



Easy RL

强化学习中文教程

<https://github.com/datawhalechina/easy-rl>

Qi Wang Yiyuan Yang Ji Jiang 编著

版本：1.0.0

2021 年 5 月 16 日

前言

李宏毅老师的《深度强化学习》是强化学习领域经典的中文视频之一。李老师幽默风趣的上课风格让晦涩难懂的强化学习理论变得轻松易懂，他会通过很多有趣的例子来讲解强化学习理论。比如老师经常会用玩 Atari 游戏的例子来讲解强化学习算法。此外，为了教程的完整性，我们整理了周博磊老师的《强化学习纲要》、李科尧老师的《百度强化学习》以及多个强化学习的经典资料作为补充。对于想入门强化学习又想看中文讲解的人来说绝对是非常推荐的。

使用说明

- 第 4 章到第 11 章为李宏毅《深度强化学习》的部分；
- 第 1 章和第 2 章根据《强化学习纲要》整理而来；
- 第 3 章和第 12 章根据《百度强化学习》整理而来。

在线阅读地址: <https://datawhalechina.github.io/easy-rl/> (内容实时更新)

最新版 PDF 获取地址: <https://github.com/datawhalechina/easy-rl/releases>

编委会

编委: Qi Wang, Yiyuan Yang, Ji Jiang

致谢

特别感谢 Sm1les、LSGOMYP 对本项目的帮助与支持。

扫描下方二维码，然后回复关键词“**强化学习**”，即可加入“Easy-RL 读者交流群”



Datawhale
一个专注于 AI 领域的开源组织

版权声明

本作品采用知识共享署名-非商业性使用-相同方式共享 4.0 国际许可协议进行许可。

目录

第 1 章 Reinforcement Learning	1
1.1 Reinforcement Learning	1
1.2 Introduction to Sequential Decision Making	7
1.2.1 Agent and Environment	7
1.2.2 Reward	7
1.2.3 Sequential Decision Making	7
1.3 Action Spaces	9
1.4 Major Components of an RL Agent	9
1.4.1 Policy	9
1.4.2 Value Function	9
1.4.3 Model	10
1.4.4 Types of RL Agents	11
1.5 Learning and Planning	13
1.6 Exploration and Exploitation	14
1.6.1 K-armed Bandit	15
1.7 Experiment with Reinforcement Learning	16
1.7.1 Gym	16
1.7.2 MountainCar-v0 Example	19
1.8 Keywords	21
1.9 Questions	22
1.10 Something About Interview	24
第 2 章 MDP	25
2.1 Markov Process(MP)	25
2.1.1 Markov Property	25
2.1.2 Markov Process/Markov Chain	25
2.1.3 Example of MP	26
2.2 Markov Reward Process(MRP)	27
2.2.1 Example of MRP	27
2.2.2 Return and Value function	27
2.2.3 Why Discount Factor	28
2.2.4 Bellman Equation	29
2.2.5 Iterative Algorithm for Computing Value of a MRP	31
2.3 Markov Decision Process(MDP)	33
2.3.1 MDP	33
2.3.2 Policy in MDP	33
2.3.3 Comparison of MP/MRP and MDP	33
2.3.4 Value function for MDP	34
2.3.5 Bellman Expectation Equation	34
2.3.6 Backup Diagram	35
2.3.7 Policy Evaluation(Prediction)	37
2.3.8 Prediction and Control	38
2.3.9 Dynamic Programming	40

2.3.10 Policy Evaluation on MDP	40
2.3.11 MDP Control	43
2.3.12 Policy Iteration	44
2.3.13 Value Iteration	46
2.3.14 Difference between Policy Iteration and Value Iteration	48
2.3.15 Summary for Prediction and Control in MDP	50
2.4 Keywords	51
2.5 Questions	52
2.6 Something About Interview	53
第 3 章 Tabular Methods	55
3.1 MDP	55
3.1.1 Model-based	55
3.1.2 Model-free	56
3.1.3 Model-based vs. Model-free	57
3.2 Q-table	57
3.3 Model-free Prediction	60
3.3.1 Monte-Carlo Policy Evaluation	60
3.3.2 Temporal Difference	63
3.3.3 Bootstrapping and Sampling for DP, MC and TD	66
3.4 Model-free Control	67
3.4.1 Sarsa: On-policy TD Control	71
3.4.2 Q-learning: Off-policy TD Control	73
3.5 On-policy vs. Off-policy	76
3.6 Summary	76
3.7 Keywords	76
3.8 Questions	77
3.9 Something About Interview	78
3.10 Solve CliffWalking with Q-learning	79
3.10.1 CliffWalking-v0 环境简介	80
3.10.2 RL 基本训练接口	80
3.10.3 任务要求	81
3.10.4 主要代码清单	81
3.10.5 备注	82
第 4 章 Policy Gradient	83
4.1 Policy Gradient	83
4.2 Tips	90
4.2.1 Tip 1: Add a Baseline	90
4.2.2 Tip 2: Assign Suitable Credit	92
4.3 REINFORCE: Monte Carlo Policy Gradient	94
4.4 Keywords	98
4.5 Questions	98
4.6 Something About Interview	100

第 5 章 PPO	102
5.1 From On-policy to Off-policy	102
5.1.1 Importance Sampling	102
5.2 PPO	106
5.2.1 PPO-Penalty	107
5.2.2 PPO-Clip	108
5.3 Keywords	111
5.4 Questions	111
5.5 Something About Interview	111
第 6 章 DQN	113
6.1 State Value Function	113
6.1.1 State Value Function Estimation	114
6.2 State-action Value Function(Q-function)	116
6.3 Target Network	120
6.3.1 Intuition	121
6.4 Exploration	122
6.5 Experience Replay	123
6.6 DQN	125
6.7 Keywords	126
6.8 Questions	126
6.9 Something About Interview	129
第 7 章 Tips of DQN	130
7.1 Double DQN	130
7.2 Dueling DQN	131
7.3 Prioritized Experience Replay	134
7.4 Balance between MC and TD	134
7.5 Noisy Net	135
7.6 Distributional Q-function	136
7.7 Rainbow	138
7.8 Keywords	139
7.9 Questions	140
7.10 Something About Interview	141
7.11 Solve Cartpole with DQN	141
7.11.1 CartPole-v0	141
7.11.2 强化学习基本接口	141
7.11.3 CartPole-v0	142
7.11.4 代码清单	143
第 8 章 Q-learning for Continuous Actions	145
8.1 Solution 1 & Solution 2	145
8.2 Solution 3: Design a network	146
8.3 Solution 4: Don't use Q-learning	147
8.4 Questions	147

第 9 章 Actor-Critic	149
9.1 Actor-Critic	149
9.1.1 Review: Policy Gradient	149
9.1.2 Review: Q-learning	150
9.1.3 Actor-Critic	151
9.1.4 Advantage Actor-Critic	151
9.2 A3C	154
9.3 Pathwise Derivative Policy Gradient	155
9.4 Connection with GAN	158
9.5 Keywords	159
9.6 Questions	159
9.7 Something About Interview	160
第 10 章 Sparse Reward	162
10.0.1 Reward Shaping	162
10.0.2 Curiosity	163
10.1 Curriculum Learning	166
10.1.1 Reverse Curriculum Generation	167
10.2 Hierarchical RL	168
10.3 Keywords	170
10.4 Questions	170
第 11 章 Imitation Learning	172
11.1 Behavior Cloning	172
11.2 Inverse RL	175
11.3 Third Person Imitation Learning	179
11.4 Recap: Sentence Generation & Chat-bot	180
11.5 Keywords	181
11.6 Questions	181
第 12 章 DDPG	183
12.1 Discrete Action vs. Continuous Action	183
12.2 DDPG(Deep Deterministic Policy Gradient)	184
12.2.1 Exploration vs. Exploitation	187
12.3 Twin Delayed DDPG(TD3)	187
12.3.1 Exploration vs. Exploitation	189
12.4 Keywords	189
12.5 Questions	189
12.6 Something About Interview	189
12.7 Solve Pendulum with DDPG	190
12.7.1 Pendulum-v0	190
12.7.2 强化学习基本接口	191
12.7.3 任务要求	191
12.7.4 注意	191
12.7.5 代码清单	194

第 13 章 AlphaStar 论文解读	195
13.1 AlphaStar 以及背景简介	195
13.2 AlphaStar 的模型输入输出是什么? ——环境设计	195
13.2.1 状态 (网络的输入)	195
13.2.2 动作 (网络的输出)	196
13.3 AlphaStar 的计算模型是什么呢? ——网络结构	196
13.3.1 输入部分	197
13.3.2 中间过程	197
13.3.3 输出部分	197
13.4 庞大的 AlphaStar 如何训练呢? ——学习算法	198
13.4.1 监督学习	198
13.4.2 强化学习	198
13.4.3 模仿学习	200
13.4.4 多智能体学习/自学习	200
13.5 AlphaStar 实验结果如何呢? ——实验结果	200
13.5.1 宏观结果	200
13.5.2 其他实验 (消融实验)	201
13.6 关于 AlphaStar 的总结	201
13.6.1 总结	201

第 1 章 Reinforcement Learning

1.1 Reinforcement Learning

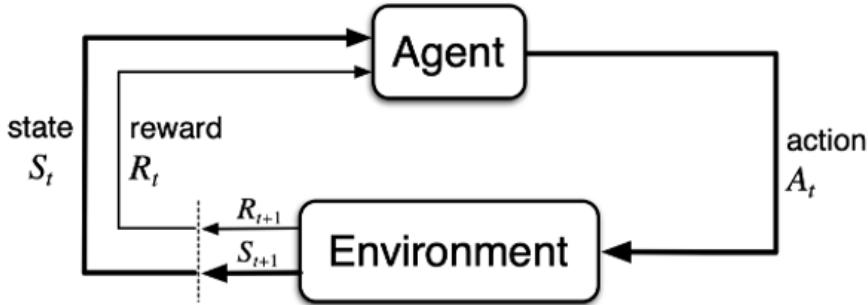


图 1.1 强化学习示意图

强化学习讨论的问题是一个智能体 (agent) 怎么在一个复杂不确定的环境 (environment) 里面去极大化它能获得的奖励。如图 1.1 所示，示意图由两部分组成：agent 和 environment。在强化学习过程中，agent 跟 environment 一直在交互。Agent 在环境里面获取到状态，agent 会利用这个状态输出一个动作 (action)，一个决策。然后这个决策会放到环境之中去，环境会根据 agent 采取的决策，输出下一个状态以及当前的这个决策得到的奖励。Agent 的目的就是为了尽可能多地从环境中获取奖励。

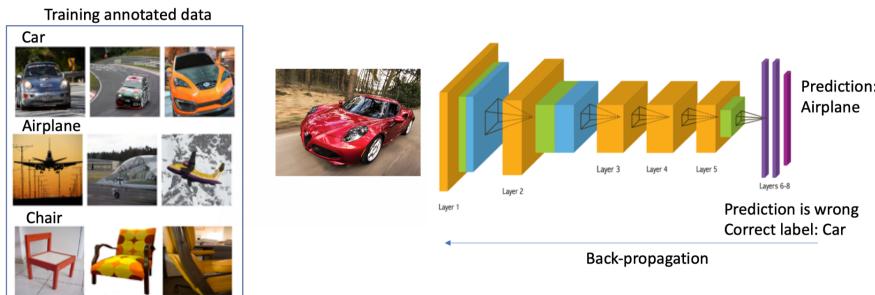


图 1.2 监督学习

我们可以把强化学习跟监督学习做一个对比。

举个图片分类的例子，如图 1.2 所示，[监督学习 \(supervised learning\)](#)就是说我们有一大堆标注的数据，比如车、飞机、凳子这些标注的图片，这些图片都要满足独立同分布 (i.i.d.)，就是它们之间是没有关联的。

然后我们训练一个分类器，比如说右边这个神经网络。为了分辨出这个图片是车辆还是飞机，训练过程中，我们把真实的标签给了这个网络。当这个网络做出一个错误的预测，比如现在输入了汽车的图片，它预测出来是飞机。我们就会直接告诉它，你这个预测是错误的，正确的标签应该是车。然后我们把这个错误写成一个损失函数 (loss function)，通过反向传播 (Backpropagation) 来训练这个网络。

所以在监督学习过程中，有两个假设：

- 输入的数据（标注的数据）都是没有关联的，尽可能没有关联。因为如果有关联的话，这个网络是不好学习的。
- 我们告诉学习器 (learner) 正确的标签是什么，这样它可以通过正确的标签来修正自己的预测。

通常假设样本空间中全体样服从一个未知分布，我们获得的每个样本都是独立地从这个分布上采样获得的，即独立同分布 (independent and identically distributed，简称 i.i.d.)。



图 1.3 Atari 游戏：Breakout

在强化学习里面，这两点其实都不满足。举一个 Atari Breakout 游戏的例子，如图 1.3 所示，这是一个打砖块的游戏，控制木板左右移动把球反弹到上面来消除砖块。

在游戏过程中，大家可以发现这个 agent 得到的观测不是个独立同分布的分布，上一帧下一帧其实有非常强的连续性。这就是说，得到的数据是相关的时间序列数据，不满足独立同分布。

另外一点，在玩游戏的过程中，你并没有立刻获得反馈，没有告诉你哪个动作是正确动作。比如你现在把这个木板往右移，那么只会使得这个球往上或者往左上去一点，你并不会得到立刻的反馈。所以强化学习这么困难的原因是没有得到很好的反馈，然后你依然希望 agent 在这个环境里面学习。

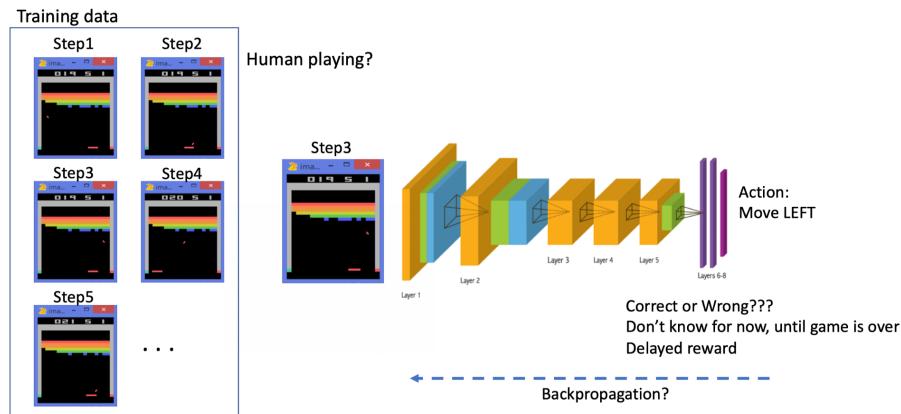


图 1.4 强化学习：玩 Breakout

强化学习的训练数据就是这样一个玩游戏的过程。你从第一步开始，采取一个决策，比如说你把这个往右移，接到这个球了。第二步你又做出决策，得到的训练数据是一个玩游戏的序列。

比如现在是在第三步，你把这个序列放进去，你希望这个网络可以输出一个决策，在当前的这个状态应该输出往右移或者往左移。这里有个问题：我们没有标签来说明你现在这个动作是正确还是错误，必须等到游戏结束才可能说明，这个游戏可能十秒过后才结束。现在这个动作到底对最后游戏结束能赢是否有帮助，其实是不清楚的。这里就面临延迟奖励 (Delayed Reward)，所以就使得训练这个网络非常困难。

我们对比下强化学习和监督学习。

- 强化学习输入的是序列数据，而不是像监督学习里面这些样本都是独立的。
- 学习器并没有被告诉你每一步正确的行为应该是什么。学习器需要自己去发现哪些行为可以得到最多的奖励，只能通过不停地尝试来发现最有利的动作。
- Agent 获得自己能力的过程中，其实是通过不断地试错探索 (trial-and-error exploration)。
 - 探索 (exploration) 和利用 (exploitation) 是强化学习里面非常核心的一个问题。

- 探索：你会去尝试一些新的行为，这些新的行为有可能会使你得到更高的奖励，也有可能使你一无所有。
- 利用：采取你已知的可以获得最大奖励的行为，你就重复执行这个动作就可以了，因为你已经知道可以获得一定的奖励。
- 因此，我们需要在探索和利用之间取得一个权衡，这也是在监督学习里面没有的情况。

在强化学习过程中，没有非常强的监督者 (supervisor)，只有一个奖励信号 (reward signal)，并且这个奖励信号是延迟的，就是环境会在很久以后告诉你之前你采取的行为到底是不是有效的。Agent 在这个强化学习里面学习的话就非常困难，因为你没有得到即时反馈。当你采取一个行为过后，如果是监督学习，你就立刻可以获得一个指引，就说你现在做出了一个错误的决定，那么正确的决定应该是谁。而在强化学习里面，环境可能会告诉你这个行为是错误的，但是它并没有告诉你正确的行为是什么。而且更困难的是，它可能是在一两分钟过后告诉你错误，它再告诉你之前的行为到底行不行。所以这也是强化学习和监督学习不同的地方。

通过跟监督学习比较，我们可以总结出强化学习的一些特征。

- 强化学习有这个试错探索 (trial-and-error exploration)，它需要通过探索环境来获取对环境的理解。
- 强化学习 agent 会从环境里面获得延迟的奖励。
- 在强化学习的训练过程中，时间非常重要。因为你得到的数据都是有时间关联的 (sequential data)，而不是独立同分布的。在机器学习中，如果观测数据有非常强的关联，其实会使得这个训练非常不稳定。这也是为什么在监督学习中，我们希望数据尽量是独立同分布，这样就可以消除数据之间的相关性。
- Agent 的行为会影响它随后得到的数据，这一点是非常重要的。在我们训练 agent 的过程中，很多时候我们也是通过正在学习的这个 agent 去跟环境交互来得到数据。所以如果在训练过程中，这个 agent 的模型很快死掉了，那会使得我们采集到的数据是非常糟糕的，这样整个训练过程就失败了。所以在强化学习里面一个非常重要的问题就是怎么让这个 agent 的行为一直稳定地提升。

为什么我们关注强化学习，其中非常重要的一点就是强化学习得到的模型可以有超人类的表现。

监督学习获取的这些监督数据，其实是让人来标注的。比如说 ImageNet 的图片都是人类标注的。那么我们就可以确定这个算法的上限 (upper bound) 就是人类的表现，人类的这个标注结果决定了它永远不可能超越人类。

但是对于强化学习，它在环境里面自己探索，有非常大的潜力，它可以超越人的能力的这个表现，比如谷歌 DeepMind 的 AlphaGo 这样一个强化学习的算法可以把人类最强的棋手都打败。

这里给大家举一些在现实生活中强化学习的例子。

- 在自然界中，羚羊其实也是在做一个强化学习，它刚刚出生的时候，可能都不知道怎么站立，然后它通过试错的一个尝试，三十分钟过后，它就可以跑到每小时 36 公里，很快地适应了这个环境。
- 你也可以把股票交易看成一个强化学习的问题，就怎么去买卖来使你的收益极大化。
- 玩雅达利游戏或者一些电脑游戏，也是一个强化学习的过程。



图 1.5 Pong 游戏

图 1.5 是强化学习的一个经典例子，就是雅达利的一个叫 Pong 的游戏。这个游戏就是把这个球拍到

左边，然后左边这个选手需要把这个球拍到右边。训练好的一个强化学习 agent 和正常的选手有区别，强化学习的 agent 会一直在做这种无意义的一些振动，而正常的选手不会出现这样的行为。

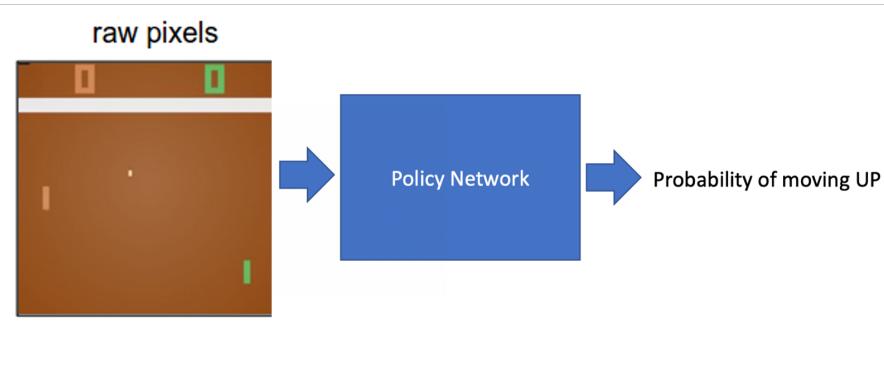


图 1.6 强化学习玩 pong

在这个 pong 的游戏里面，决策其实就是两个动作：往上或者往下。如果强化学习是通过学习一个 policy network 来分类的话，其实就是要输入当前帧的图片，policy network 就会输出所有决策的可能性。

