+\text{gamma}\cdot\max\rightarrow Q(s_{t+1},*) \tag{3-5}$$

从式(3-5)可以看出，在 QLearning 算法中，在 target 的部分中，它直接求所有动作中分值最高的值作为 Q 值，这样就成功地消去了变量 a_{t+1}，但也导致了后续的过高估计的问题，这一点在后续的章节中再来展开。总之通过上述改造，QLearning 算法成功地把自己变成了一个异策略算法，可以享受数据池带来的好处。

了解了以上理论以后，现在可以着手实现 QLearning 算法了，代码如下：

```
#第 3 章/训练
def train():
    #共更新 N 轮数据
    for epoch in range(1000):
        pool.update()

        #每次更新数据后，训练 N 次
        for i in range(200):

            #随机抽一条数据
            state, action, reward, next_state, over = pool.sample()

            #Q 矩阵当前估计的 state 下 action 的价值
            value = Q[state, action]

            #实际玩了之后得到的 reward+下一种状态的价值*0.9
            target = reward + Q[next_state].max() *0.9

            #value 和 target 应该是相等的，说明 Q 矩阵的评估准确
            #如果有误差，则应该以 target 为准更新 Q 表，修正它的偏差
            #这就是 TD 误差，指评估值之间的偏差，以实际成分高的评估为准进行修正
            update = (target - value) *0.1

            #更新 Q 表
            Q[state, action] += update

        if epoch % 100 == 0:
            print(epoch, len(pool), play()[-1])


train()
```

训练过程概述如下：

(1) 整个在训练过程中共有 1000 个 epoch。

(2) 在每个 epoch 当中让数据池更新一批数据,所以在整个训练过程中一共会更新 1000 次数据,每次会更新 200 条左右的数据,每次更新过数据之后训练 200 次。

(3) 每次训练的过程会从数据池当中随机抽取一条数据,以该数据计算 value 和 target。

(4) value 比较简单,直接地使用 Q 表查询即可,即式(3-5)的左边部分。

(5) target 的计算过程即式(3-5)的右边部分。

(6) 注意 target 计算过程中的 max,这是 QLearning 算法的重点,在 QLearning 算法中 next state 的价值直接取 next state 所有 4 个动作的最高价值,在 SARSA 算法中不会这样做。

(7) 理想情况 target 和 value 应该相等,如果两者有误差,则根据时序差分的思想,应该以 target 来修正 value,因为 target 当中有一步的事实数据,而 value 则完全是估计值,所以认为 target 更加可靠。

(8) 让 value 和 target 做差,乘以 learning rate 得到要修正的误差,更新到 Q 表当中即可。

以上就是 Q 函数的训练过程,在训练过程中的输出如下:

```
0 513 -199
100 10000 -4.0
200 10000 -103
300 10000 -103
400 10000 -4.0
500 10000 -6.0
600 10000 -4.0
700 10000 -101
800 10000 -6.0
900 10000 -4.0
```

在输出的内容当中,重点关注最后一列数值,这个数值是测试的结果,可以看到最开始的时候测试得了-199 分,在之后的测试过程中,几乎每局得-4、-5 分,可见训练的过程是有效的。

训练完成以后,可以测试一局游戏并打印动画,查看机器人玩游戏的水平,代码如下:

```
#第 2 章/测试训练好的模型,并打印动画
play(True)[-1]
```

运行的结果如图 3-11 所示。

训练完成后的机器人玩游戏的水平有了显著提升,可见训练的过程是有效并正确的。

以上就是 QLearning 算法实现和训练的过程,可见该算法还是比较简单的,训练的速度也很快,虽然这个算法的内容比较简单,但读者应该好好地掌握该算法的内容,由于这是本书介绍的第 1 个算法,后续很多算法的代码结构和 QLearning 是一样的,所以熟悉 QLearning 算法的代码结构很重要。