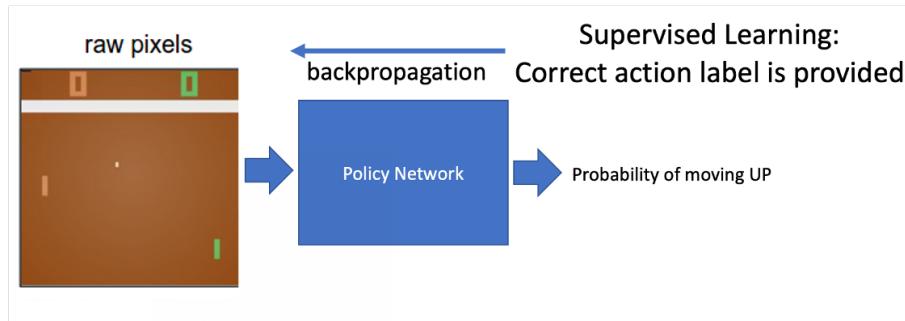


图 1.7 监督学习玩 pong

对于监督学习，我们可以直接告诉 agent 正确的标签是什么。但在这种游戏情况下面，我们并不知道它的正确的标签是什么。

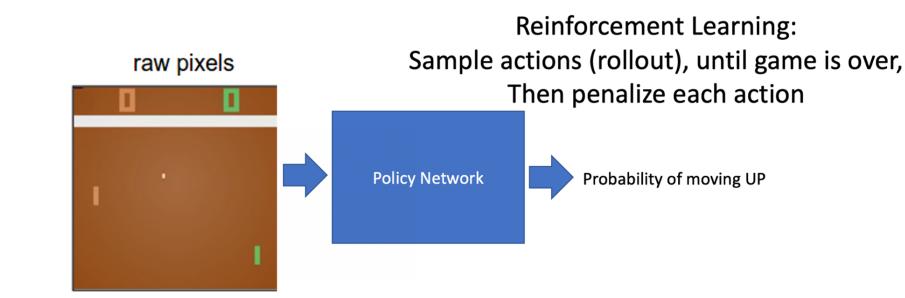


图 1.8 强化学习玩 pong 的具体过程

在强化学习里面，我们是通过让它尝试去玩这个游戏，然后直到游戏结束过后，再去说你前面的一系列动作到底是正确还是错误。

上图的过程是 rollout 的一个过程。Rollout 的意思是从当前帧去生成很多局的游戏。

当前的 agent 去跟环境交互，你就会得到一堆观测。你可以把每一个观测看成一个轨迹 (trajectory)。轨迹就是当前帧以及它采取的策略，即状态和动作的一个序列：

$$\tau = (s_0, a_0, s_1, a_1, \dots)$$

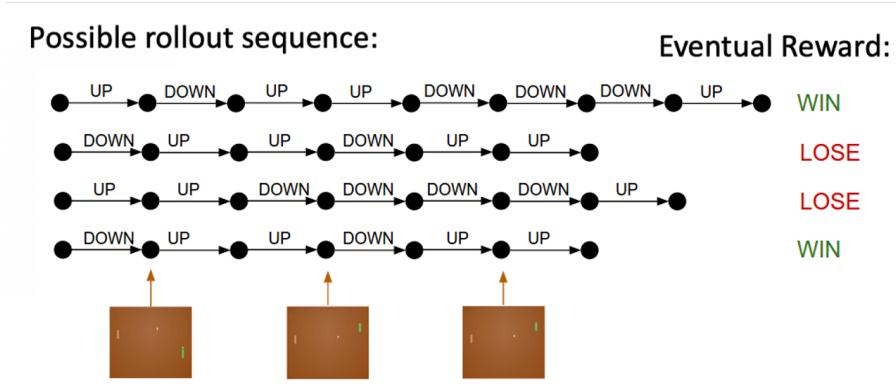


图 1.9 可能的 rollout 序列

最后结束过后，你会知道你到底有没有把这个球击到对方区域，对方没有接住，你是赢了还是输了。我们可以通过观测序列以及最终奖励 (eventual reward) 来训练这个 agent，使它尽可能地采取可以获得这个最终奖励的动作。

一场游戏叫做一个episode(回合) 或者trial(试验)。

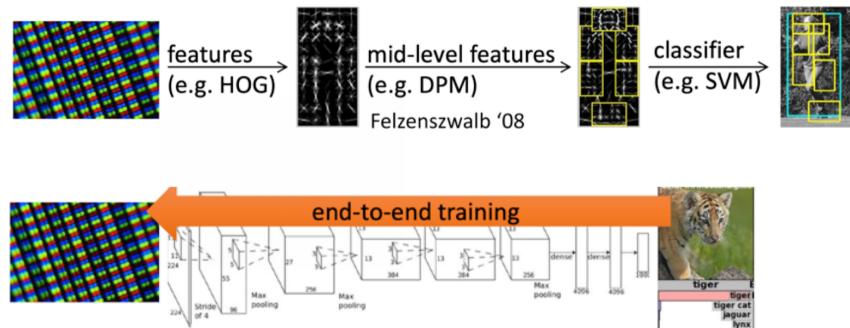


图 1.10 传统计算机视觉对比深度计算机视觉

强化学习是有一定的历史的，只是最近大家把强化学习跟深度学习结合起来，就形成了深度强化学习 (Deep Reinforcement Learning)。深度强化学习 = 深度学习 + 强化学习。这里做一个类比，把它类比于这个传统的计算机视觉以及深度计算机视觉。

如图 1.10 所示，传统的计算机视觉由两个过程组成。

- 给定一张图，我们先要提取它的特征，用一些设计好的特征 (feature)，比如说 HOG、DPM。
- 提取这些特征后，我们再单独训练一个分类器。这个分类器可以是 SVM、Boosting，然后就可以辨别这张图片是狗还是猫。

2012 年过后，我们有了卷积神经网络，大家就把特征提取以及分类两者合到一块儿去了，就是训练一个神经网络。这个神经网络既可以做特征提取，也可以做分类。它可以实现这种端到端的训练，它里面的参数可以在每一个阶段都得到极大的优化，这样就得到了一个非常重要的突破。

我们可以把神经网络放到强化学习里面，如图 1.11 所示，

- Standard RL: 之前的强化学习，比如 TD-Gammon 玩 backgammon 这个游戏，它其实是设计特征，然后通过训练价值函数的一个过程，就是它先设计了很多手工的特征，这个手工特征可以描述现在整个状态。得到这些特征过后，它就可以通过训练一个分类网络或者分别训练一个价值估计函数来做出决策。
- Deep RL: 现在我们有了深度学习，有了神经网络，那么大家也把这个过程改进成一个端到端训练 (end-to-end training) 的过程。你直接输入这个状态，我们不需要去手工地设计这个特征，就可以让它直接输出动作。那么就可以用一个神经网络来拟合我们这里的值函数或策略网络，省去了特征

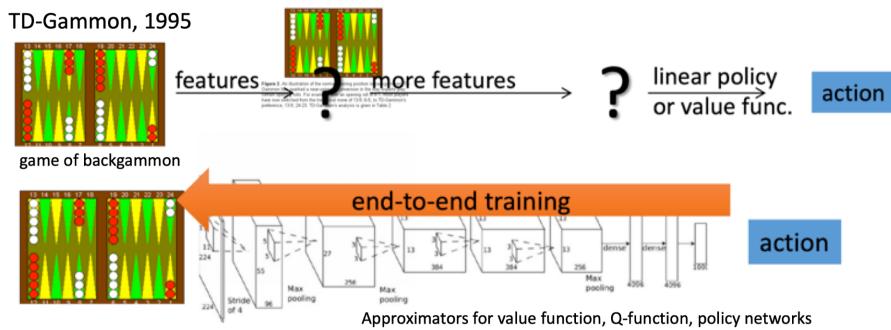


图 1.11 Standard RL 对比 Deep RL

工程 (feature engineering) 的过程。

为什么强化学习在这几年就用到各种应用中去，比如玩游戏以及机器人的一些应用，并且可以击败人类的最好棋手。这有如下几点原因：

- 我们有了更多的计算能力 (computation power)，有了更多的 GPU，可以更快地做更多的试错的尝试。
- 通过这种不同尝试使得 agent 在这个环境里面获得很多信息，然后可以在这个环境里面取得很大的奖励。
- 我们有了这个端到端的一个训练，可以把特征提取和价值估计或者决策一块来优化，这样就可以得到了一个更强的决策网络。

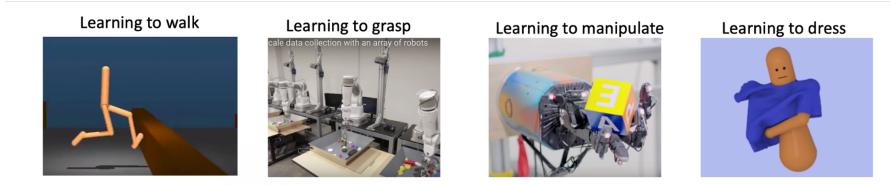


图 1.12 强化学习例子

接下来给大家再看一些强化学习里面比较有意思的例子。

1. DeepMind 研发的一个走路的 agent。这个 agent 往前走一步，你就会得到一个 reward。这个 agent 有不同的这个形态，可以学到很多有意思的功能。比如怎么跨越这个障碍物，就像那个蜘蛛那样的 agent。怎么跨越障碍物，像这个人有双腿一样，这个 agent 往前走。以及像这个人形的 agent，怎么在一个曲折的道路上面往前走。这个结果也是非常有意思，这个人形 agent 会把手举得非常高，因为它这个手的功能就是为了使它身体保持平衡，这样它就可以更快地在这个环境里面往前跑，而且这里你也可以增加这个环境的难度，加入一些扰动，这个 agent 就会变得更鲁棒。
2. 机械臂抓取。因为机械臂的应用自动去强化学习需要大量的 rollout，所以它这里就有好多机械臂，分布式系统可以让这个机械臂尝试抓取不同的物体。你发现这个盘子里面物体的形状、形态其实都是不同的，这样就可以让这个机械臂学到一个统一的行为。然后在不同的抓取物下面都可以采取最优的一个抓取特征。你的这个抓取的物件形态存在很多不同，一些传统的这个抓取算法就没法把所有物体都抓起来，因为你对每一个物体都需要做一个建模，这样的话就是非常花时间。但是通过强化学习，你就可以学到一个统一的抓取算法，在不同物体上它都可以适用。
3. OpenAI 做的一个机械臂翻魔方。这里它们 18 年的时候先设计了这个手指的一个机械臂，让它可以通过翻动手指，使得手中的这个木块达到一个预定的设定。人的手指其实非常精细，怎么使得这个机械手臂也具有这样灵活的能力就一直是个问题。它们通过这个强化学习在一个虚拟环境里面先训练，让 agent 能翻到特定的这个方向，再把它应用到真实的手臂之中。这在强化学习里面是一个比较常用的做法，就是你先在虚拟环境里面得到一个很好的 agent，然后再把它使用到真实的这个机器

人中。因为真实的机械手臂通常都是非常容易坏，而且非常贵，你没法大批量地购买。2019 年对手臂进一步改进了，这个手臂可以玩魔方了。这个结果也非常有意思，到后面，这个魔方就被恢复成了一个六面都是一样的结构了。

4. 一个穿衣服的 agent，就是训练这个 agent 穿衣服。因为很多时候你要在电影或者一些动画实现人穿衣服的场景，通过手写执行命令让机器人穿衣服其实非常困难。很多时候穿衣服也是一个非常精细的操作，那么它们这个工作就是训练这个强化学习 agent，然后就可以实现这个穿衣功能。你还可以在这里面加入一些扰动，然后 agent 可以抗扰动。可能会有失败的情况 (failure case)，agent 就穿不进去，就卡在这个地方。

1.2 Introduction to Sequential Decision Making

1.2.1 Agent and Environment

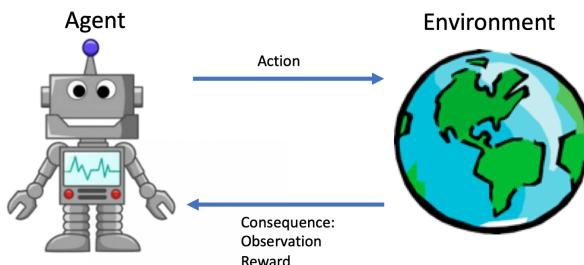


图 1.13 agent 和环境

接下来我们讲序列决策 (Sequential Decision Making) 过程。

强化学习研究的问题是 agent 跟环境交互，上图左边画的是一个 agent，agent 一直在跟环境进行交互。这个 agent 把它输出的动作给环境，环境取得这个动作过后，会进行到下一步，然后会把下一步的观测跟它上一步是否得到奖励返还给 agent。

通过这样的交互过程会产生很多观测，agent 的目的是从这些观测之中学到能极大化奖励的策略。

1.2.2 Reward

奖励是由环境给的一个标量的反馈信号 (scalar feedback signal)，这个信号显示了 agent 在某一步采取了某个策略的表现如何。

强化学习的目的就是为了最大化 agent 可以获得的奖励，agent 在这个环境里面存在的目的就是为了极大化它的期望的累积奖励 (expected cumulative reward)。

不同的环境，奖励也是不同的。这里给大家举一些奖励的例子。

- 比如说一个下象棋的选手，他的目的其实就为了赢棋。奖励是说在最后棋局结束的时候，他知道会得到一个正奖励或者负奖励。
- 羚羊站立也是一个强化学习过程，它得到的奖励就是它是否可以最后跟它妈妈一块离开或者它被吃掉。
- 在股票管理里面，奖励定义由你的股票获取的收益跟损失决定。
- 在玩雅达利游戏的时候，奖励就是你有没有在增加游戏的分数，奖励本身的稀疏程度决定了这个游戏的难度。

1.2.3 Sequential Decision Making

在一个强化学习环境里面，agent 的目的就是选取一系列的动作来极大化它的奖励，所以这些采取的动作必须有长期的影响。但在这个过程里面，它的奖励其实是被延迟了，就是说你现在采取的某一步决策

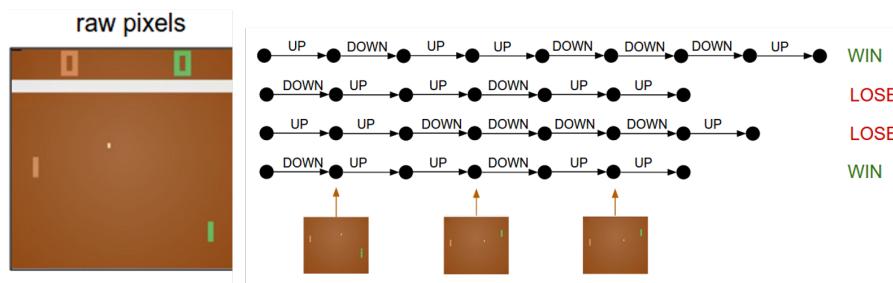


图 1.14

可能要等到时间很久过后才知道这一步到底产生了什么样的影响。

这里一个示意图就是我们玩这个 Atari 的 Pong 游戏，你可能只有到最后游戏结束过后，才知道这个球到底有没有击打过去。中间你采取的 up 或 down 行为，并不会直接产生奖励。强化学习里面一个重要的课题就是近期奖励和远期奖励的一个权衡 (trade-off)。怎么让 agent 取得更多的长期奖励是强化学习的问题。

在跟环境的交互过程中，agent 会获得很多观测。在每一个观测会采取一个动作，它也会得到一个奖励。所以历史是观测 (observation)、行为、奖励的序列：

$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t \quad (1.1)$$

Agent 在采取当前动作的时候会依赖于它之前得到的这个历史，所以你可以把整个游戏的状态看成关于这个历史的函数：

$$S_t = f(H_t) \quad (1.2)$$

Q: 状态和观测有什么关系？

A: 状态 (state) s 是对世界的完整描述，不会隐藏世界的信息。观测 (observation) o 是对状态的部分描述，可能会遗漏一些信息。在 deep RL 中，我们几乎总是用一个实值的向量、矩阵或者更高阶的张量来表示状态和观测。举个例子，我们可以用 RGB 像素值的矩阵来表示一个视觉的观测，我们可以用机器人关节的角度和速度来表示一个机器人的状态。

环境有自己的函数 $S_t^e = f^e(H_t)$ 来更新状态，在 agent 的内部也有一个函数 $S_t^a = f^a(H_t)$ 来更新状态。当 agent 的状态跟环境的状态等价的时候，我们就说这个环境是 full observability，就是全部可以观测。换句话说，当 agent 能够观察到环境的所有状态时，我们称这个环境是完全可观测的 (fully observed)。在这种情况下，强化学习通常被建模成一个 Markov decision process(MDP) 的问题。在 MDP 中， $O_t = S_t^e = S_t^a$ 。

但是有一种情况是 agent 得到的观测并不能包含环境运作的所有状态，因为在这个强化学习的设定里面，环境的状态才是真正的所有状态。

比如 agent 在玩这个 black jack 这个游戏，它能看到的其实是牌面上的牌。或者在玩雅达利游戏的时候，观测到的只是当前电视上面这一帧的信息，你并没有得到游戏内部里面所有的运作状态。

也就是说当 agent 只能看到部分的观测，我们就称这个环境是部分可观测的 (partially observed)。在这种情况下，强化学习通常被建模成一个 POMDP 的问题。

部分可观测马尔可夫决策过程 (Partially Observable Markov Decision Processes, POMDP) 是一个马尔可夫决策过程的泛化。POMDP 依然具有马尔可夫性质，但是假设智能体无法感知环境的状态 s ，只能知道部分观测值 o 。比如在自动驾驶中，智能体只能感知传感器采集的有限的环境信息。

POMDP 可以用一个 7 元组描述： $(S, A, T, R, \Omega, O, \gamma)$ ，其中 S 表示状态空间，为隐变量， A 为动作空间， $T(s'|s, a)$ 为状态转移概率， R 为奖励函数， $\Omega(o|s, a)$ 为观测概率， O 为观测空间， γ 为折扣系数。

1.3 Action Spaces

不同的环境允许不同种类的动作。在给定的环境中，有效动作的集合经常被称为动作空间 (action space)。像 Atari 和 Go 这样的环境有离散动作空间 (discrete action spaces)，在这个动作空间里，agent 的动作数量是有限的。在其他环境，比如在物理世界中控制一个 agent，在这个环境中就有连续动作空间 (continuous action spaces)。在连续空间中，动作是实值的向量。

例如，走迷宫机器人如果只有东南西北这 4 种移动方式，则其为离散动作空间；如果机器人向 360° 中的任意角度都可以移动，则为连续动作空间。

1.4 Major Components of an RL Agent

对于一个强化学习 agent，它可能有一个或多个如下的组成成分：

- 策略函数 (policy function)，agent 会用这个函数来选取下一步的动作。
- 价值函数 (value function)。我们用价值函数来对当前状态进行估价，它就是说你进入现在这个状态，可以对你后面的收益带来多大的影响。当这个价值函数大的时候，说明你进入这个状态越有利。
- 模型 (model)。模型表示了 agent 对这个环境的状态进行了理解，它决定了这个世界是如何进行的。

1.4.1 Policy

我们深入看这三个组成成分的一些细节。

Policy 是 agent 的行为模型，它决定了这个 agent 的行为，它其实是一个函数，把输入的状态变成行为。这里有两种 policy：

stochastic policy(随机性策略)，它就是 π 函数 $\pi(a|s) = P[A_t = a|S_t = s]$ 。当你输入一个状态 s 的时候，输出是一个概率。这个概率就是你所有行为的一个概率，然后你可以进一步对这个概率分布进行采样，得到真实的你采取的行为。比如说这个概率可能是有 70% 的概率往左，30% 的概率往右，那么你通过采样就可以得到一个 action。

deterministic policy(确定性策略)，就是说你这里有可能只是采取它的极大化，采取最有可能的动作，即 $a^* = \arg \max_a \pi(a | s)$ 。你现在这个概率就是事先决定好的。

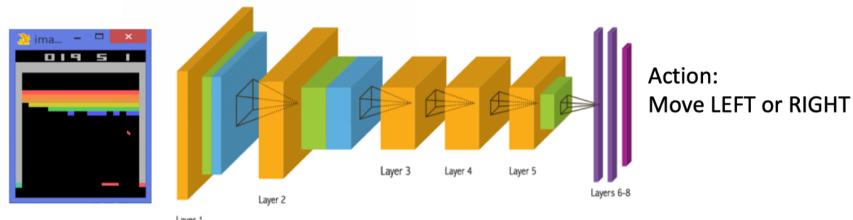


图 1.15

从 Atari 游戏来看的话，策略函数的输入就是游戏的一帧，它的输出决定你是往左走或者是往右走。

通常情况下，强化学习一般使用随机性策略。随机性策略有很多优点：

- 在学习时可以通过引入一定随机性来更好地探索环境；
- 随机性策略的动作具有多样性，这一点在多个智能体博弈时也非常重要。采用确定性策略的智能体总是对同样的环境做出相同动作，会导致它的策略很容易被对手预测。

1.4.2 Value Function

价值函数是未来奖励的一个预测，用来评估状态的好坏。

价值函数里面有一个discount factor(折扣因子)，我们希望尽可能在短的时间里面得到尽可能多的奖励。如果我们说十天过后，我给你100块钱，跟我现在给你100块钱，你肯定更希望我现在就给你100块钱，因为你可以把这100块钱存在银行里面，你就会有一些利息。所以我们就通过把这个折扣因子放到价值函数的定义里面，价值函数的定义其实是一个期望，如式(1.3)所示：

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \text{ for all } s \in \mathcal{S} \quad (1.3)$$

这里有一个期望 \mathbb{E}_{π} ，这里有个小角标是 π 函数，这个 π 函数就是说在我们已知某一个策略函数的时候，到底可以得到多少的奖励。

我们还有一种价值函数：Q 函数。Q 函数里面包含两个变量：状态和动作，其定义如式(2.25)所示：

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \quad (1.4)$$

所以你未来可以获得多少的奖励，它的这个期望取决于你当前的状态和当前的行为。这个 Q 函数是强化学习算法里面要学习的一个函数。因为当我们得到这个 Q 函数后，进入某一种状态，它最优的行为就可以通过这个 Q 函数来得到。

1.4.3 Model

第三个组成部分是模型，模型决定了下一个状态会是什么样的，就是说下一步的状态取决于你当前的状态以及你当前采取的行为。它由概率和奖励函数两个部分组成，概率：这个转移状态之间是怎么转移的，如所式(1.5)所示：

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \quad (1.5)$$

奖励函数：当你在当前状态采取了某一个行为，可以得到多大的奖励，如式(1.6)所示。

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \quad (1.6)$$

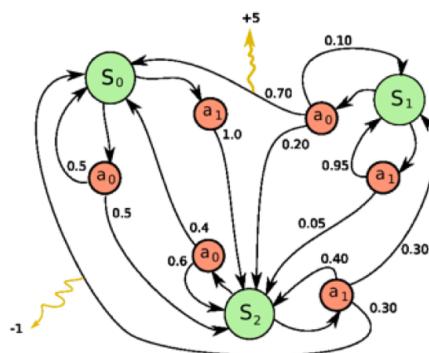


图 1.16 马尔可夫决策过程

当我们有了这三个组成部分过后，就形成了一个马尔可夫决策过程 (Markov Decision Process)。如图 1.16 所示，这个决策过程可视化了状态之间的转移以及采取的行为。

我们来看一个走迷宫的例子。如图 1.17 所示，要求 agent 从 start 开始，然后到达 goal 的位置；每走一步，你就会得到 -1 的奖励；可以采取的动作是往上下左右走；当前状态用现在 agent 所在的位置来描述。