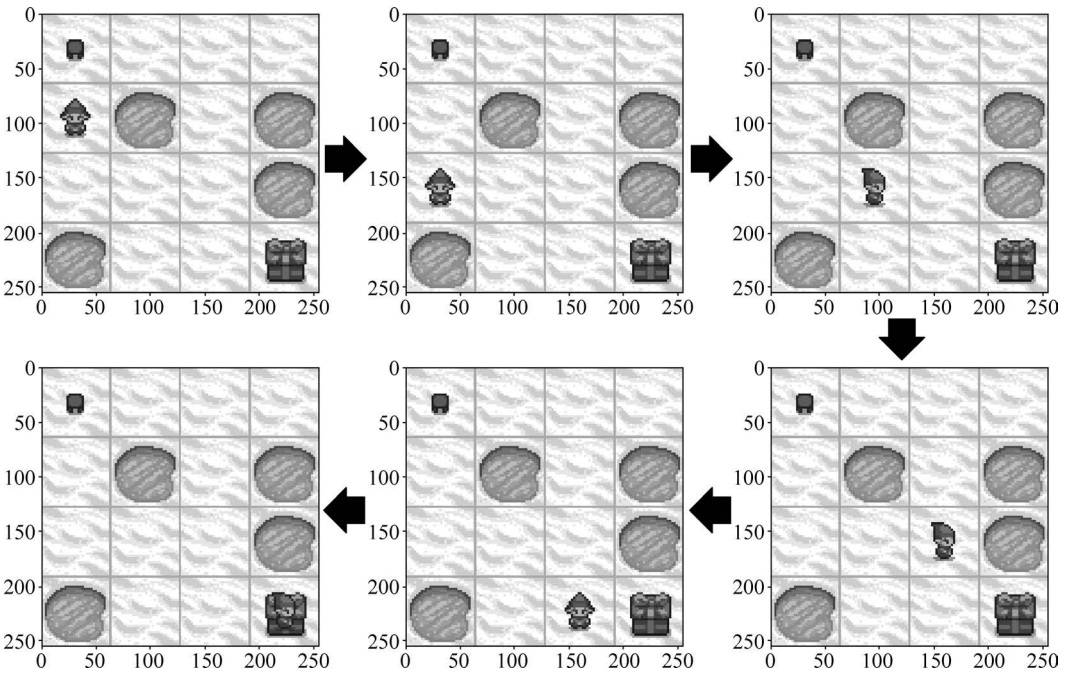


图 3-11　测试结果

3.8　SARSA 算法

上面介绍了 QLearning 算法，下面介绍经常和 QLearning 放在一起的 SARSA（State Action Reward next State next Action）算法，这两个算法如此相似，以至于从 QLearning 算法修改到 SARSA 算法只需改动两行代码。

虽然 SARSA 和 QLearning 看起来非常相似，但其实它们是完全不同的算法，稍后展开叙述，这里先跳过复杂的理论部分，直接给出 SARSA 算法的代码实现，代码如下：

```
#第 3 章/训练
def train():
    #共更新 N 轮数据
    for epoch in range(2000):
        pool.update()

        #每次更新数据后训练 N 次
        for i in range(200):

            #随机抽一条数据
            state, action, reward, next_state, over = pool.sample()

            #Q 矩阵当前估计的 state 下 action 的价值
```

```
            value = Q[state, action]

            #求下一个动作,这是和 Q 学习唯一的区别点
            next_action = Q[next_state].argmax()

            #实际玩了之后得到的 reward+下一种状态的价值*0.9
            target = reward + Q[next_state, next_action] *0.9

            #value 和 target 应该是相等的,说明 Q 矩阵的评估准确
            #如果有误差,则应该以 target 为准更新 Q 表,修正它的偏差
            #这就是 TD 误差,指评估值之间的偏差,以实际成分高的评估为准进行修正
            update = (target - value) *0.02

            #更新 Q 表
            Q[state, action] += update

        if epoch % 100 == 0:
            print(epoch, len(pool), play()[-1])


train()
```

上面的代码需要注意加粗的部分,这两行是 SARSA 算法和 QLearning 算法代码实现上仅有的区别。

因为在本章所使用的游戏环境比较简单,所以 Q 表也比较简单,导致这里的计算感觉有点多余,但如果是在一个比较复杂的游戏环境下,则可能会导致 Q 表非常复杂,求出的下一个时刻的动作可能就不是 max 了。

在训练过程中的输出如下:

```
0 554 -116
100 10000 -4.0
200 10000 -6.0
300 10000 -6.0
400 10000 -5.0
500 10000 -5.0
600 10000 -105
700 10000 -4.0
800 10000 -4.0
900 10000 -5.0
1000 10000 -6.0
1100 10000 -5.0
1200 10000 -4.0
1300 10000 -4.0
1400 10000 -7.0
1500 10000 -5.0
1600 10000 -4.0
1700 10000 -5.0
```

```
1800 10000 -4.0
1900 10000 -4.0
```