我们可以用不同的强化学习算法来解这个环境。

如果采取的是基于策略的 (policy-based)RL，当学习好了这个环境过后，在每一个状态，我们就会得到一个最佳的行为。

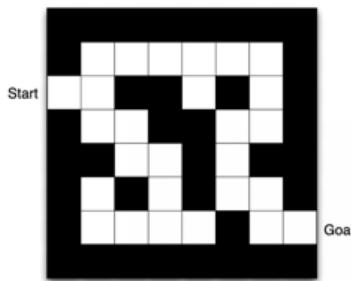


图 1.17 走迷宫的例子

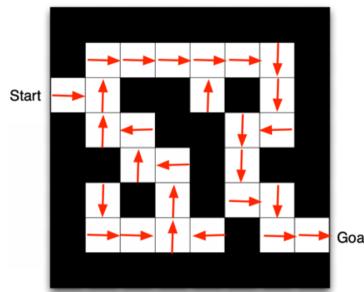


图 1.18 使用基于策略的 RL 得到的结果

如图 1.18 所示，比如说现在在第一格开始的时候，我们知道它最佳行为是往右走，然后第二格的时候，得到的最佳策略是往上走，第三格是往右走。通过这个最佳的策略，我们就可以最快地到达终点。

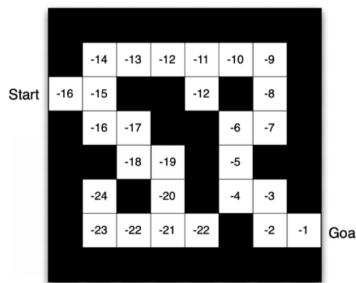


图 1.19 使用基于价值的 RL 得到的结果

如果换成基于价值的 (value-based)RL 这个算法，利用价值函数来作为导向，我们就会得到另外一种表征，这里就表征了你每一个状态会返回一个价值。

如图 1.19 所示，比如说你在 start 位置的时候，价值是 -16，因为你最快可以 16 步到达终点。因为每走一步会减一，所以你这里的值是 -16。

当我们快接近最后终点的时候，这个数字变得越来越大。在拐角的时候，比如要现在在第二格 -15。然后 agent 会看上下，它看到上面值变大了，变成 -14 了，它下面是 -16，那么 agent 肯定就会采取一个往上走的策略。所以通过这个学习的值的不同，我们可以抽取出现在最佳的策略。

1.4.4 Types of RL Agents

根据 agent 学习的东西不同，我们可以把 agent 进行归类。

基于价值的 agent(value-based agent)。这一类 agent 显式地学习的是价值函数，隐式地学习了它的策略。策略是从我们学到的价值函数里面推算出来的。

基于策略的 agent(policy-based agent)。这一类 agent 直接去学习 policy，就是说你直接给它一个状

态，它就会输出这个动作的概率。在基于策略的 agent 里面并没有去学习它的价值函数。

把 value-based 和 policy-based 结合起来就有了 [Actor-Critic agent](#)。这一类 agent 把它的策略函数和价值函数都学习了，然后通过两者的交互得到一个最佳的行为。

Q: 基于策略迭代和基于价值迭代的强化学习方法有什么区别？

A: 对于一个状态转移概率已知的马尔可夫决策过程，我们可以使用动态规划算法来求解；从决策方式来看，强化学习又可以划分为基于策略迭代的方法和基于价值迭代的方法。决策方式是智能体在给定状态下从动作集合中选择一个动作的依据，它是静态的，不随状态变化而变化。

在[基于策略迭代](#)的强化学习方法中，智能体会制定一套动作策略（确定在给定状态下需要采取何种动作），并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励。

而在基于价值迭代的强化学习方法中，智能体不需要制定显式的策略，它维护一个价值表格或价值函数，并通过这个价值表格或价值函数来选取价值最大的动作。基于价值迭代的方法只能应用在不连续的、离散的环境下（如围棋或某些游戏领域），对于行为集合规模庞大、动作连续的场景（如机器人控制领域），其很难学习到较好的结果（此时基于策略迭代的方法能够根据设定的策略来选择连续的动作）。

基于价值迭代的强化学习算法有 Q-learning、Sarsa 等，而基于策略迭代的强化学习算法有策略梯度算法等。此外，Actor-Critic 算法同时使用策略和价值评估来做出决策，其中，智能体会根据策略做出动作，而价值函数会对做出的动作给出价值，这样可以在原有的策略梯度算法的基础上加速学习过程，取得更好的效果。

另外，我们是可以通过 agent 到底有没有学习这个环境模型来分类。[model-based\(有模型\)](#) RL agent，它通过学习这个状态的转移来采取动作。[model-free\(免模型\)](#) RL agent，它没有去直接估计这个状态的转移，也没有得到环境的具体转移变量。它通过学习价值函数和策略函数进行决策。Model-free 的模型里面没有一个环境转移的模型。

我们可以用马尔可夫决策过程来定义强化学习任务，并表示为四元组 $\langle S, A, P, R \rangle$ ，即状态集合、动作集合、状态转移函数和奖励函数。如果这四元组中所有元素均已知，且状态集合和动作集合在有限步数内是有限集，则机器可以对真实环境进行建模，构建一个虚拟世界来模拟真实环境的状态和交互反应。

具体来说，当智能体知道状态转移函数 $P(s_{t+1}|s_t, a_t)$ 和奖励函数 $R(s_t, a_t)$ 后，它就能知道在某一状态下执行某一动作后能带来的奖励和环境的下一状态，这样智能体就不需要在真实环境中采取动作，直接在虚拟世界中学习和规划策略即可。这种学习方法称为[有模型学习](#)。

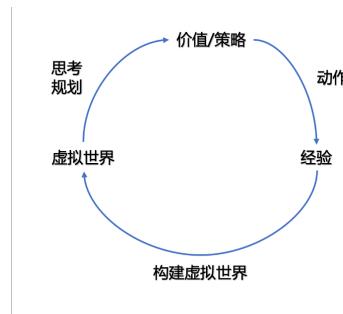


图 1.20 有模型强化学习流程图

有模型强化学习的流程图如图 1.20 所示。

然而在实际应用中，智能体并不是那么容易就能知晓 MDP 中的所有元素的。通常情况下，状态转移函数和奖励函数很难估计，甚至连环境中的状态都可能是未知的，这时就需要采用免模型学习。免模型学习没有对真实环境进行建模，智能体只能在真实环境中通过一定的策略来执行动作，等待奖励和状态迁移，然后根据这些反馈信息来更新行为策略，这样反复迭代直到学习到最优策略。

Q: 有模型强化学习和免模型强化学习有什么区别？

A: 针对是否需要对真实环境建模，强化学习可以分为有模型学习和免模型学习。有模型学习是指根

据环境中的经验，构建一个虚拟世界，同时在真实环境和虚拟世界中学习；免模型学习是指不对环境进行建模，直接与真实环境进行交互来学习到最优策略。

总的来说，有模型学习相比于免模型学习仅仅多出一个步骤，即对真实环境进行建模。因此，一些有模型的强化学习方法，也可以在免模型的强化学习方法中使用。在实际应用中，如果不清楚该用有模型强化学习还是免模型强化学习，可以先思考一下，在智能体执行动作前，是否能对下一步的状态和奖励进行预测，如果可以，就能够对环境进行建模，从而采用有模型学习。

免模型学习通常属于数据驱动型方法，需要大量的采样来估计状态、动作及奖励函数，从而优化动作策略。例如，在 Atari 平台上的 Space Invader 游戏中，免模型的深度强化学习需要大约 2 亿帧游戏画面才能学到比较理想的效果。相比之下，有模型学习可以在一定程度上缓解训练数据匮乏的问题，因为智能体可以在虚拟世界中行训练。

免模型学习的泛化性要优于有模型学习，原因是免模型学习算需要对真实环境进行建模，并且虚拟世界与真实环境之间可能还有差异，这限制了有模型学习算法的泛化性。

有模型的强化学习方法可以对环境建模，使得该类方法具有独特魅力，即“想象能力”。在免模型学习中，智能体只能一步一步地采取策略，等待真实环境的反馈；而有模型学习可以在虚拟世界中预测出所有将要发生的事，并采取对自己最有利的策略。

目前，大部分深度强化学习方法都采用了免模型学习，这是因为：免模型学习更为简单直观且有丰富的开源资料，像 DQN、AlphaGo 系列等都采用免模型学习；在目前的强化学习研究中，大部分情况下环境都是静态的、可描述的，智能体的状态是离散的、可观察的（如 Atari 游戏平台），这种相对简单确定的问题并不需要评估状态转移函数和奖励函数，直接采用免模型学习，使用大量的样本进行训练就能获得较好的效果。

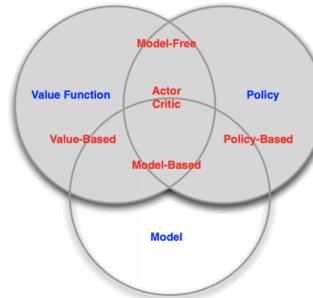


图 1.21 强化学习 agent 的类型

把几类模型放到同一个饼图里面。饼图有三个组成部分：价值函数、策略和模型。按一个 agent 具不具有三者中的两者或者一者可以把它分成很多类。

1.5 Learning and Planning

Learning 和 Planning 是序列决策的两个基本问题。

如图 1.22 所示，在强化学习中，环境初始时是未知的，agent 不知道环境如何工作，agent 通过不断地与环境交互，逐渐改进策略。

如图 1.23 所示，在 planning 中，环境是已知的，我们被告知了整个环境的运作规则的详细信息。Agent 能够计算出一个完美的模型，并且在不需要与环境进行任何交互的时候进行计算。Agent 不需要实时地与环境交互就能知道未来环境，只需要知道当前的状态，就能够开始思考，来寻找最优解。

在这个游戏中，规则是制定的，我们知道选择 left 之后环境将会产生什么变化。我们完全可以通过已知的变化规则，来在内部进行模拟整个决策过程，无需与环境交互。

一个常用的强化学习问题解决思路是，先学习环境如何工作，也就是了解环境工作的方式，即学习得到一个模型，然后利用这个模型进行规划。

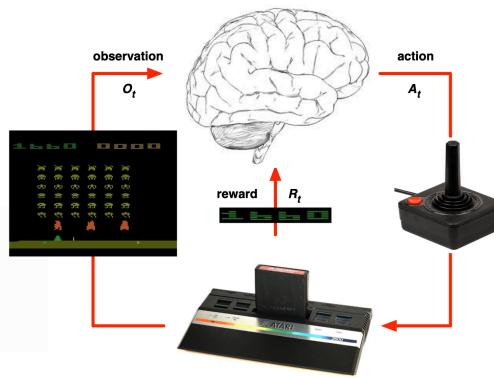


图 1.22 learning

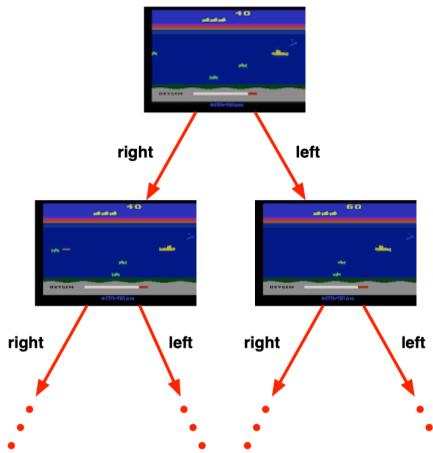


图 1.23 planning

1.6 Exploration and Exploitation

在强化学习里面，探索和利用是两个很核心的问题。探索是说我们怎么去探索这个环境，通过尝试不同的行为来得到一个最佳的策略，得到最大奖励的策略。利用是说我们不去尝试新的东西，就采取已知的可以得到很大奖励的行为。

因为在刚开始的时候强化学习 agent 不知道它采取了某个行为会是什么，所以它只能通过试错去探索。所以探索就是在试错来理解采取的这个行为到底可不可以得到好的奖励。利用是说我们直接采取已知的可以得到很好奖励的行为。所以这里就面临一个权衡，怎么通过牺牲一些短期的奖励来获得行为的理解，从而学习到更好的策略。

下面举一些探索和利用的例子。

- 以选择餐馆为例，
 - 利用：我们直接去你最喜欢的餐馆，因为你去过这个餐馆很多次了，所以你知道这里面的菜都非常可口。
 - 探索：你把手机拿出来，你直接搜索一个新的餐馆，然后去尝试它到底好不好吃。你有可能对这个新的餐馆非常不满意，钱就浪费了。
- 以做广告为例，
 - 利用：我们直接采取最优的这个广告策略。
 - 探索：我们换一种广告策略，看看这个新的广告策略到底可不可以得到奖励。
- 以挖油为例，

- 利用：我们直接在已知的地方挖油，我们就可以确保挖到油。
- 探索：我们在一个新的地方挖油，就有很大的概率，你可能不能发现任何油，但也可能有比较小的概率可以发现一个非常大的油田。
- 以玩游戏为例，
 - 利用：你总是采取某一种策略。比如说，你可能打街霸，你采取的策略可能是蹲在角落，然后一直触脚。这个策略很可能可以奏效，但可能遇到特定的对手就失效。
 - 探索：你可能尝试一些新的招式，有可能你会发出大招来，这样就可能一招毙命。

1.6.1 K-armed Bandit

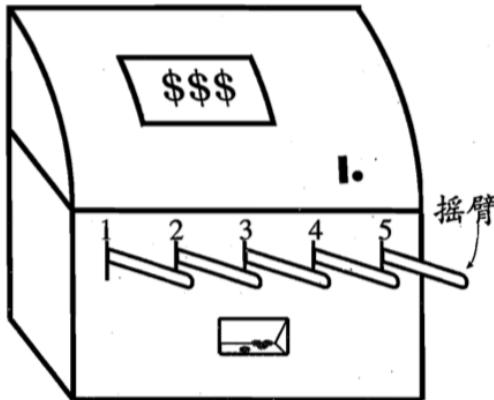


图 1.24 K-臂赌博机图示

与监督学习不同，强化学习任务的最终奖赏是在多步动作之后才能观察到，这里我们不妨先考虑比较简单的情形：最大化单步奖赏，即仅考虑一步操作。需注意的是，即便在这样的简化情形下，强化学习仍与监督学习有显著不同，因为机器需通过尝试来发现各个动作产生的结果，而没有训练数据告诉机器应当做哪个动作。

想要最大化单步奖赏需考虑两个方面：一是需知道每个动作带来的奖赏，二是要执行奖赏最大的动作。若每个动作对应的奖赏是一个确定值，那么尝试遍所有的动作便能找出奖赏最大的动作。然而，更一般的情形是，一个动作的奖赏值是来自于一个概率分布，仅通过一次尝试并不能确切地获得平均奖赏值。

实际上，单步强化学任务对应了一个理论模型，即**K-臂赌博机 (K-armed bandit)**。K-臂赌博机也被称为**多臂赌博机 (Multi-armed bandit)**。如上图所示，K-摇臂赌博机有 K 个摇臂，赌徒在投入一个硬币后可选择按下其中一个摇臂，每个摇臂以一定的概率吐出硬币，但这个概率赌徒并不知道。赌徒的目标是通过一定的策略最大化自己的奖赏，即获得最多的硬币。

- 若仅为获知每个摇臂的期望奖赏，则可采用**仅探索 (exploration-only) 法**：将所有的尝试机会平均分配给每个摇臂（即轮流按下每个摇臂），最后以每个摇臂各自的平均吐币概率作为其奖赏期望的近似估计。
- 若仅为执行奖赏最大的动作，则可采用**仅利用 (exploitation-only) 法**：按下目前最优的（即到目前为止平均奖赏最大的）摇臂，若有多个摇臂同为最优，则从中随机选取一个。

显然，仅探索法能很好地估计每个摇臂的奖赏，却会失去很多选择最优摇臂的机会；仅利用法则相反，它没有很好地估计摇臂期望奖赏，很可能经常选不到最优摇臂。因此，这两种方法都难以使最终的累积奖赏最大化。

事实上，探索（即估计摇臂的优劣）和利用（即选择当前最优摇臂）这两者是矛盾的，因为尝试次数（即总投币数）有限，加强了一方则会自然削弱另一方，这就是强化学习所面临的**探索-利用窘境 (Exploration-Exploitation dilemma)**。显然，想要累积奖赏最大，则必须在探索与利用之间达成较好的折中。

1.7 Experiment with Reinforcement Learning

强化学习是一个理论跟实践结合的机器学习分支，需要去推导很多算法公式，去理解它算法背后的一些数学原理。另外一方面，上机实践通过实现算法，在很多实验环境里面去探索这个算法是不是可以得到预期效果也是一个非常重要的过程。

在这个链接里面，公布了一些 RL 相关的代码，利用了 Python 和深度学习的一些包（主要是用 PyTorch 为主）。

你可以直接调用现有的包来实践。现在有很多深度学习的包可以用，比如 PyTorch、TensorFlow、Keras，熟练使用这里面的两三种，就可以实现非常多的功能。所以你并不需要从头去造轮子。

OpenAI是一个非盈利的人工智能研究公司。Open AI 公布了非常多的学习资源以及算法资源，他们之所以叫 Open AI，就是他们把所有开发的算法都 open source 出来。

1.7.1 Gym

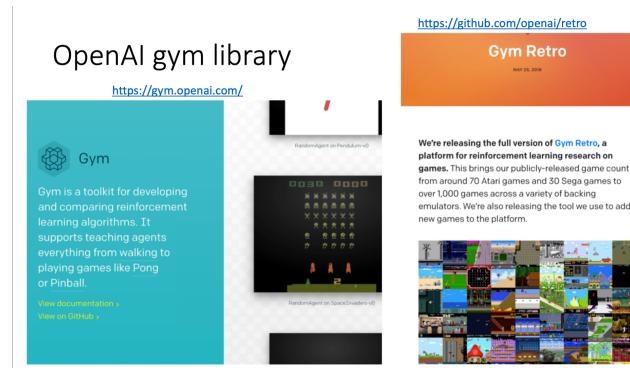


图 1.25 OpenAI gym 库

OpenAI Gym 是一个环境仿真库，里面包含了很多现有的环境。针对不同的场景，我们可以选择不同的环境，

- 离散控制场景（输出的动作是可数的，比如 Pong 游戏中输出的向上或向下动作）：一般使用 Atari 环境评估
- 连续控制场景（输出的动作是不可数的，比如机器人走路时不仅有方向，还要角度，角度就是不可数的，是一个连续的量）：一般使用 mujoco 环境评估

Gym Retro 是对 Gym 环境的进一步扩展，包含了更多的一些游戏。

我们可以通过 pip 来安装 Gym：

```
pip install gym
```

在 Python 环境中导入 Gym，如果不报错，就可以认为 Gym 安装成功。

```
$python
>>>import gym
```

```
import gym
env = gym.make("Taxi-v3")
observation = env.reset()
agent = load_agent()
for step in range(100):
    action = agent(observation)
    observation, reward, done, info = env.step(action)
```

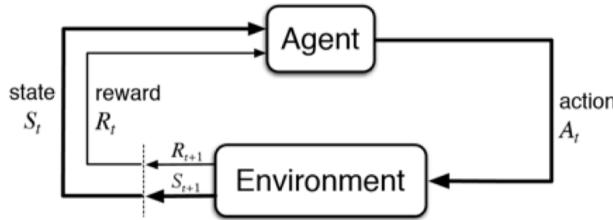


图 1.26 强化学习的算法界面

强化学习的这个交互就是由 agent 跟环境进行交互。所以算法的 interface 也是用这个来表示。比如说我们现在安装了 OpenAI Gym。

1. 我们就可以直接调入 Taxi-v3 的环境，就建立了这个环境。
2. 初始化这个环境过后，就可以进行交互了。
3. Agent 得到这个观测过后，它就会输出一个 action。
4. 这个动作会被环境拿进去执行这个 step，然后环境就会往前走一步，返回新的 observation、reward 以及一个 flag variable `done`，`done` 决定这个游戏是不是结束了。

几行代码就实现了强化学习的框架。

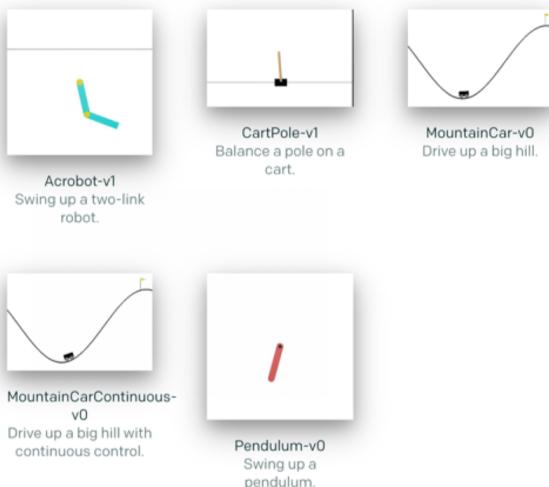


图 1.27 经典控制问题

在 OpenAI Gym 里面有很经典的控制类游戏。

- 比如说 Acrobot 就是把两节铁杖甩了立起来。
- CartPole 是通过控制一个平板，让木棍立起来。
- MountainCar 是通过前后移动这个车，让它到达这个旗子的位置。

大家可以点这个链接看一看这些环境。在刚开始测试强化学习的时候，可以选择这些简单环境，因为这些环境可以在一两分钟之内见到一个效果。

这里我们看一下 CartPole 的这个环境。对于这个环境，有两个动作，Cart 往左移还是往右移。这里得到了观测：这个车当前的位置，Cart 当前往左往右移的速度，这个杆的角度以及杆的最高点的速度。

如果 observation 越详细，就可以更好地描述当前这个所有的状态。这里有 reward 的定义，如果能多保留一步，你就会得到一个奖励，所以你需要在尽可能多的时间存活来得到更多的奖励。当这个杆的角度大于某一个角度（没能保持平衡）或者这个车已经出到外面的时候，游戏就结束了，你就输了。所以这个 agent 的目的就是为了控制木棍，让它尽可能地保持平衡以及尽可能保持在这个环境的中央。

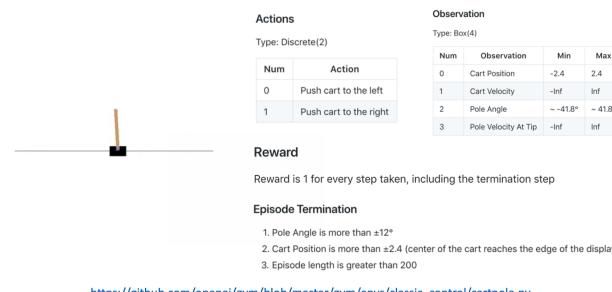


图 1.28 CartPole-v0 的例子

```
import gym # 导入 Gym 的 Python 接口环境包
env = gym.make('CartPole-v0') # 构建实验环境
env.reset() # 重置一个 episode
for _ in range(1000):
    env.render() # 显示图形界面
    action = env.action_space.sample() # 从动作空间中随机选取一个动作
    env.step(action) # 用于提交动作，括号内是具体的动作
env.close() # 关闭环境
```

注意：如果绘制了实验的图形界面窗口，那么关闭该窗口的最佳方式是调用 `env.close()`。试图直接关闭图形界面窗口可能会导致内存不能释放，甚至会导致死机。

当你执行这段代码时，机器人会完全无视那根本该立起来的杆子，驾驶着小车朝某个方向一通跑，直到不见踪影，这是因为我们还没开始训练机器人。

Gym 中的小游戏，大部分都可以用一个普通的实数或者向量来充当动作。

打印 `env.action_space.sample()` 的返回值，能看到输出为 1 或者 0。

`env.action_space.sample()` 的含义是，在该游戏的所有动作空间里随机选择一个作为输出。在这个例子中，意思就是，动作只有两个：0 和 1，一左一右。

`env.step()` 这个方法的作用不止于此，它还有四个返回值，分别是 `observation`、`reward`、`done`、`info`。

`observation(object)` 是状态信息，是在游戏中观测到的屏幕像素值或者盘面状态描述信息。`reward(float)` 是奖励值，即 `action` 提交以后能够获得的奖励值。这个奖励值因游戏的不同而不同，但总体原则是，对完成游戏有帮助的动作会获得比较高的奖励值。`done(boolean)` 表示游戏是否已经完成。如果完成了，就需要重置游戏并开始一个新的 `episode`。`info(dict)` 是一些比较原始的用于诊断和调试的信息，或许对训练有帮助。不过，OpenAI 团队在评价你提交的机器人时，是不允许使用这些信息的。

在每个训练中都要使用的返回值有 `observation`、`reward`、`done`。但 `observation` 的结构会由于游戏的不同而发生变化。以 CartPole-v0 小游戏为例，我们修改下代码：

```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    print(observation)
env.close()
```

输出：

```
[ 0.01653398 0.19114579 0.02013859 -0.28050058]
[ 0.0203569 -0.00425755 0.01452858 0.01846535]
[ 0.02027175 -0.19958481 0.01489789 0.31569658]
.....
```

从输出可以看出这是一个四维的 Observation。在其他游戏中会有维度很多的情况。

`env.step()` 完成了一个完整的 $S \rightarrow A \rightarrow R \rightarrow S'$ 过程。我们只要不断观测这样的过程，并让机器在其中用相应的算法完成训练，就能得到一个高质量的强化学习模型。

想要查看当前 Gym 库已经注册了哪些环境，可以使用以下代码：

```
from gym import envs
env_specs = envs.registry.all()
envs_ids = [env_spec.id for env_spec in env_specs]
print(envs_ids)
```

每个环境都定义了自己的观测空间和动作空间。环境 `env` 的观测空间用 `env.observation_space` 表示，动作空间用 `env.action_space` 表示。观测空间和动作空间既可以是离散空间（即取值是有限个离散的值），也可以是连续空间（即取值是连续的）。在 Gym 库中，离散空间一般用 `gym.spaces.Discrete` 类表示，连续空间用 `gym.spaces.Box` 类表示。

例如，环境 '`MountainCar-v0`' 的观测空间是 `Box(2,)`，表示观测可以用 2 个 float 值表示；环境 '`MountainCar-v0`' 的动作空间是 `Discrete(3)`，表示动作取值自 0,1,2。对于离散空间，`gym.spaces.Discrete` 类实例的成员 `n` 表示有几个可能的取值；对于连续空间，`Box` 类实例的成员 `low` 和 `high` 表示每个浮点数的取值范围。

1.7.2 MountainCar-v0 Example

接下来，我们通过一个例子来学习如何与 Gym 库进行交互。我们选取[小车上山 \(MountainCar-v0\)](#)作为例子。

首先我们来看看这个任务的观测空间和动作空间：

```
import gym
env = gym.make('MountainCar-v0')
print('观测空间 = {}'.format(env.observation_space))
print('动作空间 = {}'.format(env.action_space))
print('观测范围 = {} ~ {}'.format(env.observation_space.low,
    env.observation_space.high))
print('动作数 = {}'.format(env.action_space.n))
```

输出：

```
观测空间 = Box(2,)
动作空间 = Discrete(3)
观测范围 = [-1.2 -0.07] ~ [0.6 0.07]
动作数 = 3
```

由输出可知，观测空间是形状为 (2,) 的浮点型 np.array，动作空间是取 0,1,2 的 int 型数值。

接下来考虑智能体。智能体往往是我们自己实现的。我们可以实现一个智能体类：`BespokeAgent` 类，代码如下所示：

```
class BespokeAgent:
    def __init__(self, env):
        pass
```

```

def decide(self, observation): # 决策
    position, velocity = observation
    lb = min(-0.09 * (position + 0.25) ** 2 + 0.03,
              0.3 * (position + 0.9) ** 4 - 0.008)
    ub = -0.07 * (position + 0.38) ** 2 + 0.07
    if lb < velocity < ub:
        action = 2
    else:
        action = 0
    return action # 返回动作

def learn(self, *args): # 学习
    pass

agent = BespokeAgent(env)

```

智能体的 `decide()` 方法实现了决策功能，而 `learn()` 方法实现了学习功能。`BespokeAgent`类是一个比较简单的类，它只能根据给定的数学表达式进行决策，不能有效学习。所以它并不是一个真正意义上的强化学习智能体类。但是，用于演示智能体和环境的交互已经足够了。

接下来我们试图让智能体与环境交互，代码如下所示：

```

def play_montecarlo(env, agent, render=False, train=False):
    episode_reward = 0. # 记录回合总奖励，初始化为0
    observation = env.reset() # 重置游戏环境，开始新回合
    while True: # 不断循环，直到回合结束
        if render: # 判断是否显示
            env.render() # 显示图形界面，图形界面可以用 env.close() 语句关闭
        action = agent.decide(observation)
        next_observation, reward, done, _ = env.step(action) # 执行动作
        episode_reward += reward # 收集回合奖励
        if train: # 判断是否训练智能体
            agent.learn(observation, action, reward, done) # 学习
        if done: # 回合结束，跳出循环
            break
        observation = next_observation
    return episode_reward # 返回回合总奖励

```

上面代码中的 `play_montecarlo` 函数可以让智能体和环境交互一个回合。这个函数有 4 个参数：

`env` 是环境类，`agent` 是智能体类，`render` 是 `bool` 类型变量，指示在运行过程中是否要图形化显示。如果函数参数 `render` 为 `True`，那么在交互过程中会调用 `env.render()` 以显示图形化界面，而这个界面可以通过调用 `env.close()` 关闭。

`train` 是 `bool` 类型的变量，指示在运行过程中是否训练智能体。在训练过程中应当设置为 `True`，以调用 `agent.learn()` 函数；在测试过程中应当设置为 `False`，使得智能体不变。

这个函数有一个返回值 `episode_reward`，是 `float` 类型的数值，表示智能体与环境交互一个回合的回合总奖励。

接下来，我们使用下列代码让智能体和环境交互一个回合，并在交互过程中图形化显示，可用 `env.close()` 语句关闭图形化界面。

```

env.seed(0) # 设置随机数种子，只是为了让结果可以精确复现，一般情况下可删去
episode_reward = play_montecarlo(env, agent, render=True)
print('回合奖励 = {}'.format(episode_reward))

```

```
env.close() # 此语句可关闭图形界面
```

输出：

```
回合奖励 = -105.0
```

为了系统评估智能体的性能，下列代码求出了连续交互 100 回合的平均回合奖励。

```
episode_rewards = [play_monte_carlo(env, agent) for _ in range(100)]
print('平均回合奖励 = {:.1f}'.format(np.mean(episode_rewards)))
```

输出：

```
平均回合奖励 = -102.61
```

小车上山环境有一个参考的回合奖励值 -110，如果当连续 100 个回合的平均回合奖励大于 -110，则认为这个任务被解决了。BespokeAgent 类对应的策略的平均回合奖励大概就在 -110 左右。

测试 agent 在 Gym 库中某个任务的性能时，学术界一般最关心 100 个回合的平均回合奖励。至于为什么是 100 个回合而不是其他回合数（比如 128 个回合），完全是习惯使然，没有什么特别的原因。对于有些环境，还会指定一个参考的回合奖励值，当连续 100 个回合的奖励大于指定的值时，就认为这个任务被解决了。但是，并不是所有的任务都指定了这样的值。对于没有指定值的任务，就无所谓任务被解决了或者没有被解决。

总结一下 Gym 的用法：使用 `env=gym.make(环境名)` 取出环境，使用 `env.reset()` 初始化环境，使用 `env.step(动作)` 执行一步环境，使用 `env.render()` 显示环境，使用 `env.close()` 关闭环境。

最后提一下，Gym 有对应的官方文档，大家可以阅读文档来学习 Gym。

1.8 Keywords

- 强化学习 (Reinforcement Learning): Agent 可以在与复杂且不确定的 Environment 进行交互时，尝试使所获得的 Reward 最化的计算算法。
- Action: Environment 接收到的 Agent 当前状态的输出。
- State: Agent 从 Environment 中获取到的状态。
- Reward: Agent 从 Environment 中获取的反馈信号，这个信号指定了 Agent 在某一步采取了某个策略以后是否得到奖励。
- Exploration: 在当前的情况下，继续尝试新的 Action，其有可能会使你得到更高的这个奖励，也有可能使你一无所有。
- Exploitation: 在当前的情况下，继续尝试已知的可以获得最大 Reward 的过程，即重复执行这个 Action 就可以了。
- 深度强化学习 (Deep Reinforcement Learning): 不需要手工设计特征，仅需要输入 State 让系统直接输出 Action 的一个 end-to-end training 的强化学习方法。通常使用神经网络来拟合 value function 或者 policy network。
- Full observability、fully observed 和 partially observed: 当 Agent 的状态跟 Environment 的状态等价的时候，我们就说现在 Environment 是 full observability (全部可观测)，当 Agent 能够观察到 Environment 的所有状态时，我们称这个环境是 fully observed (完全可观测)。一般我们的 Agent 不能观察到 Environment 的所有状态时，我们称这个环境是 partially observed (部分可观测)。
- POMDP (Partially Observable Markov Decision Processes): 部分可观测马尔可夫决策过程，即马尔可夫决策过程的泛化。POMDP 依然具有马尔可夫性质，但是假设智能体无法感知环境的状态 s ，只能知道部分观测值 o 。

- Action space (discrete action spaces and continuous action spaces): 在给定的 Environment 中，有效动作的集合经常被称为动作空间 (Action space)，Agent 的动作数量是有限的动作空间为离散动作空间 (discrete action spaces)，反之，称为连续动作空间 (continuous action spaces)。
- policy-based (基于策略的): Agent 会制定一套动作策略 (确定在给定状态下需要采取何种动作)，并根据这个策略进行操作。强化学习算法直接对策略进行优化，使制定的策略能够获得最大的奖励。
- valued-based (基于价值的): Agent 不需要制定显式的策略，它维护一个价值表格或价值函数，并通过这个价值表格或价值函数来选取价值最大的动作。
- model-based (有模型结构): Agent 通过学习状态的转移来采取措施。
- model-free (无模型结构): Agent 没有去直接估计状态的转移，也没有得到 Environment 的具体转移变量。它通过学习 value function 和 policy function 进行决策。

1.9 Questions

- 强化学习的基本结构是什么？

答：本质上是 Agent 和 Environment 间的交互。具体地，当 Agent 在 Environment 中得到当前时刻的 State，Agent 会基于此状态输出一个 Action。然后这个 Action 会加入到 Environment 中去并输出下一个 State 和当前的这个 Action 得到的 Reward。Agent 在 Environment 里面存在的目的就是为了极大它的期望积累的 Reward。

- 强化学习相对于监督学习为什么训练会更加困难？(强化学习的特征)

答：

1. 强化学习处理的多是序列数据，其很难像监督学习的样本一样满足 IID (独立同分布) 条件。
2. 强化学习有奖励的延迟 (Delay Reward)，即在 Agent 的 action 作用在 Environment 中时，Environment 对于 Agent 的 State 的奖励的延迟 (Delayed Reward)，使得反馈不及时。
3. 相比于监督学习有正确的 label，可以通过其修正自己的预测，强化学习相当于一个“试错”的过程，其完全根据 Environment 的“反馈”更新对自己最有利的 Action。

- 强化学习的基本特征有哪些？

答：

1. 有 trial-and-error exploration 的过程，即需要通过探索 Environment 来获取对这个 Environment 的理解。
2. 强化学习的 Agent 会从 Environment 里面获得延迟的 Reward。
3. 强化学习的训练过程中时间非常重要，因为数据都是有时间关联的，而不是像监督学习一样是 IID 分布的。
4. 强化学习中 Agent 的 Action 会影响它随后得到的反馈。

- 近几年强化学习发展迅速的原因？

答：

1. 算力 (GPU、TPU) 的提升，我们可以更快地做更多的 trial-and-error 的尝试来使得 Agent 在 Environment 里面获得很多信息，取得更大的 Reward。
2. 我们有了深度强化学习这样一个端到端的训练方法，可以把特征提取和价值估计或者决策一起优化，这样就可以得到一个更强的决策网络。

- 状态和观测有什么关系？

答：状态 (state) 是对世界的完整描述，不会隐藏世界的信息。观测 (observation) 是对状态的部分描述，可能会遗漏一些信息。在深度强化学习中，我们几乎总是用一个实值向量、矩阵或者更高阶的张量来表示状态和观测。

- 对于一个强化学习 Agent，它由什么组成？

答：

1. 策略函数 (policy function) , Agent 会用这个函数来选取它下一步的动作, 包括随机性策略 (stochastic policy) 和确定性策略 (deterministic policy)。
 2. 价值函数 (value function), 我们用价值函数来对当前状态进行评估, 即进入现在的状态, 到底可以对你后面的收益带来多大的影响。当这个价值函数大的时候, 说明你进入这个状态越有利。
 3. 模型 (model), 其表示了 Agent 对这个 Environment 的状态进行的理解, 它决定了这个系统是如何进行的。
- 根据强化学习 Agent 的不同, 我们可以将其分为哪几类?
答:
 1. 基于价值函数的 Agent。显式学习的就是价值函数, 隐式的学到了它的策略。因为这个策略是从我们学到的价值函数里面推算出来的。
 2. 基于策略的 Agent。它直接去学习 policy, 就是说你直接给它一个 state, 它就会输出这个动作的概率。然后在这个 policy-based agent 里面并没有去学习它的价值函数。
 3. 然后另外还有一种 Agent 是把这两者结合。把 value-based 和 policy-based 结合起来就有了 Actor-Critic agent。这一类 Agent 就把它的策略函数和价值函数都学到了, 然后通过两者的交互得到一个更佳的状态。
 - 基于策略迭代和基于价值迭代的强化学习方法有什么区别?
答: 1. 基于策略迭代的强化学习方法, agent 会制定一套动作策略 (确定在给定状态下需要采取何种动作), 并根据这个策略进行操作。强化学习算法直接对策略进行优化, 使制定的策略能够获得最大的奖励; 基于价值迭代的强化学习方法, agent 不需要制定显式的策略, 它维护一个价值表格或价值函数, 并通过这个价值表格或价值函数来选取价值最大的动作。
2. 基于价值迭代的方法只能应用在不连续的、离散的环境下 (如围棋或某些游戏领域), 对于行为集合规模庞大、动作连续的场景 (如机器人控制领域), 其很难学习到较好的结果 (此时基于策略迭代的方法能够根据设定的策略来选择连续的动作);
3. 基于价值迭代的强化学习算法有 Q-learning、Sarsa 等, 而基于策略迭代的强化学习算法有策略梯度算法等。
4. 此外, Actor-Critic 算法同时使用策略和价值评估来做出决策, 其中, 智能体会根据策略做出动作, 而价值函数会对做出的动作给出价值, 这样可以在原有的策略梯度算法的基础上加速学习过程, 取得更好的效果。
 - 有模型 (model-based) 学习和免模型 (model-free) 学习有什么区别?
答: 针对是否需要对真实环境建模, 强化学习可以分为有模型学习和免模型学习。有模型学习是指根据环境中的经验, 构建一个虚拟世界, 同时在真实环境和虚拟世界中学习; 免模型学习是指不对环境进行建模, 直接与真实环境进行交互来学习到最优策略。总的来说, 有模型学习相比于免模型学习仅仅多出一个步骤, 即对真实环境进行建模。免模型学习通常属于数据驱动型方法, 需要大量的采样来估计状态、动作及奖励函数, 从而优化动作策略。免模型学习的泛化性要优于有模型学习, 原因是有模型学习算需要对真实环境进行建模, 并且虚拟世界与真实环境之间可能还有差异, 这限制了有模型学习算法的泛化性。
 - 强化学习的通俗理解
答: environment 跟 reward function 不是我们可以控制的, environment 跟 reward function 是在开始学习之前, 就已经事先给定的。我们唯一能做的事情是调整 actor 里面的 policy, 使得 actor 可以得到最大的 reward。Actor 里面会有一个 policy, 这个 policy 决定了 actor 的行为。Policy 就是给一个外界的输入, 然后它会输出 actor 现在应该要执行的行为。

1.10 Something About Interview

- 高冷的面试官：看来你对于 RL 还是具有一定了解的，那么可以用一句话谈一下你对于强化学习的认识吗？
答：强化学习包含环境、动作和奖励三部分，其本质是 agent 通过与环境的交互，使得其作出的 action 所得到的决策得到的总的奖励达到最大，或者说是期望最大。
- 高冷的面试官：你认为强化学习与监督学习和无监督学习有什么区别？
答：首先强化学习和无监督学习是不需要标签的，而监督学习需要许多有标签的样本来进行模型的构建；对于强化学习与无监督学习，无监督学习是直接对于给定的数据进行建模，寻找数据（特征）给定的隐藏的结构，一般对应的聚类问题，而强化学习需要通过延迟奖励学习策略来得到“模型”对于正确目标的远近（通过奖励惩罚函数进行判断），这里我们可以将奖励惩罚函数视为正确目标的一个稀疏、延迟形式。另外强化学习处理的多是序列数据，样本之间通常具有强相关性，但其很难像监督学习的样本一样满足 IID 条件。
- 高冷的面试官：根据你上面介绍的内容，你认为强化学习的使用场景有哪些呢？
答：七个字的话就是多序列决策问题。或者说是对应的模型未知，需要通过学习逐渐逼近真实模型的问题并且当前的动作会影响环境的状态，即服从马尔可夫性的问题。同时应满足所有状态是可重复到达的（满足可学习型的）。
- 高冷的面试官：强化学习中所谓的损失函数与 DL 中的损失函数有什么区别呀？
答：DL 中的 loss function 目的是使预测值和真实值之间的差距最小，而 RL 中的 loss function 是是奖励和的期望最大。
- 高冷的面试官：你了解 model-free 和 model-based 吗？两者有什么区别呢？
答：两者的区别主要在于是否需要对于真实的环境进行建模，model-free 不需要对于环境进行建模，直接与真实环境进行交互即可，所以其通常需要较大的数据或者采样工作来优化策略，这也帮助 model-free 对于真实环境具有更好的泛化性能；而 model-based 需要对于环境进行建模，同时在真实环境与虚拟环境中进行学习，如果建模的环境与真实环境的差异较大，那么会限制其泛化性能。现在通常使用 model-free 进行模型的构建工作。

References

- 百面深度学习
- 强化学习：原理与 Python 实现
- 强化学习基础 David Silver 笔记
- David Silver 强化学习公开课中文讲解及实践
- UCL Course on RL(David Silver)
- 白话强化学习与 PyTorch
- OpenAI Spinning Up
- 神经网络与深度学习
- 机器学习

第 2 章 MDP

本章给大家介绍马尔可夫决策过程。在介绍马尔可夫决策过程之前，先介绍它的简化版本：马尔可夫链以及马尔可夫奖励过程，通过跟这两种过程的比较，我们可以更容易理解马尔可夫决策过程。第二部分会介绍马尔可夫决策过程中的 [policy evaluation](#)，就是当给定一个决策过后，怎么去计算它的价值函数。第三部分会介绍马尔可夫决策过程的控制，具体有两种算法：[policy iteration](#) 和 [value iteration](#)。

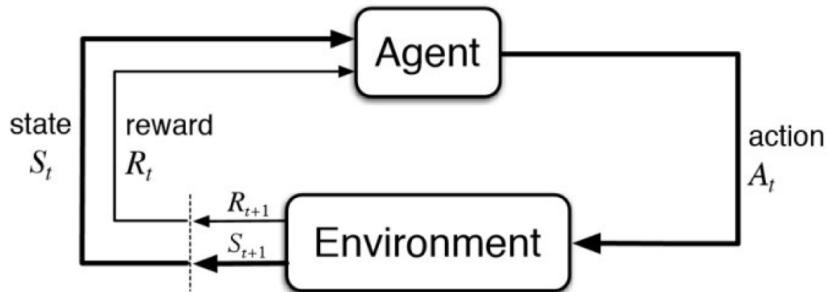


图 2.1

上图介绍了在强化学习里面 agent 跟 environment 之间的交互，agent 在得到环境的状态过后，它会采取动作，它会把这个采取的动作返还给环境。环境在得到 agent 的动作过后，它会进入下一个状态，把下一个状态传回 agent。在强化学习中，agent 跟环境就是这样进行交互的，这个交互过程是可以通过马尔可夫决策过程来表示的，所以马尔可夫决策过程是强化学习里面的一个基本框架。

在马尔可夫决策过程中，它的环境是全部可以观测的 (fully observable)。但是很多时候环境里面有些量是不可观测的，但是这个部分观测的问题也可以转换成一个 MDP 的问题。