输出结果当中重点关注最后一列数值即可，该数值表明在每次测试当中当前机器人玩一局游戏取得的成绩。可以看到在训练之前机器人玩了一局游戏并得到了－116 分的成绩，在经过训练后，几乎每局得到的分数都在－5 左右，可见训练的过程是有效的。

从上面的代码能看出来，SARSA 算法和 QLearning 算法的区别只在于 target 的计算方式不同，SARSA 的 target 的计算方法如式(3-6)所示。

$$\text{target}=Q(s_{t+1},a_{t+1})\cdot\text{gamma}+r_t \tag{3-6}$$

从式(3-6)可以看出在 SARSA 算法中，计算 target 需要 s_{t+1} 和 a_{t+1}，这符合 Q 函数的定义。

对比之下 QLearning 的 target 的计算方式就比较简单了，直接取下一个时刻的状态下所有动作的最高的价值，如式(3-7)所示。

$$\text{target}=\max\rightarrow Q(s_{t+1},*)\cdot\text{gamma}+r_t \tag{3-7}$$

target 的不同计算方式是 SARSA 算法和 QLearning 算法主要的区别点。

细心的读者读到这里可能已经发现了关键，很显然 SARSA 和 QLearning 算法有一处关键性的不同，导致它们成为完全不同的算法，式(3-6)表明 SARSA 算法计算 target 需要使用变量 a_{t+1}，而该变量只能来自 SARSA 算法本身，所以 SARSA 算法是一个同策略的算法，它不能使用其他人产生的数据进行学习，进而导致它不能使用数据池，因为过去的自己也不是自己，也只能被视为他人。

因此，QLearning 算法是异策略的，而 SARSA 算法是同策略的。它们是完全不同的算法。

虽然 SARSA 算法是同策略的，理论上它不能使用数据池，但出于简单起见上面的实现还是给它使用了数据池。由于本章使用的游戏环境比较简单，所以还是能通过学习得到比较好的结果，但这违反 SARSA 算法的使用原则，因为 SARSA 算法是一个同策略算法，所以理论上不能使用数据池。

3.9　实现无数据池的 SARSA 算法

实现无池化版本的 SARSA 算法，代码如下：

```
#第 3 章/训练
def train():
    #共更新 N 轮数据
    for epoch in range(2000):

        #玩一局游戏并得到数据
```

```
        for (state, action, reward, next_state, over) in play()[0]:

            #Q 矩阵当前估计的 state 下 action 的价值
            value = Q[state, action]

            #实际玩了之后得到的 reward+(next_state,next_action)的价值*0.9
            target = reward + Q[next_state, Q[next_state].argmax()] *0.9

            #value 和 target 应该是相等的,说明 Q 矩阵的评估准确
            #如果有误差,则应该以 target 为准更新 Q 表,修正它的偏差
            #这就是 TD 误差,指评估值之间的偏差,以实际成分高的评估为准进行修正
            update = (target - value) *0.02

            #更新 Q 表
            Q[state, action] += update

        if epoch % 100 == 0:
            print(epoch, play()[-1])


train()
```

从上面的代码中可以看到,每次训练的时候都是去现玩一局游戏,然后针对这局游戏当中的每步的数据进行优化,所以这是一个无池化的实现,运行结果如下:

```
0 -101
100 -128
200 -7.0
300 -4.0
400 -6.0
500 -4.0
600 -4.0
700 -4.0
800 -4.0
900 -4.0
1000 -6.0
1100 -4.0
1200 -4.0
1300 -4.0
1400 -4.0
1500 -4.0
1600 -4.0
1700 -6.0
1800 -4.0
1900 -4.0
```

可以看到也能够得到很好的训练结果。

3.10　小结

本章介绍了游戏环境包 Gym，介绍了 Gym 包的基本用法，以冰湖游戏环境为例进行了讲解。

本章介绍了两个基础的基于表格的强化学习算法，分别是异策略的 QLearning 算法和同策略的 SARSA 算法。

本章讲解了在强化学习算法中应用数据池的必要性，并实现了数据池工具类，该工具类在后续章节中将反复使用，读者务必熟悉该工具类的用法。

本章讲解了时序差分方法在强化学习算法中的应用，并讲解了同策略和异策略算法的差异。

本章的内容在本书结构中非常重要，后续所有章节将根据本章的代码结构进行扩展，读者务必熟悉本章代码的结构。

在强化学习的过程中代码调试是不必避免的，本书的代码量不大，读者务必熟悉代码的运算过程。