在介绍马尔可夫决策过程 (Markov Decision Process, MDP) 之前，先给大家梳理一下马尔可夫过程 (Markov Process, MP)、马尔可夫奖励过程 (Markov Reward Processes, MRP)。这两个过程是马尔可夫决策过程的一个基础。

2.1 Markov Process(MP)

2.1.1 Markov Property

如果一个状态转移是符合马尔可夫的，那就是说一个状态的下一个状态只取决于它当前状态，而跟它当前状态之前的状态都没有关系。

我们设状态的历史为 $h_t = \{s_1, s_2, s_3, \dots, s_t\}$ (h_t 包含了之前的所有状态)，如果一个状态转移是符合马尔可夫的，也就是满足如下条件：

$$\begin{aligned} p(s_{t+1} | s_t) &= p(s_{t+1} | h_t) \\ p(s_{t+1} | s_t, a_t) &= p(s_{t+1} | h_t, a_t) \end{aligned} \tag{2.1}$$

从当前 s_t 转移到 s_{t+1} 这个状态，它是直接就等于它之前所有的状态转移到 s_{t+1} 。如果某一个过程满足[马尔可夫性质 \(Markov Property\)](#)，就是说未来的转移跟过去是独立的，它只取决于现在。马尔可夫性质是所有马尔可夫过程的基础。

2.1.2 Markov Process/Markov Chain

首先看一看[马尔可夫链 \(Markov Chain\)](#)。举个例子，这个图里面有四个状态，这四个状态从 s_1, s_2, s_3, s_4 之间互相转移。比如说从 s_1 开始，

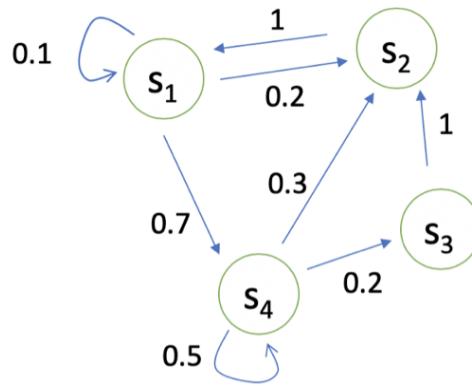


图 2.2

s_1 有 0.1 的概率继续存活在 s_1 状态，有 0.2 的概率转移到 s_2 ，有 0.7 的概率转移到 s_4 。

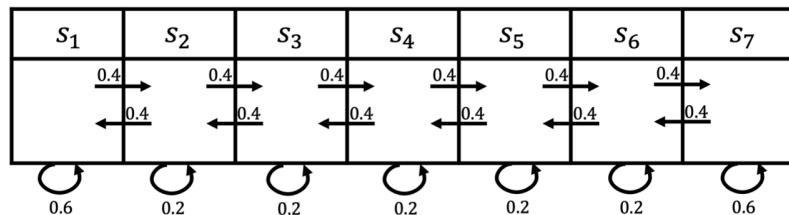
如果 s_4 是我们当前状态的话，它有 0.3 的概率转移到 s_2 ，有 0.2 的概率转移到 s_3 ，有 0.5 的概率留在这里。

我们可以用状态转移矩阵 (State Transition Matrix) P 来描述状态转移 $p(s_{t+1} = s' | s_t = s)$ ，如下式所示。

$$P = \begin{bmatrix} P(s_1 | s_1) & P(s_2 | s_1) & \dots & P(s_N | s_1) \\ P(s_1 | s_2) & P(s_2 | s_2) & \dots & P(s_N | s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_1 | s_N) & P(s_2 | s_N) & \dots & P(s_N | s_N) \end{bmatrix} \quad (2.2)$$

状态转移矩阵类似于一个 conditional probability，当我们知道当前我们在 s_t 这个状态过后，到达下面所有状态的一个概念。所以它每一行其实描述了是从一个节点到达所有其它节点的概率。

2.1.3 Example of MP



① Sample episodes starting from s_3

- ① s_3, s_4, s_5, s_6, s_7
- ② s_3, s_2, s_3, s_2, s_1
- ③ s_3, s_4, s_4, s_5, s_5

图 2.3

上图是一个马尔可夫链的例子，我们这里有七个状态。比如说从 s_1 开始到 s_2 ，它有 0.4 的概率，然后它有 0.6 的概率继续存活在它当前的状态。 s_2 有 0.4 的概率到左边，有 0.4 的概率到 s_3 ，另外有 0.2 的

概率存活在现在的状态，所以给定了这个状态转移的马尔可夫链后，我们可以对这个链进行采样，这样就会得到一串的轨迹。

下面我们有三个轨迹，都是从同一个起始点开始。假设还是从 s_3 这个状态开始，

- 第一条链先到了 s_4 ，又到了 s_5 ，又往右到了 s_6 ，然后继续存活在 s_6 状态。
- 第二条链从 s_3 开始，先往左走到了 s_2 。然后它又往右走，又回到了 s_3 ，然后它又往左走，然后再往左走到了 s_1 。
- 通过对这个状态的采样，我们生成了很多这样的轨迹。

2.2 Markov Reward Process(MRP)

马尔可夫奖励过程 (Markov Reward Process, MRP) 是马尔可夫链再加上了一个奖励函数。在 MRP 中，转移矩阵和状态都是跟马尔可夫链一样的，多了一个奖励函数 (reward function)。奖励函数 R 是一个期望，就是说当你到达某一个状态的时候，可以获得多大的奖励。然后这里另外定义了一个 discount factor γ 。如果状态数是有限的， R 可以是一个向量。

2.2.1 Example of MRP

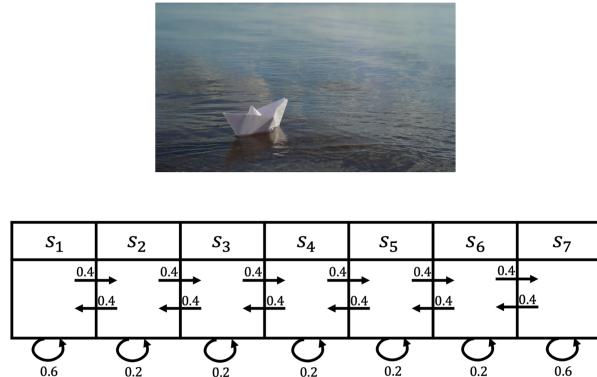


图 2.4

这里是我们刚才看的马尔可夫链，如果说把奖励也放上去的话，就是说到达每一个状态，我们都会获得一个奖励。这里我们可以设置对应的奖励，比如说到达 s_1 状态的时候，可以获得 5 的奖励，到达 s_7 的时候，可以得到 10 的奖励，其它状态没有任何奖励。因为这里状态是有限的，所以我们可以用向量 $R = [5, 0, 0, 0, 0, 0, 10]$ 来表示这个奖励函数，这个向量表示了每个点的奖励大小。

我们通过一个形象的例子来理解 MRP。我们把一个纸船放到河流之中，那么它就会随着这个河流而流动，它自身是没有动力的。所以你可以把 MRP 看成是一个随波逐流的例子，当我们从某一个点开始的时候，这个纸船就会随着事先定义好的状态转移进行流动，它到达每个状态过后，我们就有可能获得一些奖励。

2.2.2 Return and Value function

这里我们进一步定义一些概念。

- Horizon 是指一个回合的长度（每个回合最大的时间步数），它是由有限个步数决定的。
- Return(回报) 说的是把奖励进行折扣后所获得的收益。Return 可以定义为奖励的逐步叠加，如下所示：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots + \gamma^{T-t-1} R_T \quad (2.3)$$

这里有一个叠加系数，越往后得到的奖励，折扣得越多。这说明我们其实更希望得到现有的奖励，未来的奖励就要把它打折扣。

- 当我们有了 return 过后，就可以定义一个状态的价值了，就是 state value function。对于 MRP，state value function 被定义成是 return 的期望，如下式所示：

$$\begin{aligned} V_t(s) &= \mathbb{E}[G_t | s_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T | s_t = s] \end{aligned} \quad (2.4)$$

G_t 是之前定义的 discounted return，我们这里取了一个期望，期望就是说从这个状态开始，你有可能获得多大的价值。所以这个期望也可以看成是对未来可能获得奖励的当前价值的一个表现，就是当你进入某一个状态过后，你现在就有多少大的价值。

2.2.3 Why Discount Factor

这里解释一下为什么需要 discount factor。

- 有些马尔可夫过程是带环的，它并没有终结，我们想避免这个无穷的奖励。
- 我们并没有建立一个完美的模拟环境的模型，也就是说，我们对未来的评估不一定是准确的，我们不一定完全信任我们的模型，因为这种不确定性，所以我们对未来的评估增加一个折扣。我们想把这个不确定性表示出来，希望尽可能快地得到奖励，而不是在未来某一个点得到奖励。
- 如果这个奖励是有实际价值的，我们可能是更希望立刻就得到奖励，而不是后面再得到奖励（现在的钱比以后的钱更有价值）。
- 在人的行为里面来说的话，大家也是想得到即时奖励。
- 有些时候可以把这个系数设为 0， $\gamma = 0$ ：我们就只关注了它当前的奖励。我们也可以把它设为 1， $\gamma = 1$ ：对未来并没有折扣，未来获得的奖励跟当前获得的奖励是一样的。

Discount factor 可以作为强化学习 agent 的一个超参数来进行调整，然后就会得到不同行为的 agent。

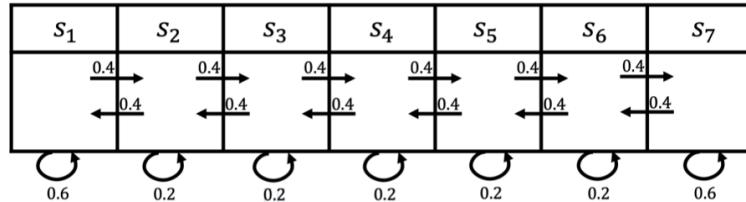


图 2.5 MRP 的例子

这里我们再来看一看，在这个 MRP 里面，如何计算它的价值。这个 MRP 依旧是这个状态转移。它的奖励函数是定义成这样，它在进入第一个状态 s_1 的时候会得到 5 的奖励，进入第七个状态 s_7 的时候会得到 10 的奖励，其它状态都没有奖励。因此，我们可以用表示 $R = [5, 0, 0, 0, 0, 0, 10]$ 。

我们对 4 步的回合 ($\gamma = 1/2$) 来采样回报 G :

- s_4, s_5, s_6, s_7 的回报： $0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 10 = 1.25$
- s_4, s_3, s_2, s_1 的回报： $0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 5 = 0.625$
- s_4, s_5, s_6, s_6 的回报 := 0

我们现在可以计算每一个轨迹得到的奖励，比如我们对于这个 s_4, s_5, s_6, s_7 轨迹的奖励进行计算，这里折扣系数是 0.5。

- 在 s_4 的时候，奖励为零。
- 下一个状态 s_5 的时候，因为我们已经到了下一步了，所以我们要把 s_5 进行一个折扣， s_5 本身也是没有奖励的。

- 然后是到 s_6 , 也没有任何奖励, 折扣系数应该是 $\frac{1}{4}$ 。
- 到达 s_7 后, 我们获得了一个奖励, 但是因为 s_7 这个状态是未来才获得的奖励, 所以我们要进行三次折扣。

所以对于这个轨迹, 它的 return 就是一个 1.25, 类似地, 我们可以得到其它轨迹的 return。

这里就引出了一个问题, 当我们有了一些轨迹的实际 return, 怎么计算它的价值函数。比如说我们想知道 s_4 状态的价值, 就是当你进入 s_4 后, 它的价值到底如何。一个可行的做法就是说我们可以产生很多轨迹, 然后把这里的轨迹都叠加起来。比如我们可以从 s_4 开始, 采样生成很多轨迹, 都把它的 return 计算出来, 然后可以直接把它取一个平均作为你进入 s_4 它的价值。这其实是一种计算价值函数的办法, 通过这个蒙特卡罗采样的办法计算 s_4 的状态。接下来会进一步介绍蒙特卡罗算法。

2.2.4 Bellman Equation

但是这里我们采取了另外一种计算方法, 我们从这个价值函数里面推导出 [Bellman Equation \(贝尔曼等式\)](#), 如下式所示:

$$V(s) = \underbrace{R(s)}_{\text{Immediate reward}} + \underbrace{\gamma \sum_{s' \in S} P(s' | s) V(s')}_{\text{Discounted sum of future reward}} \quad (2.5)$$

其中:

- s' 可以看成未来的所有状态。
- 转移 $P(s'|s)$ 是指从当前状态转移到未来状态的概率。
- $V(s')$ 代表的是未来某一个状态的价值。我们从当前这个位置开始, 有一定的概率去到未来的所有状态, 所以我们要把这个概率也写上去, 这个转移矩阵也写上去, 然后我们就得到了未来状态, 然后再乘以一个 γ , 这样就可以把未来的奖励打折扣。
- 第二部分可以看成是未来奖励的折扣总和 (Discounted sum of future reward)。

Bellman Equation 定义了当前状态跟未来状态之间的这个关系。

未来打了折扣的奖励加上当前立刻可以得到的奖励, 就组成了这个 Bellman Equation。

Law of Total Expectation

在推导 Bellman equation 之前, 我们先使用 [Law of Total Expectation\(全期望公式\)](#) 来证明下面的式子:

$$\mathbb{E}[V(s_{t+1})|s_t] = \mathbb{E}[\mathbb{E}[G_{t+1}|s_{t+1}]|s_t] = E[G_{t+1}|s_t] \quad (2.6)$$

Law of total expectation 也被称为 law of iterated expectations(LIE)。

如果 A_i 是样本空间的有限或可数的划分 (partition), 则全期望公式可以写成如下形式:

$$\mathbb{E}(X) = \sum_i \mathbb{E}(X | A_i) P(A_i) \quad (2.7)$$

证明:

为了记号简洁并且易读, 我们丢掉了下标, 令 $s = s_t, g' = G_{t+1}, s' = s_{t+1}$ 。按照惯例, 我们可以重写这个回报的期望为:

$$\begin{aligned} \mathbb{E}[G_{t+1} | s_{t+1}] &= \mathbb{E}[g' | s'] \\ &= \sum_{g'} g' p(g' | s') \end{aligned} \quad (2.8)$$

令 $s_t = s$, 我们对上述表达式求期望可得:

$$\begin{aligned}
 \mathbb{E}[\mathbb{E}[G_{t+1} | s_{t+1}] | s_t] &= \mathbb{E}[\mathbb{E}[g' | s'] | s] \\
 &= \sum_{s'} \sum_{g'} g' p(g' | s', s) p(s' | s) \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g' | s', s) p(s' | s) p(s)}{p(s)} \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g' | s', s) p(s', s)}{p(s)} \\
 &= \sum_{s'} \sum_{g'} \frac{g' p(g', s' | s)}{p(s)} \\
 &= \sum_{g'} \sum_{s'} g' p(g', s' | s) \\
 &= \sum_{g'} g' p(g' | s) \\
 &= \mathbb{E}[g' | s] = \mathbb{E}[G_{t+1} | s_t]
 \end{aligned} \tag{2.9}$$

Bellman Equation Derivation

Bellman equation 的推导过程如下:

$$\begin{aligned}
 V(s) &= \mathbb{E}[G_t | s_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t = s] \\
 &= \mathbb{E}[R_{t+1} | s_t = s] + \gamma \mathbb{E}[R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots | s_t = s] \\
 &= R(s) + \gamma \mathbb{E}[G_{t+1} | s_t = s] \\
 &= R(s) + \gamma \mathbb{E}[V(s_{t+1}) | s_t = s] \\
 &= R(s) + \gamma \sum_{s' \in S} P(s' | s) V(s')
 \end{aligned} \tag{2.10}$$

Bellman Equation 就是当前状态与未来状态的迭代关系, 表示当前状态的值函数可以通过下个状态的值函数来计算。Bellman Equation 因其提出者、动态规划创始人 Richard Bellman 而得名, 也叫作“动态规划方程”。

Bellman Equation 定义了状态之间的迭代关系, 如下式所示。

$$V(s) = R(s) + \gamma \sum_{s' \in S} P(s' | s) V(s') \tag{2.11}$$

假设有一个马尔可夫转移矩阵是右边这个样子, Bellman Equation 描述的就是当前状态到未来状态的一个转移。假设我们当前是在 s_1 , 那么它只可能去到三个未来的状态: 有 0.1 的概率留在它当前这个位置, 有 0.2 的概率去到 s_2 状态, 有 0.7 的概率去到 s_4 的状态, 所以我们要把这个转移乘以它未来的状态的价值, 再加上它的 immediate reward 就会得到它当前状态的价值。所以 Bellman Equation 定义的就是当前状态跟未来状态的一个迭代的关系。

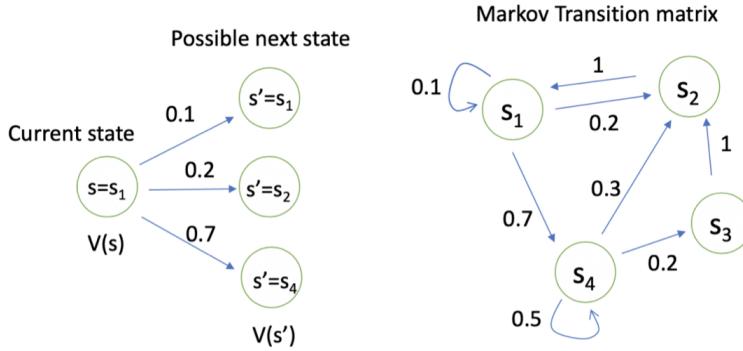


图 2.6

我们可以把 Bellman Equation 写成一种矩阵的形式，如下式所示。

$$\begin{bmatrix} V(s_1) \\ V(s_2) \\ \vdots \\ V(s_N) \end{bmatrix} = \begin{bmatrix} R(s_1) \\ R(s_2) \\ \vdots \\ R(s_N) \end{bmatrix} + \gamma \begin{bmatrix} P(s_1 | s_1) & P(s_2 | s_1) & \dots & P(s_N | s_1) \\ P(s_1 | s_2) & P(s_2 | s_2) & \dots & P(s_N | s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_1 | s_N) & P(s_2 | s_N) & \dots & P(s_N | s_N) \end{bmatrix} \begin{bmatrix} V(s_1) \\ V(s_2) \\ \vdots \\ V(s_N) \end{bmatrix} \quad (2.12)$$

首先有这个转移矩阵。我们当前这个状态是一个向量 $[V(s_1), V(s_2), \dots, V(s_N)]^T$ 。我们可以写成迭代的形式。我们每一行来看的话， V 这个向量乘以了转移矩阵里面的某一行，再加上它当前可以得到的 reward，就会得到它当前的价值。

当我们把 Bellman Equation 写成矩阵形式后，可以直接求解：

$$\begin{aligned} V &= R + \gamma PV \\ IV &= R + \gamma PV \\ (I - \gamma P)V &= R \\ V &= (I - \gamma P)^{-1}R \end{aligned} \quad (2.13)$$

我们可以直接得到一个解析解 (analytic solution)：

$$V = (I - \gamma P)^{-1}R \quad (2.14)$$

我们可以通过矩阵求逆把这个 V 的这个价值直接求出来。但是一个问题就是这个矩阵求逆的过程的复杂度是 $O(N^3)$ 。所以当状态非常多的时候，比如说从十个状态到一千个状态，到一百万个状态。那么当我们有一百万个状态的时候，这个转移矩阵就会是个一千万乘以一千万的矩阵，这样一个大矩阵的话求逆是非常困难的，所以这种通过解析解去求解的方法只适用于很小量的 MRP。

2.2.5 Iterative Algorithm for Computing Value of a MRP

接下来我们来求解这个价值函数。我们可以通过迭代的方法来解这种状态非常多的 MRP(large MRPs)，比如说：动态规划的方法，蒙特卡罗的办法(通过采样的办法去计算它)，时序差分学习(Temporal-Difference Learning)的办法。Temporal-Difference Learning 叫 TD Learning，它是动态规划和蒙特卡罗的一个结合。

首先我们用蒙特卡罗 (Monte Carlo) 的办法来计算它的价值函数。蒙特卡罗就是说当得到一个 MRP 过后，我们可以从某一个状态开始，把这个小船放进去，让它随波逐流，这样就会产生一个轨迹。产生了一个轨迹过后，就会得到一个奖励，那么就直接把它的折扣的奖励 g 算出来。算出来过后就可以把它积累起来，得到 return G_t 。当积累到一定的轨迹数量过后，直接用 G_t 除以轨迹数量，就会得到它的价值。

Monte Carlo Algorithm for Computing Value of a MRP

Algorithm 1 Monte Carlo simulation to calculate MRP value function

```

1:  $i \leftarrow 0, G_t \leftarrow 0$ 
2: while  $i \neq N$  do
3:   generate an episode, starting from state  $s$  and time  $t$ 
4:   Using the generated episode, calculate return  $g = \sum_{i=t}^{H-1} \gamma^{i-t} r_i$ 
5:    $G_t \leftarrow G_t + g, i \leftarrow i + 1$ 
6: end while
7:  $V_t(s) \leftarrow G_t/N$ 
  
```

- ① For example: to calculate $V(s_4)$ we can generate a lot of trajectories then take the average of the returns:

- ① return for $s_4, s_5, s_6, s_7 : 0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 10 = 1.25$
- ② return for $s_4, s_3, s_2, s_1 : 0 + \frac{1}{2} \times 0 + \frac{1}{4} \times 0 + \frac{1}{8} \times 5 = 0.625$
- ③ return $s_4, s_5, s_6, s_6 : = 0$
- ④ more trajectories

图 2.7

比如说我们要算 s_4 状态的价值。我们就可以从 s_4 状态开始，随机产生很多轨迹，就是说产生很多小船，把小船扔到这个转移矩阵里面去，然后它就会随波逐流，产生轨迹。每个轨迹都会得到一个 return，我们得到大量的 return，比如说一百个、一千个 return，然后直接取一个平均，那么就可以等价于现在 s_4 这个价值，因为 s_4 的价值 $V(s_4)$ 定义了你未来可能得到多少的奖励。这就是蒙特卡罗采样的方法。

Algorithm 2 Iterative algorithm to calculate MRP value function

```

1: for all states  $s \in S, V'(s) \leftarrow 0, V(s) \leftarrow \infty$ 
2: while  $\|V - V'\| > \epsilon$  do
3:    $V \leftarrow V'$ 
4:   For all states  $s \in S, V'(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s) V(s')$ 
5: end while
6: return  $V'(s)$  for all  $s \in S$ 
  
```

图 2.8 计算 MRP 值的迭代算法

我们也可以用这个动态规划的办法，一直去迭代它的 Bellman equation，让它最后收敛，我们就可以得到它的一个状态。所以在这里算法二就是一个迭代的算法，通过 bootstrapping(自举) 的办法，然后去不停地迭代这个 Bellman Equation。当这个最后更新的状态跟你上一个状态变化并不大的时候，更新就可以停止，我们就可以输出最新的 $V'(s)$ 作为它当前的状态。所以这里就是把 Bellman Equation 变成一个 Bellman Update，这样就可以得到它的一个价值。

动态规划的方法基于后继状态值的估计来更新状态值的估计（算法二中的第 3 行用 V' 来更新 V ）。也就是说，它们根据其他估算值来更新估算值。我们称这种基本思想为 bootstrapping。

Bootstrap 本意是“解靴带”；这里是在使用德国文学作品《吹牛大王历险记》中解靴带自助（拔靴自助）的典故，因此将其译为“自举”。

2.3 Markov Decision Process(MDP)

2.3.1 MDP

相对于 MRP，[马尔可夫决策过程 \(Markov Decision Process\)](#)多了一个 decision，其它的定义跟 MRP 都是类似的：

- 这里多了一个决策，多了一个动作。
- 状态转移也多了一个条件，变成了 $P(s_{t+1} = s' | s_t = s, a_t = a)$ 。你采取某一种动作，然后你未来的状态会不同。未来的状态不仅是依赖于你当前的状态，也依赖于在当前状态 agent 采取的这个动作。
- 对于这个价值函数，它也是多了一个条件，多了一个你当前的这个动作，变成了 $R(s_t = s, a_t = a) = \mathbb{E}[r_t | s_t = s, a_t = a]$ 。你当前的状态以及你采取的动作会决定你在当前可能得到的奖励多少。

2.3.2 Policy in MDP

Policy 定义了在某一个状态应该采取什么样的动作。知道当前状态过后，我们可以把当前状态带入 policy function，然后就会得到一个概率，即

$$\pi(a | s) = P(a_t = a | s_t = s) \quad (2.15)$$

概率就代表了在所有可能的动作里面怎样采取行动，比如可能有 0.7 的概率往左走，有 0.3 的概率往右走，这是一个概率的表示。

另外这个策略也可能是确定的，它有可能是直接输出一个值。或者就直接告诉你当前应该采取什么样的动作，而不是一个动作的概率。

假设这个概率函数应该是稳定的 (stationary)，不同时间点，你采取的动作其实都是对这个 policy function 进行采样。

我们可以将 MRP 转换成 MDP。已知一个 MDP 和一个 policy π 的时候，我们可以把 MDP 转换成 MRP。

在 MDP 里面，转移函数 $P(s'|s, a)$ 是基于它当前状态以及它当前的 action。因为我们现在已知它 policy function，就是说在每一个状态，我们知道它可能采取的动作的概率，那么就可以直接把这个 action 进行加和，直接把这个 a 去掉，那我们就可以得到对于 MRP 的一个转移，这里就没有 action。

$$P^\pi(s' | s) = \sum_{a \in A} \pi(a | s) P(s' | s, a) \quad (2.16)$$

对于这个奖励函数，我们也可以把 action 拿掉，这样就会得到一个类似于 MRP 的奖励函数。

$$R^\pi(s) = \sum_{a \in A} \pi(a | s) R(s, a) \quad (2.17)$$

2.3.3 Comparison of MP/MRP and MDP

这里我们看一看，MDP 里面的状态转移跟 MRP 以及 MP 的一个差异。

- 马尔可夫过程的转移是直接就决定。比如当前状态是 s ，那么就直接通过这个转移概率决定了下一个状态是什么。
- 但对于 MDP，它的中间多了一层这个动作 a ，就是说在你当前这个状态的时候，首先要决定的是采取某一种动作，那么你会到了某一个黑色的节点。到了这个黑色的节点，因为你有一定的不确定性，当你当前状态决定过后以及你当前采取的动作过后，你到未来的状态其实也是一个概率分布。所以在这个当前状态跟未来状态转移过程中这里多了一层决策性，这是 MDP 跟之前的马尔可夫过程很不同的一个地方。在马尔可夫决策过程中，动作是由 agent 决定，所以多了一个 component，agent 会采取动作来决定未来的状态转移。

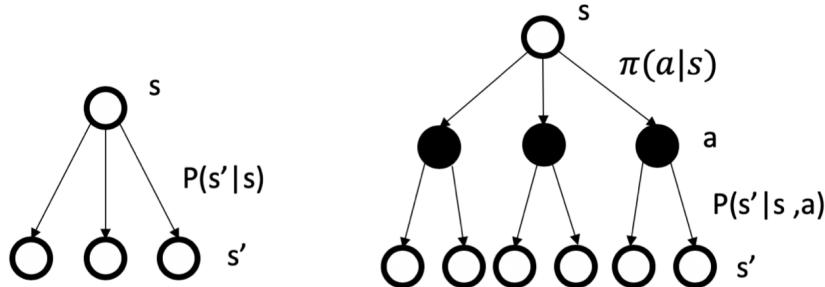


图 2.9

2.3.4 Value function for MDP

顺着 MDP 的定义，我们可以把状态-价值函数 (state-value function)，就是在 MDP 里面的价值函数也进行一个定义，它的定义是跟 MRP 是类似的，如式 (2.18) 所示

$$v^\pi(s) = \mathbb{E}_\pi [G_t \mid s_t = s] \quad (2.18)$$

但是这里 expectation over policy，就是这个期望是基于你采取的这个 policy，就当你的 policy 决定过后，我们通过对这个 policy 进行采样来得到一个期望，那么就可以计算出它的这个价值函数。

这里我们另外引入了一个Q 函数 (Q-function)。Q 函数也被称为action-value function。Q 函数定义的是在某一个状态采取某一个动作，它有可能得到的这个 return 的一个期望，如式 (2.19) 所示：

$$q^\pi(s, a) = \mathbb{E}_\pi [G_t \mid s_t = s, A_t = a] \quad (2.19)$$

这里期望其实也是 over policy function。所以你需要对这个 policy function 进行一个加和，然后得到它的这个价值。对 Q 函数中的动作函数进行加和，就可以得到价值函数，如式 (2.20) 所示：

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s) q^\pi(s, a) \quad (2.20)$$

Bellman Equation Derivation

此处我们给出 Q 函数的 Bellman equation：

$$\begin{aligned} q(s, a) &= \mathbb{E}[G_t \mid s_t = s, a_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid s_t = s, a_t = a] \\ &= \mathbb{E}[R_{t+1} \mid s_t = s, a_t = a] + \gamma \mathbb{E}[R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots \mid s_t = s, a_t = a] \\ &= R(s, a) + \gamma \mathbb{E}[G_{t+1} \mid s_t = s, a_t = a] \\ &= R(s, a) + \gamma \mathbb{E}[V(s_{t+1}) \mid s_t = s, a_t = a] \\ &= R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V(s') \end{aligned} \quad (2.21)$$

2.3.5 Bellman Expectation Equation

我们可以把状态-价值函数和 Q 函数拆解成两个部分：即时奖励 (immediate reward) 和后续状态的折扣价值 (discounted value of successor state)。

通过对状态-价值函数进行一个分解，我们就可以得到一个类似于之前 MRP 的 Bellman Equation，这里叫 Bellman Expectation Equation，如式 (2.22) 所示：

$$v^\pi(s) = E_\pi [R_{t+1} + \gamma v^\pi(s_{t+1}) \mid s_t = s] \quad (2.22)$$

对于 Q 函数，我们也可以做类似的分解，也可以得到 Q 函数的 Bellman Expectation Equation，如式 (2.23) 所示：

$$q^\pi(s, a) = E_\pi [R_{t+1} + \gamma q^\pi(s_{t+1}, A_{t+1}) \mid s_t = s, A_t = a] \quad (2.23)$$

Bellman expectation equation 定义了你当前状态跟未来状态之间的一个关联。

我们进一步进行一个简单的分解。

我们先给出式 (2.24)：

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s) q^\pi(s, a) \quad (2.24)$$

再给出式 (2.25)：

$$q^\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P(s' \mid s, a) v^\pi(s') \quad (2.25)$$

式 (2.24) 和式 (2.25) 代表了价值函数跟 Q 函数之间的一个关联。

也可以把式 (2.25) 插入式 (2.24) 中，得到式 (2.26)：

$$v^\pi(s) = \sum_{a \in A} \pi(a \mid s) \left(R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) v^\pi(s') \right) \quad (2.26)$$

式 (2.26) 代表了当前状态的价值跟未来状态价值之间的一个关联。

我们把式 (2.24) 插入到式 (2.25)，就可以得到式 (2.27)：

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) \sum_{a' \in A} \pi(a' \mid s') q^\pi(s', a') \quad (2.27)$$

式 (2.27) 代表了当前时刻的 Q 函数跟未来时刻的 Q 函数之间的一个关联。

式 (2.26) 和式 (2.27) 是 Bellman expectation equation 的另一种形式。

2.3.6 Backup Diagram

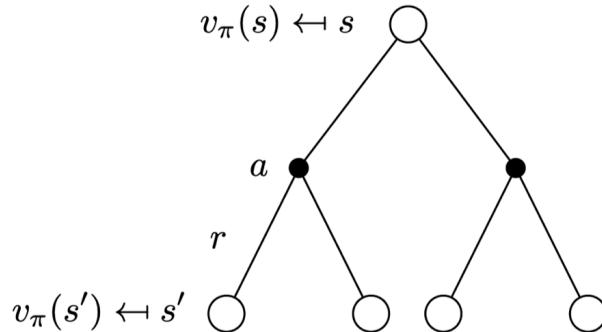


图 2.10 Backup Diagram for V^π

这里有一个概念叫 **backup**。Backup 类似于 bootstrapping 之间这个迭代关系，就对于某一个状态，它的当前价值是跟它的未来价值线性相关的。

我们把上面这样的图称为 **backup diagram(备份图)**，因为它们图示的关系构成了更新或备份操作的基础，而这些操作是强化学习方法的核心。这些操作将价值信息从一个状态（或状态-动作对）的后继状态（或状态-动作对）转移回它。

每一个空心圆圈代表一个状态，每一个实心圆圈代表一个状态-动作对。

$$v^\pi(s) = \sum_{a \in A} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^\pi(s') \right) \quad (2.28)$$

如式 (2.28) 所示，我们这里有两层加和：

- 第一层加和就是这个叶子节点，往上走一层的话，我们就可以把未来的价值 (s' 的价值) backup 到黑色的节点。
- 第二层加和是对 action 进行加和。得到黑色节点的价值过后，再往上 backup 一层，就会推到根节点的价值，即当前状态的价值。

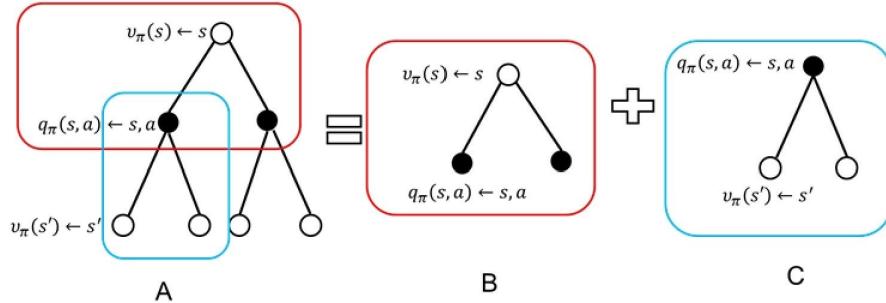


图 2.11

上图是状态-价值函数的计算分解图，上图 B 计算公式为

$$v^\pi(s) = \sum_{a \in A} \pi(a | s) q^\pi(s, a) \quad (2.29)$$

上图 B 给出了状态-价值函数与 Q 函数之间的关系。上图 C 计算 Q 函数为

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^\pi(s') \quad (2.30)$$

将式 (2.30) 代入式 (2.29) 可得：

$$v^\pi(s) = \sum_{a \in A} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^\pi(s') \right) \quad (2.31)$$

所以 backup diagram 定义了未来下一时刻的状态-价值函数跟上一时刻的状态-价值函数之间的关联。

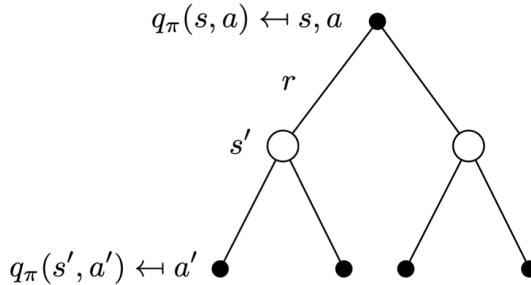


图 2.12 Backup Diagram for Q^π

对于 Q 函数，我们也可以进行这样的一个推导。现在的根节点是这个 Q 函数的一个节点。Q 函数对应于黑色的节点。我们下一时刻的 Q 函数是叶子节点，有四个黑色节点。

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) \sum_{a' \in A} \pi(a' | s') q^\pi(s', a') \quad (2.32)$$

如式 (2.32) 所示，我们这里也有两个加和：

- 第一层加和是先把这个叶子节点从黑色节点推到这个白色的节点，进了它的这个状态。
- 当我们到达某一个状态过后，再对这个白色节点进行一个加和，这样就把它重新推回到当前时刻的一个 Q 函数。

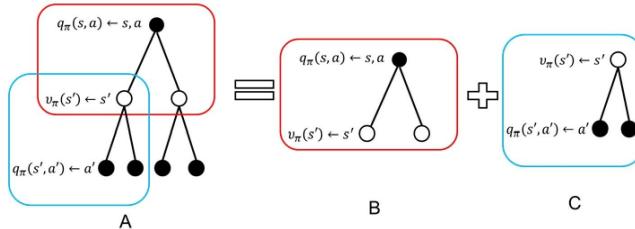


图 2.13

在上图 C 中，

$$v^\pi(s') = \sum_{a' \in A} \pi(a' | s') q^\pi(s', a') \quad (2.33)$$

将式 (2.33) 代入式 (2.30) 可得到 Q 函数：

$$q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) \sum_{a' \in A} \pi(a' | s') q^\pi(s', a') \quad (2.34)$$

所以这个等式就决定了未来 Q 函数跟当前 Q 函数之间的这个关联。

2.3.7 Policy Evaluation(Prediction)

当我们知道一个 MDP 以及要采取的策略 π ，计算价值函数 $v^\pi(s)$ 的过程就是 policy evaluation。就像我们在评估这个策略，我们会得到多大的奖励。Policy evaluation 在有些地方也被叫做 (value) prediction，也就是预测你当前采取的这个策略最终会产生多少的价值。



Figure: Markov Chain/MRP: Go with river stream



Figure: MDP: Navigate the boat

图 2.14 例子：驾驶船

- MDP，你其实可以把它想象成一个摆渡的人在这个船上面，她可以控制这个船的移动，这样就避免了这个船随波逐流。因为在每一个时刻，这个人会决定采取什么样的一个动作，这样会把这个船进行导向。

- MRP 跟 MP 的话，这个纸的小船会随波逐流，然后产生轨迹。
- MDP 的不同就是有一个 agent 去控制这个船，这样我们就可以尽可能多地获得奖励。

Example: Policy Evaluation

s_1	s_2	s_3	s_4	s_5	s_6	s_7

- ① Two actions: *Left* and *Right*
- ② For all actions, reward: +5 in s_1 , +10 in s_7 , 0 in all other states. So that we can represent $R = [5, 0, 0, 0, 0, 0, 10]$
- ③ Let's have a deterministic policy $\pi(s) = \text{Left}$ and $\gamma = 0$ for any state s , then what is the value of the policy?
 ① $V^\pi = [5, 0, 0, 0, 0, 0, 10]$ since $\gamma = 0$

图 2.15

我们再看下 policy evaluation 的例子，怎么在决策过程里面计算它每一个状态的价值。

- 假设环境里面有两种动作：往左走和往右走。
- 现在的奖励函数应该是关于动作以及状态两个变量的一个函数。但我们这里规定，不管你采取什么动作，只要到达状态 s_1 ，就有 5 的奖励。只要你到达状态 s_7 了，就有 10 的奖励，中间没有任何奖励。
- 假设我们现在采取的一个策略，这个策略是说不管在任何状态，我们采取的策略都是往左走。假设价值折扣因子是零，那么对于确定性策略 (deterministic policy)，最后估算出的价值函数是一致的，即 $V^\pi = [5, 0, 0, 0, 0, 0, 10]$

Q: 怎么得到这个结果？

A: 我们可以直接在去 run 下面这个 iterative equation：

$$v_k^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in S} P(s' | s, \pi(s)) v_{k-1}^\pi(s') \quad (2.35)$$

就把 Bellman expectation equation 拿到这边来，然后不停地迭代，最后它会收敛。收敛过后，它的值就是它每一个状态的价值。

再来看一个例子 (practice 1)，如果折扣因子是 0.5，我们可以通过下面这个等式进行迭代：

$$v_t^\pi(s) = \sum_a P(\pi(s) = a) \left(r(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v_{t-1}^\pi(s') \right) \quad (2.36)$$

然后就会得到它的状态价值。

另外一个例子 (practice 2)，就是说我们现在采取的 policy 在每个状态下，有 0.5 的概率往左走，有 0.5 的概率往右走，那么放到这个状态里面去如何计算。其实也是把这个 Bellman expectation equation 拿出来，然后进行迭代就可以算出来了。一开始的时候，我们可以初始化，不同的 $v(s')$ 都会有一个值，放到 Bellman expectation equation 里面去迭代，然后就可以算出它的状态价值。

2.3.8 Prediction and Control

MDP 的 prediction 和 control 是 MDP 里面的核心问题。

s_1	s_2	s_3	s_4	s_5	s_6	s_7
						

- ① $R = [5, 0, 0, 0, 0, 0, 10]$
- ② Practice 1: Deterministic policy $\pi(s) = \text{Left}$ with $\gamma = 0.5$ for any state s , then what are the state values under the policy?
- ③ Practice 2: Stochastic policy $P(\pi(s) = \text{Left}) = 0.5$ and $P(\pi(s) = \text{Right}) = 0.5$ and $\gamma = 0.5$ for any state s , then what are the state values under the policy?
- ④ Iteration t:
 $v_t^\pi(s) = \sum_a P(\pi(s) = a)(r(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)v_{t-1}^\pi(s'))$

图 2.16 例子：策略评估

预测（评估一个给定的策略）：

- 输入：MDP $< S, A, P, R, \gamma >$ 和 policy π 或者 MRP $< S, P^\pi, R^\pi, \gamma >$ 。
- 输出：value function v^π 。
- Prediction 是说给定一个 MDP 以及一个 policy π ，去计算它的 value function，就对于每个状态，它的价值函数是多少。

控制（搜索最佳策略）：

- 输入：MDP $< S, A, P, R, \gamma >$ 。
- 输出：最佳价值函数 (optimal value function) v^* 和最佳策略 (optimal policy) π^* 。
- Control 就是说我们去寻找一个最佳的策略，然后同时输出它的最佳价值函数以及最佳策略。

在 MDP 里面，prediction 和 control 都可以通过动态规划去解决。要强调的是，这两者的区别就在于，预测问题是给定一个 policy，我们要确定它的 value function 是多少。而控制问题是在没有 policy 的前提下，我们要确定最优的 value function 以及对应的决策方案。实际上，这两者是递进的关系，在强化学习中，我们通过解决预测问题，进而解决控制问题。

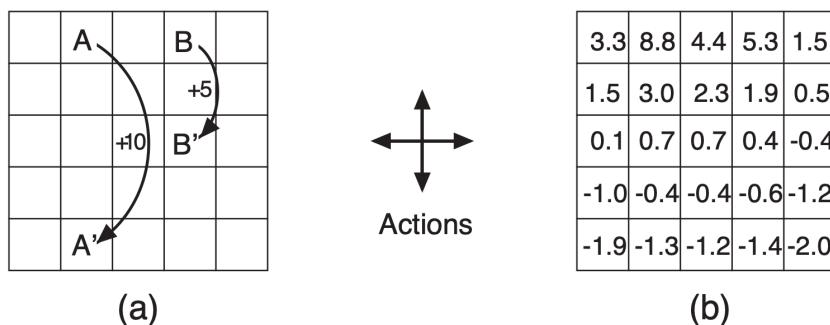


图 2.17 GridWorld Example: Prediction

举一个例子来说明 prediction 与 control 的区别。

首先是预测问题：

- 在上图的方格中，我们规定从 $A \rightarrow A'$ 可以得到 +10 的奖励，从 $B \rightarrow B'$ 可以得到 +5 的奖励，其

它步骤的奖励为 -1。

- 现在，我们给定一个 policy：在任何状态中，它的行为模式都是随机的，也就是上下左右的概率各 25%。
- 预测问题要做的就是，在这种决策模式下，我们的 value function 是什么。上图 b 是对应的 value function。

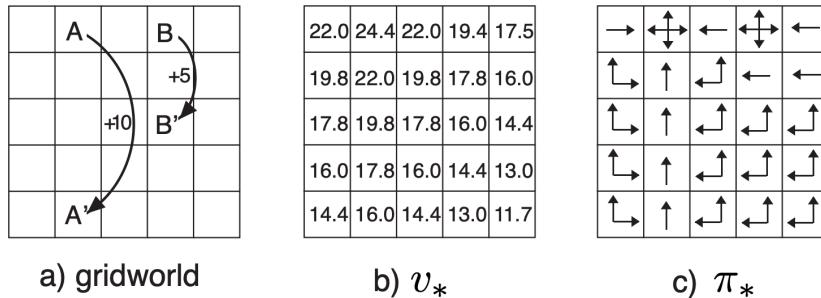


图 2.18 GridWorld Example: Control

接着是控制问题：

在控制问题中，问题背景与预测问题相同，唯一的区别就是：不再限制 policy。也就是说行为模式是未知的，我们要自己确定。所以我们通过解决控制问题，求得每一个状态的最优的 value function（如上图 b 所示），也得到了最优的 policy（如上图 c 所示）。

控制问题要做的就是，给定同样的条件，在所有可能的策略下最优的价值函数是什么？最优策略是什么？

2.3.9 Dynamic Programming

动态规划 (Dynamic Programming, DP) 适合解决满足如下两个性质的问题：

最优子结构 (optimal substructure)。最优子结构意味着，我们的问题可以拆分成一个个的小问题，通过解决这个小问题，最后，我们能够通过组合小问题的答案，得到大问题的答案，即最优的解。**重叠子问题 (Overlapping subproblems)**。重叠子问题意味着，子问题出现多次，并且子问题的解决方案能够被重复使用。

MDP 是满足动态规划的要求的，

在 Bellman equation 里面，我们可以把它分解成一个递归的结构。当我们把它分解成一个递归的结构的时候，如果我们的子问题子状态能得到一个值，那么它的未来状态因为跟子状态是直接相连的，那我们也可以继续推算出来。价值函数就可以储存并重用它的最佳的解。

动态规划应用于 MDP 的规划问题 (planning) 而不是学习问题 (learning)，我们必须对环境是完全已知的 (Model-Based)，才能做动态规划，直观的说，就是要知道状态转移概率和对应的奖励才行

动态规划能够完成预测问题和控制问题的求解，是解 MDP prediction 和 control 一个非常有效的方式。

2.3.10 Policy Evaluation on MDP

Policy evaluation 就是给定一个 MDP 和一个 policy，我们可以获得多少的价值。就对于当前这个策略，我们可以得到多大的 value function。

这里有一个方法是说，我们直接把这个 Bellman Expectation Backup 拿过来，变成一个迭代的过程，这样反复迭代直到收敛。这个迭代过程可以看作是 synchronous backup 的过程。

同步备份 (synchronous backup) 是指每一次的迭代都会完全更新所有的状态，这样对于程序资源需求特别大。异步备份 (asynchronous backup) 的思想就是通过某种方式，使得每一次迭代不需要更新所有的状态，因为事实上，很多的状态也不需要被更新。

$$v_{t+1}(s) = \sum_{a \in \mathcal{A}} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' | s, a) v_t(s') \right) \quad (2.37)$$

式 (2.37) 说的是说我们可以把 Bellman Expectation Backup 转换成一个动态规划的迭代。当我们得到上一时刻的 v_t 的时候，就可以通过这个递推的关系来推出下一时刻的值。反复去迭代它，最后它的值就是从 v_1, v_2 到最后收敛过后的这个值 v^π 。 v^π 就是当前给定的 policy π 对应的价值函数。

Policy evaluation 的核心思想就是把如式 (2.37) 所示的 Bellman expectation backup 拿出来反复迭代，然后就会得到一个收敛的价值函数的值。

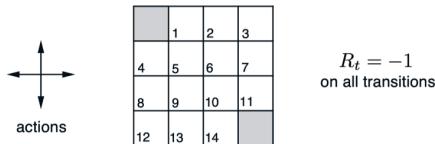
因为已经给定了这个函数的 policy function，那我们可以直接把它简化成一个 MRP 的表达形式，这样的话，形式就更简洁一些，就相当于我们把这个 a 去掉，如下式所示：

$$v_{t+1}(s) = R^\pi(s) + \gamma P^\pi(s' | s) v_t(s') \quad (2.38)$$

这样它就只有价值函数跟转移函数了。通过去迭代这个更简化的一个函数，我们也可以得到它每个状态的价值。因为不管是在 MRP 以及 MDP，它的价值函数包含的这个变量都是只跟这个状态有关，就相当于进入某一个状态，未来可能得到多大的价值。

Evaluating a Random Policy in the Small Gridworld

Example 4.1 in the Sutton RL textbook.



- ① Undiscounted episodic MDP ($\gamma = 1$)
- ② Nonterminal states 1, ..., 14
- ③ Two terminal states (two shaded squares)
- ④ Action leading out of grid leaves state unchanged, $P(7|7, right) = 1$
- ⑤ Reward is -1 until the terminal state is reach
- ⑥ Transition is deterministic given the action, e.g., $P(6|5, right) = 1$
- ⑦ Uniform random policy $\pi(l|.) = \pi(r|.) = \pi(u|.) = \pi(d|.) = 0.25$

图 2.19

比如现在的环境是一个 small gridworld。这个 agent 的目的是从某一个状态开始，然后到达终点状态。它的终止状态就是左上角跟右下角，这里总共有 14 个状态，因为我们把每个位置用一个状态来表示。这个 agent 采取的动作，它的 policy function 就直接先给定了，它在每一个状态都是随机游走，它们在每一个状态就是上下左右行走。它在边缘状态的时候，比如说在第四号状态的时候，它往左走的话，它是依然存在第四号状态，我们加了这个限制。

这里我们给的奖励函数就是说你每走一步，就会得到 -1 的奖励，所以 agent 需要尽快地到达终止状态。状态之间的转移也是确定的。比如从第六号状态往上走，它就会直接到达第二号状态。很多时候有些环境是概率性的 (probabilistic)，就是说 agent 在第六号状态，它选择往上走的时候，有可能地板是滑的，

然后它可能滑到第三号状态或者第一号状态，这就是有概率的一个转移。但这里把这个环境进行了简化，从六号往上走，它就到了二号。所以直接用这个迭代来解它，因为我们已经知道每一个概率以及它的这个概率转移，那么就直接可以进行一个简短的迭代，这样就会算出它每一个状态的价值。

我们再来看一个动态的例子，首先推荐斯坦福大学的一个网站：GridWorld: Dynamic Programming Demo，这个网站模拟了如式 (2.37) 所示的单步更新的过程中，所有格子的一个状态价值的变化过程。

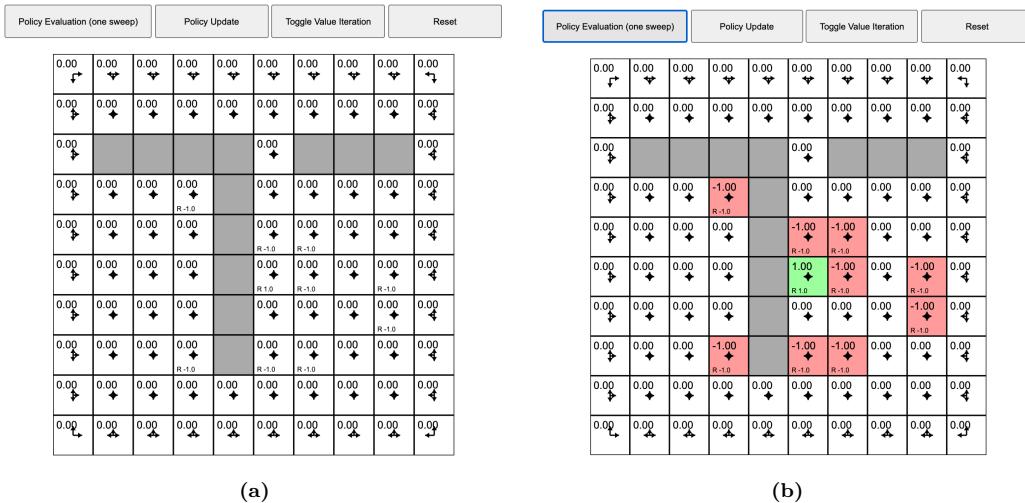


图 2.20

如图 2.20a 所示，这里有很多格子，每个格子都代表了一个状态。在每个格子里面有一个初始值零。然后在每一个状态，它还有一些箭头，这个箭头就是说它在当前这个状态应该采取什么样的策略。我们这里采取一个随机的策略，不管它在哪一个状态，它上下左右的概率都是相同的。比如在某个状态，它都有上下左右 0.25 的概率采取某一个动作，所以它的动作是完全随机的。

在这样的环境里面，我们想计算它每一个状态的价值。我们也定义了它的 reward function，你可以看到有些状态上面有一个 R 的值。比如我们这边有些值是为负的，我们可以看到格子里面有几个 -1 的奖励，只有一个 +1 奖励的格子。在这个棋盘的中间这个位置，可以看到有一个 R 的值是 1.0，为正的一个价值函数。所以每个状态对应了一个值，然后有一些状态没有任何值，就说明它的这个 reward function，它的奖励是为零的。

如图 2.20b 所示，我们开始做这个 policy evaluation，policy evaluation 是一个不停迭代的过程。当我们初始化的时候，所有的 $v(s)$ 都是 0。我们现在迭代一次，迭代一次过后，你发现有些状态上面，值已经产生了变化。比如有些状态的值的 R 为 -1，迭代一次过后，它就会得到 -1 的这个奖励。对于中间这个绿色的，因为它的奖励为正，所以它是 +1 的状态。

如图 2.21a 所示，所以当迭代第一次的时候， $v(s)$ 某些状态已经有些值的变化。

如图 2.21b 所示，我们再迭代一次 (one sweep)，然后发现它就从周围的状态也开始有值。因为周围状态跟之前有值的状态是临近的，所以它就相当于把旁边这个状态转移过来。所以当我们逐渐迭代的话，你会发现这个值一直在变换。

等迭代了很多次过后，很远的这些状态的价值函数已经有些值了，而且你可以发现它这里整个过程呈现逐渐扩散开的一个过程，这其实也是 policy evaluation 的一个可视化。当我们每一步在进行迭代的时候，远的状态就会得到了一些值，就逐渐从一些已经有奖励的这些状态，逐渐扩散，当你 run 很多次过后，它就逐渐稳定下来，最后值就会确定不变，这样收敛过后，每个状态上面的值就是它目前得到的这个 value function 的值。



图 2.21

2.3.11 MDP Control

Policy evaluation 是说给定一个 MDP 和一个 policy，我们可以估算出它的价值函数。还有问题是说如果我们只有一个 MDP，如何去寻找一个最佳的策略，然后可以得到一个最佳价值函数 (Optimal Value Function)。

Optimal Value Function 的定义如下式所示：

$$v^*(s) = \max_{\pi} v^{\pi}(s) \quad (2.39)$$

Optimal Value Function 是说，我们去搜索一种 policy π 来让每个状态的价值最大。 v^* 就是到达每一个状态，它的值的极大化情况。

在这种极大化情况上面，我们得到的策略可以说它是最佳策略 (optimal policy)，如下式所示：

$$\pi^*(s) = \arg \max_{\pi} v^{\pi}(s) \quad (2.40)$$

Optimal policy 使得每个状态的价值函数都取得最大值。所以如果我们可以得到一个 optimal value function，就可以说某一个 MDP 的环境被解。在这种情况下，它的最佳的价值函数是一致的，就它达到的这个上限的值是一致的，但这里可能有多个最佳的 policy，就是说多个 policy 可以取得相同的最佳价值。

Q: 怎么去寻找这个最佳的 policy ?

A: 当取得最佳的价值函数过后，我们可以通过对这个 Q 函数进行极大化来得到最佳策略，如式 (2.41) 所示。

$$\pi^*(a | s) = \begin{cases} 1, & \text{if } a = \arg \max_{a \in A} q^*(s, a) \\ 0, & \text{otherwise} \end{cases} \quad (2.41)$$

当所有东西都收敛后，因为 Q 函数是关于状态跟动作的一个函数，所以在某一个状态采取一个动作，可以使得这个 Q 函数最大化，那么这个动作就应该是最佳的动作。所以如果我们能优化出一个 Q 函数 $q^*(s, a)$ ，就可以直接在这个 Q 函数上面取一个让 Q 函数最大化的 action 的值，就可以提取出它的最佳策略。

Q: 怎样进行策略搜索？

A: 最简单的策略搜索办法就是穷举。假设状态和动作都是有限的，那么每个状态我们可以采取这个 A 种动作的策略，那么总共就是 $|A|^{|S|}$ 个可能的 policy。那我们可以把策略都穷举一遍，然后算出每种策略的 value function，对比一下就可以得到最佳策略。

但是穷举非常没有效率，所以我们要采取其他方法。搜索最佳策略有两种常用的方法：policy iteration 和 value iteration。

寻找这个最佳策略的过程就是 MDP control 过程。MDP control 说的就是怎么去寻找一个最佳的策略来让我们得到一个最大的价值函数，如下式所示：

$$\pi^*(s) = \arg \max_{\pi} v^{\pi}(s) \quad (2.42)$$

对于一个事先定好的 MDP 过程，当 agent 去采取最佳策略的时候，我们可以说最佳策略一般都是确定的，而且是稳定的（它不会随着时间的变化）。但是不一定是唯一的，多种动作可能会取得相同的这个价值。

我们可以通过 policy iteration 和 value iteration 来解 MDP 的控制问题。

2.3.12 Policy Iteration

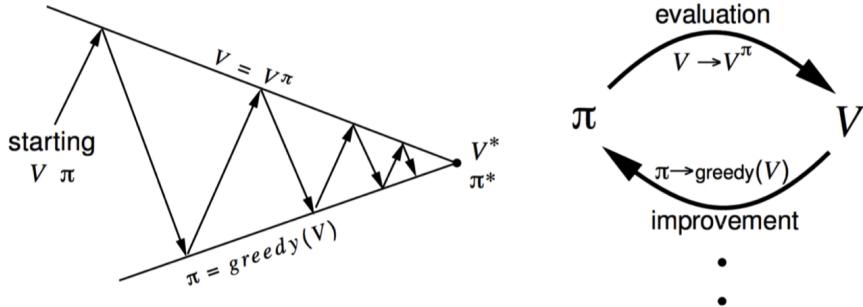


图 2.22 Policy Iteration

Policy iteration 由两个步骤组成：policy evaluation 和 policy improvement。

第一个步骤是 policy evaluation，当前我们在优化这个 policy π ，在优化过程中得到一个最新的 policy。我们先保证这个 policy 不变，然后去估计它出来的这个价值。给定当前的 policy function 来估计这个 v 函数。

第二个步骤是 policy improvement，得到 v 函数过后，我们可以进一步推算出它的 Q 函数。得到 Q 函数过后，我们直接在 Q 函数上面取极大化，通过在这个 Q 函数上面做一个贪心的搜索来进一步改进它的策略。

这两个步骤就一直在迭代进行，所以在 policy iteration 里面，在初始化的时候，我们有一个初始化的 V 和 π ，然后就是在这两个过程之间迭代。左边这幅图上面的线就是我们当前 v 的值，下面的线是 policy 的值。跟踢皮球一样，我们先给定当前已有的这个 policy function，然后去算它的 v 。算出 v 过后，我们会得到一个 Q 函数。Q 函数我们采取 greedy 的策略，这样就像踢皮球，踢回这个 policy。然后进一步改进那个 policy，得到一个改进的 policy 过后，它还不是最佳的，我们再进行 policy evaluation，然后又会得到一个新的 value function。基于这个新的 value function 再进行 Q 函数的极大化，这样就逐渐迭代，然后就会得到收敛。

这里再来看一下第二个步骤：policy improvement，我们是如何改进它的这个策略。得到这个 v 值过后，我们就可以通过这个 reward function 以及状态转移把它的这个 Q-function 算出来，如下式所示：

$$q^{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^{\pi_i}(s') \quad (2.43)$$

对于每一个状态，第二个步骤会得到它的新一轮的这个 policy，就在每一个状态，我们去取使它得到

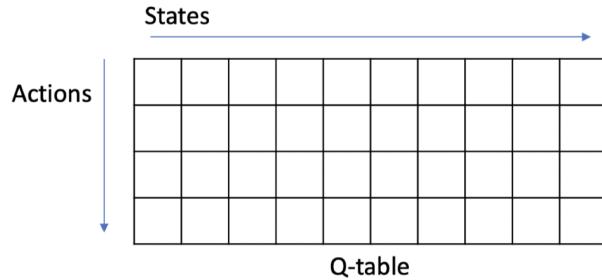


图 2.23

最大值的 action，如下式所示：

$$\pi_{i+1}(s) = \arg \max_a q^{\pi_i}(s, a) \quad (2.44)$$

你可以把 Q 函数看成一个 Q-table:

横轴是它的所有状态，纵轴是它的可能的 action。

得到 Q 函数后，Q-table 也就得到了。

那么对于某一个状态，每一列里面我们会取最大的那个值，最大值对应的那个 action 就是它现在应该采取的 action。所以 arg max 操作就说在每个状态里面采取一个 action，这个 action 是能使这一列的 Q 最化化的那个动作。

Bellman Optimality Equation

当一直在采取 arg max 操作的时候，我们会得到一个单调的递增。通过采取这种 greedy，即 arg max 操作，我们就会得到更好的或者不变的 policy，而不会使它这个价值函数变差。所以当这个改进停止过后，我们就会得到一个最佳策略。

当改进停止过后，我们取它最大化的这个 action，它直接就会变成它的价值函数，如下式所示：

$$q^\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q^\pi(s, a) = q^\pi(s, \pi(s)) = v^\pi(s) \quad (2.45)$$

所以我们有了一个新的等式：

$$v^\pi(s) = \max_{a \in \mathcal{A}} q^\pi(s, a) \quad (2.46)$$

上式被称为 Bellman optimality equation。从直觉上讲，Bellman optimality equation 表达了这样一个事实：最佳策略下的一个状态的价值必须等于在这个状态下采取最好动作得到的回报的期望。

当 MDP 满足 Bellman optimality equation 的时候，整个 MDP 已经到达最佳的状态。它到达最佳状态过后，对于这个 Q 函数，取它最大的 action 的那个值，就是直接等于它的最佳的 value function。只有当整个状态已经收敛过后，得到一个最佳的 policy 的时候，这个条件才是满足的。

最佳的价值函数到达过后，这个 Bellman optimality equation 就会满足。

满足过后，就有这个 max 操作，如式 (2.47) 所示：

$$v^*(s) = \max_a q^*(s, a) \quad (2.47)$$

当我们取最大的这个 action 的时候对应的值就是当前状态的最佳的价值函数。

另外，我们给出 Q 函数的 Bellman equation：

$$q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^*(s') \quad (2.48)$$

我们可以把式 (2.47) 插入到式 (2.48) 里面去，如式 (2.49) 所示：

$$\begin{aligned} q^*(s, a) &= R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^*(s') \\ &= R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) \max_a q^*(s', a') \end{aligned} \quad (2.49)$$

我们就会得到 Q 函数之间的转移。它下一步这个状态，取了 max 这个值过后，就会跟它最佳的这个状态等价。

Q-learning 是基于 Bellman Optimality Equation 来进行的，当取它最大的这个状态的时候($\max_{a'} q^*(s', a')$)，它会满足下面这个等式：

$$q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) \max_{a'} q^*(s', a') \quad (2.50)$$

我们还可以把式 (2.48) 插入到式 (2.47)，如下式所示：

$$\begin{aligned} v^*(s) &= \max_a q^*(s, a) \\ &= \max_a \mathbb{E}[G_t | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v^*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[R_{t+1}] + \max_a \mathbb{E}[\gamma v^*(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a R(s, a) + \max_a \gamma \sum_{s' \in S} P(s' | s, a) v^*(s') \\ &= \max_a \left(R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v^*(s') \right) \end{aligned} \quad (2.51)$$

我们就会得到状态-价值函数的一个转移。

2.3.13 Value Iteration

Principle of Optimality

我们从另一个角度思考问题，动态规划的方法将优化问题分成两个部分：

第一步执行的是最优的 action；之后后继的状态每一步都按照最优的 policy 做，那么我最后的结果就是最优的。

Principle of Optimality Theorem:

一个 policy $\pi(s|a)$ 在状态 s 达到了最优价值，也就是 $v^\pi(s) = v^*(s)$ 成立，当且仅当：

对于任何能够从 s 到达的 s' ，都已经达到了最优价值，也就是，对于所有的 s' ， $v^\pi(s') = v^*(s')$ 恒成立。

Deterministic Value Iteration

如果我们知道子问题 $v^*(s')$ 的最优解，我们就可以通过 value iteration 来得到最优的 $v^*(s)$ 的解。Value iteration 就是把 Bellman Optimality Equation 当成一个 update rule 来进行，如下式所示：

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v(s') \right) \quad (2.52)$$

之前我们说上面这个等式只有当整个 MDP 已经到达最佳的状态时才满足。但这里可以把它转换成一个 backup 的等式。Backup 就是说一个迭代的等式。我们不停地去迭代 Bellman Optimality Equation，到了最后，它能逐渐趋向于最佳的策略，这是 value iteration 算法的精髓。

为了得到最佳的 v^* ，对于每个状态的 v^* ，我们直接把这个 Bellman Optimality Equation 进行迭代，迭代了很多次之后，它就会收敛。

Value Iteration 算法：

1. 初始化：令 $k = 1$ ，对于所有状态 s , $v_0(s) = 0$

2. For $k = 1 : H$

(a) 对于所有状态 s

$$q_{k+1}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v_k(s') \quad (2.53)$$

$$v_{k+1}(s) = \max_a q_{k+1}(s, a) \quad (2.54)$$

(b) $k \leftarrow k + 1$

3. 为了在迭代后提取最优策略：

$$\pi(s) = \arg \max_a R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v_{k+1}(s') \quad (2.55)$$

我们使用 value iteration 算法是为了得到一个最佳的策略 π 。解法：我们可以直接把 Bellman Optimality backup 这个等式拿进来进行迭代，迭代很多次，收敛过后得到的那个值就是它的最佳的值。这个算法开始的时候，它是先把所有值初始化，通过每一个状态，然后它会进行这个迭代。把式 (2.53) 插到式 (2.54) 里面，就是 Bellman optimality backup 的那个等式。有了式 (2.53) 和式 (2.54) 过后，然后进行不停地迭代，迭代过后，然后收敛，收敛后就会得到这个 v^* 。当我们有了 v^* 过后，一个问题是如何进一步推算出它的最佳策略。提取最佳策略的话，我们可以直接用 $\arg \max$ 。就先把它 Q 函数重构出来，重构出来过后，每一个列对应的最大的那个 action 就是它现在的最佳策略。这样就可以从最佳价值函数里面提取出最佳策略。我们只是在解决一个 planning 的问题，而不是强化学习的问题，因为我们知道环境如何变化。

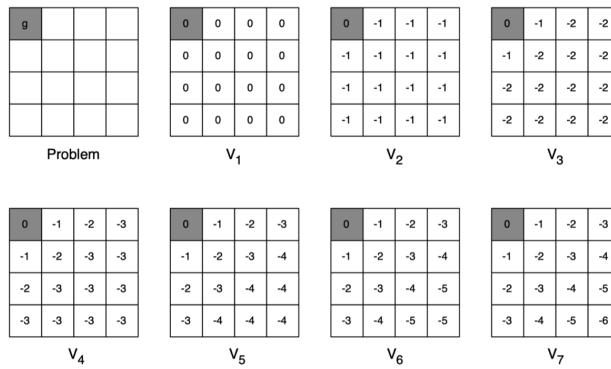


图 2.24 例子：最短路径

value function 做的工作类似于 value 的反向传播，每次迭代做一步传播，所以中间过程的 policy 和 value function 是没有意义的。不像是 policy iteration，它每一次迭代的结果都是有意义的，都是一个完整的 policy。上图是一个可视化的过程，在一个 gridworld 中，我们设定了一个终点 (goal)，也就是左上角的点。不管你在哪一个位置开始，我们都希望能够到终点（实际上这个终点是在迭代过程中不必要的，只是为了更好的演示）。Value iteration 的迭代过程像是一个从某一个状态（这里是我们的 goal）反向传播其他各个状态的过程。因为每次迭代只能影响到与之直接相关的关系。让我们回忆下 Principle of Optimality Theorem：当你这次迭代求解的某个状态 s 的 value function $v_{k+1}(s)$ 是最优解，它的前提是能够从该状态到达的所有状态 s' 此时都已经得到了最优解；如果不是的话，它做的事情只是一个类似传递 value function 的过程。以上图为例，实际上，对于每一个状态，我们都可以看成一个终点。迭代由每一个终点开始，每次都根据 Bellman optimality equation 重新计算 value。如果它的相邻节点 value 发生变化，变得更好，那

么它也会变得更好，一直到相邻节点都不变了。因此，在我们迭代到 v_7 之前，也就是还没将每个终点的最优的 value 传递给其他的所有状态之前，中间的几个 value function 只是一种暂存的不完整的数据，它不能代表每一个 state 的 value function，所以生成的 policy 是一个没有意义的 policy。因为它是一个迭代过程，这里可视化了从 v_1 到 v_7 每一个状态的值的变化，它的这个值逐渐在变化。而且因为它每走一步，就会得到一个负的值，所以它需要尽快地到达左上角，可以发现离它越远的，那个值就越小。 v_7 收敛过后，右下角那个值是 -6，相当于它要走六步，才能到达最上面那个值。而且离目的地越近，它的价值越大。

当我们得到了最优值后，我们可以使用策略提取来得到最佳策略。

2.3.14 Difference between Policy Iteration and Value Iteration

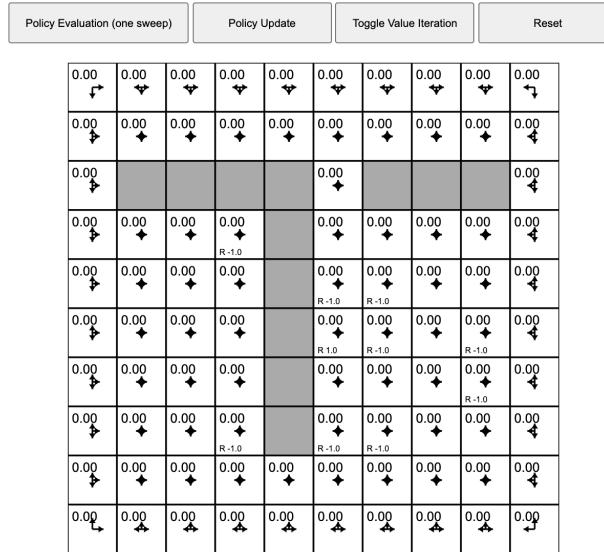


图 2.25

我们来看一个 MDP control 的 Demo。

首先来看 policy iteration。之前的例子在每个状态都是采取固定的随机策略，就每个状态都是 0.25 的概率往上往下往左往右，没有策略的改变。但是我们现在想做 policy iteration，就是每个状态的策略都进行改变。Policy iteration 的过程是一个迭代过程。

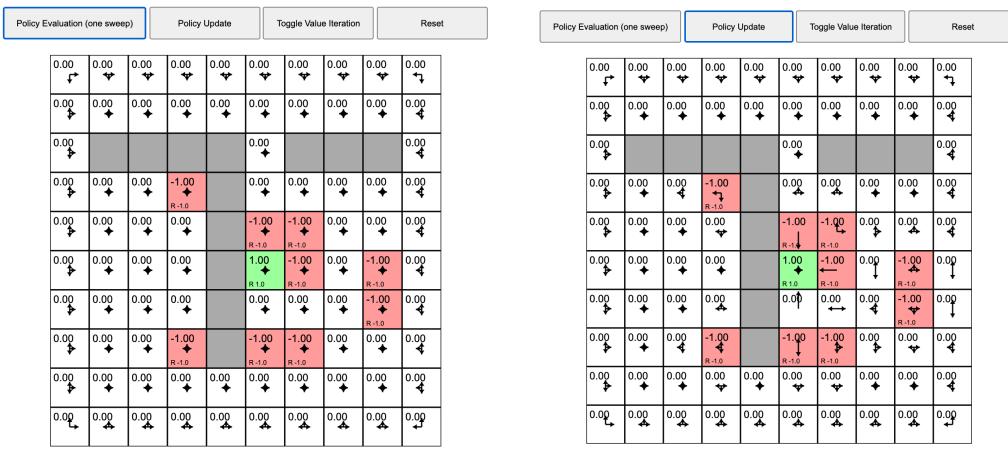


图 2.26

如图 2.26a 所示，我们先在这个状态里面 run 一遍 policy evaluation，就得到了一个 value function，每个状态都有一个 value function。

如图 2.26b 所示，现在进行 policy improvement，点一下 policy update。点一下 policy update 过后，你可以发现有些格子里面的 policy 已经产生变化。比如说对于中间这个 -1 的这个状态，它的最佳策略是往下走。当你到达这个状态后，你应该往下，这样就会得到最佳的这个值。绿色右边的这个方块的策略也改变了，它现在选取的最佳策略是往左走，也就是说在这个状态的时候，最佳策略应该是往左走。

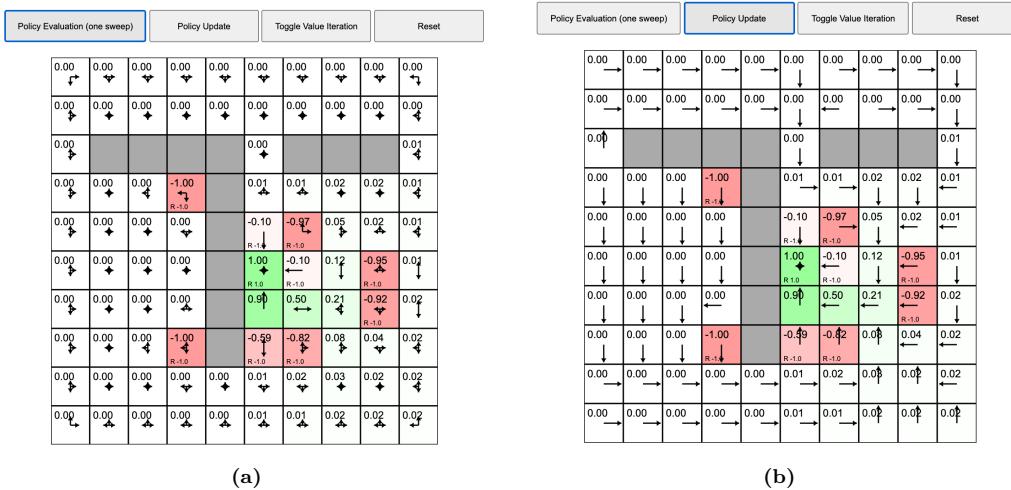


图 2.27

如图 2.27a 所示，我们再 run 下一轮的 policy evaluation，你发现它的值又被改变了，很多次过后，它会收敛。

如图 2.27b 所示，我们再 run policy update，你发现每个状态里面的值基本都改变，它不再是上下左右随机在变了，它会选取一个最佳的策略。

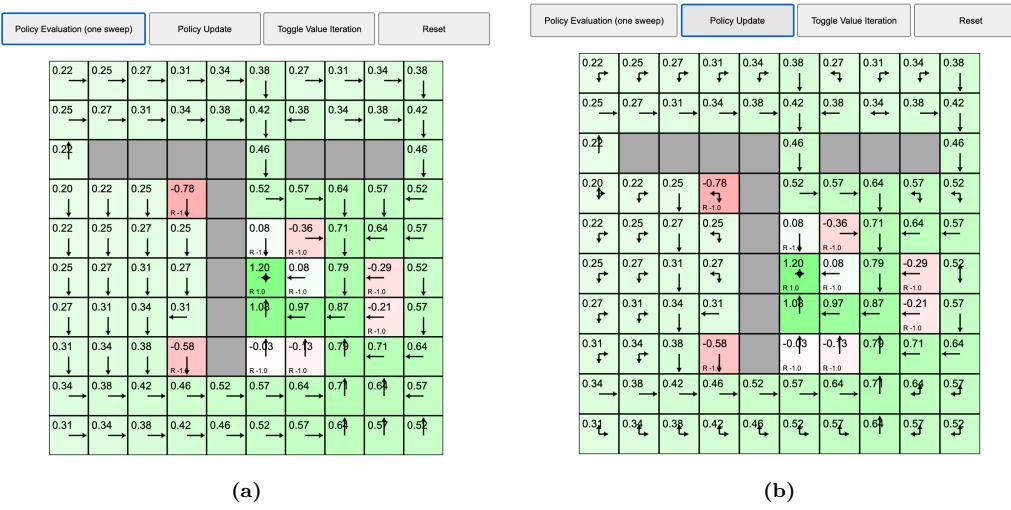


图 2.28

如图 2.28a 所示，我们再 run 这个 policy evaluation，它的值又在不停地变化，变化之后又收敛了。

如图 2.28b 所示，我们再来 run 一遍 policy update。现在它的值又会有变化，就在每一个状态，它的这个最佳策略也会产生一些改变。

如图 2.29a 所示，再来在这个状态下面进行改变，现在你看基本没有什么变化，就说明整个 MDP 已

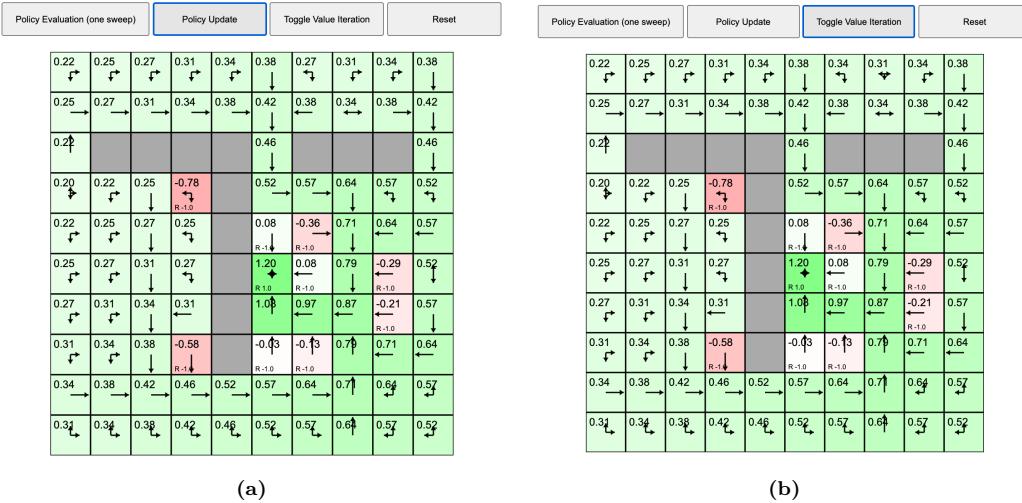


图 2.29

经收敛了。所以现在它每个状态的值就是它当前最佳的 value function 的值以及它当前状态对应的这个 policy 就是最佳的 policy。

比如说现在我们在右上角 0.38 的这个位置，然后它说现在应该往下走，我们往下走一步。它又说往下走，然后再往下走。现在我们有两个选择：往左走和往下走。我们现在往下走，随着这个箭头的指示，我们就会到达中间 1.20 的一个状态。如果能达到这个状态，我们就会得到很多 reward。

这个 Demo 说明了 policy iteration 可以把 gridworld 解决掉。解决掉的意思是说，不管在哪个状态，都可以顺着状态对应的最佳的策略来到达可以获得最多奖励的一个状态。

如图 2.29b 所示，我们再用 value iteration 来解 MDP，点 Toggle value iteration。

- 当它的这个值确定下来过后，它会产生它的最佳状态，这个最佳状态提取的策略跟 policy iteration 得出来的最佳策略是一致的。
- 在每个状态，我们跟着这个最佳策略走，就会到达可以得到最多奖励的一个状态。

这个 Demo 里面是一个代码，就是为了解一个叫 FrozenLake 的例子，这个例子是 OpenAI Gym 里的一个环境，跟 gridworld 很像，不过它每一个状态转移是一个概率。

我们再来对比下 policy iteration 和 value iteration，这两个算法都可以解 MDP 的控制问题。

Policy Iteration 分两步，首先进行 policy evaluation，即对当前已经搜索到的策略函数进行一个估值。得到估值过后，进行 policy improvement，即把 Q 函数算出来，我们进一步进行改进。不断重复这两步，直到策略收敛。

Value iteration 直接把 Bellman Optimality Equation 拿进来，然后去寻找最佳的 value function，没有 policy function 在这里面。当算出 optimal value function 过后，我们再来提取最佳策略。

2.3.15 Summary for Prediction and Control in MDP

表 2.1 Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

总结如表 2.1 所示，就对于 MDP 里面的 prediction 和 control 都是用动态规划来解，我们其实采取

了不同的 Bellman Equation。

如果是一个 prediction 的问题，即 policy evaluation 的问题，直接就是不停地 run 这个 Bellman Expectation Equation，这样我们就可以去估计出给定的这个策略，然后得到价值函数。

对于 control，

- 如果采取的算法是 policy iteration，那这里用的是 Bellman Expectation Equation。把它分成两步，先上它的这个价值函数，再去优化它的策略，然后不停迭代。这里用到的只是 Bellman Expectation Equation。
- 如果采取的算法是 value iteration，那这里用到的 Bellman Equation 就是 Bellman Optimality Equation，通过 $\arg \max$ 这个过程，不停地去 $\arg \max$ 它，最后它就会达到最优的状态。

2.4 Keywords

- 马尔可夫性质 (Markov Property): 如果某一个过程未来的转移跟过去是无关，只由现在的状态决定，那么其满足马尔可夫性质。换句话说，一个状态的下一个状态只取决于它当前状态，而跟它当前状态之前的状态都没有关系。
- 马尔可夫链 (Markov Chain): 概率论和数理统计中具有马尔可夫性质 (Markov property) 且存在于离散的指数集 (index set) 和状态空间 (state space) 内的随机过程 (stochastic process)。
- 状态转移矩阵 (State Transition Matrix): 状态转移矩阵类似于一个 conditional probability，当我们知道当前我们在 s_t 这个状态过后，到达下面所有状态的一个概念，它每一行其实描述了是从一个节点到达所有其它节点的概率。
- 马尔可夫奖励过程 (Markov Reward Process, MRP): 即马尔可夫链再加上了一个奖励函数。在 MRP 之中，转移矩阵跟它的这个状态都是跟马尔可夫链一样的，多了一个奖励函数 (reward function)。奖励函数是一个期望，它说当你到达某一个状态的时候，可以获得多大的奖励。
- horizon: 定义了同一个 episode 或者是整个一个轨迹的长度，它是由有限个步数决定的。
- return: 把奖励进行折扣 (discounted)，然后获得的对应的收益。
- Bellman Equation (贝尔曼等式): 定义了当前状态与未来状态的迭代关系，表示当前状态的值函数可以通过下个状态的值函数来计算。Bellman Equation 因其提出者、动态规划创始人 Richard Bellman 而得名，同时也被叫作“动态规划方程”。 $V(s) = R(S) + \gamma \sum_{s' \in S} P(s'|s)V(s')$ ，特别地，矩阵形式： $V = R + \gamma PV$ 。
- Monte Carlo Algorithm (蒙特卡罗方法): 可用来计算价值函数的值。通俗的讲，我们当得到一个 MRP 过后，我们可以从某一个状态开始，然后让它让把这个小船放进去，让它随波逐流，这样就会产生一个轨迹。产生了一个轨迹过后，就会得到一个奖励，那么就直接把它的 Discounted 的奖励 g 直接算出来。算出来过后就可以把它积累起来，当积累到一定的轨迹数量过后，然后直接除以这个轨迹，然后就会得到它 Iterative Algorithm (动态规划方法): 可用来计算价值函数的值。通过一直迭代对应的 Bellman Equation，最后使其收敛。当这个最后更新的状态跟你上一个状态变化并不大的时候，这个更新就可以停止。
- Q 函数 (action-value function): 其定义的是某一个状态某一个行为，对应的它有可能得到的 return 的一个期望 (over policy function)。
- MDP 中的 prediction (即 policy evaluation 问题): 给定一个 MDP 以及一个 policy π ，去计算它的 value function，即每个状态它的价值函数是多少。其可以通过动态规划方法 (Iterative Algorithm) 解决。
- MDP 中的 control 问题: 寻找一个最佳的一个策略，它的 input 就是 MDP，输出是通过去寻找它的最佳策略，然后同时输出它的最佳价值函数 (optimal value function) 以及它的这个最佳策略 (optimal policy)。其可以通过动态规划方法 (Iterative Algorithm) 解决。
- 最佳价值函数 (Optimal Value Function): 我们去搜索一种 policy π ，然后我们会得到每个状态它

的状态值最大的一个情况， v^* 就是到达每一个状态，它的值的极大化情况。在这种极大化情况上面，我们得到的策略就可以说它是最佳策略 (optimal policy)。optimal policy 使得每个状态，它的状态函数都取得最大值。所以当我们说某一个 MDP 的环境被解了过后，就是说我们可以得到一个 optimal value function，然后我们就说它被解了。

2.5 Questions

- 为什么在马尔可夫奖励过程 (MRP) 中需要有 discount factor?

答：

1. 首先，是有些马尔可夫过程是带环的，它并没有终结，然后我们想避免这个无穷的奖励；2. 另外，我们是想把这个不确定性也表示出来，希望尽可能快地得到奖励，而不是在未来某一个点得到奖励；3. 接上面一点，如果这个奖励是有实际价值的了，我们可能是更希望立刻就得到奖励，而不是我们后面再得到奖励。4. 还有在有些时候，这个系数也可以把它设为 0。比如说，当我们设为 0 过后，然后我们就只关注了它当前的奖励。我们也可以把它设为 1，设为 1 的话就是对未来并没有折扣，未来获得的奖励跟我们当前获得的奖励是一样的。

所以，这个系数其实是应该可以作为强化学习 agent 的一个 hyperparameter 来进行调整，然后就会得到不同行为的 agent。

- 为什么矩阵形式的 Bellman Equation 的解析解比较难解?

答：通过矩阵求逆的过程，就可以把这个 V 的这个价值的解析解直接求出来。但是一个问题就是这个矩阵求逆的过程的复杂度是 $O(N^3)$ 。所以就当我们状态非常多的时候，比如说从我们现在十个状态到一千个状态，到一百万个状态。那么当我们有一百万个状态的时候，这个转移矩阵就会是个一千万乘以一千万的一个矩阵。这样一个大矩阵的话求逆是非常困难的，所以这种通过解析解去解，只能对于很小量的 MRP。

- 计算贝尔曼等式 (Bellman Equation) 的常见方法以及区别?

答：

1. Monte Carlo Algorithm (蒙特卡罗方法)：可用来计算价值函数的值。通俗的讲，我们当得到一个 MRP 过后，我们可以从某一个状态开始，然后让它让把这个小船放进去，让它随波逐流，这样就会产生一个轨迹。产生了一个轨迹过后，就会得到一个奖励，那么就直接把它的 Discounted 的奖励 g 直接算出来。算出来过后就可以把它积累起来，当积累到一定的轨迹数量过后，然后直接除以这个轨迹，然后就会得到它的这个价值。
2. Iterative Algorithm (动态规划方法)：可用来计算价值函数的值。通过一直迭代对应的 Bellman Equation，最后使其收敛。当这个最后更新的状态跟你上一个状态变化并不大的时候，通常是一个阈值 γ ，这个更新就可以停止。
3. 以上两者的结合方法：另外我们也可以通过 Temporal-Difference Learning 的那个办法。这个 Temporal-Difference Learning 叫 TD Learning，就是动态规划和蒙特卡罗的一个结合。

- 马尔可夫奖励过程 (MRP) 与马尔可夫决策过程 (MDP) 的区别?

答：相对于 MRP，马尔可夫决策过程 (Markov Decision Process) 多了一个 decision，其它的定义跟 MRP 都是类似的。这里我们多了一个决策，多了一个 action，那么这个状态转移也多了一个 condition，就是采取某一种行为，然后你未来的状态会不同。它不仅是依赖于你当前的状态，也依赖于在当前状态你这个 agent 它采取的这个行为会决定它未来的这个状态走向。对于这个价值函数，它也是多了一个条件，多了一个你当前的这个行为，就是说你当前的状态以及你采取的行为会决定你在当前可能得到的奖励多少。

另外，两者之间是有转换关系的。具体来说，已知一个 MDP 以及一个 policy π 的时候，我们可以把 MDP 转换成 MRP。在 MDP 里面，转移函数 $P(s'|s, a)$ 是基于它当前状态以及它当前的 action，因为我们现在已知它 policy function，就是说在每一个状态，我们知道它可能采取的行为的概率，那么

就可以直接把这个 action 进行加和，那我们就可以得到对于 MRP 的一个转移，这里就没有 action。同样地，对于奖励，我们也可以把 action 拿掉，这样就会得到一个类似于 MRP 的奖励。

- MDP 里面的状态转移跟 MRP 以及 MP 的结构或者计算方面的差异？

答：

- 对于之前的马尔可夫链的过程，它的转移是直接就决定，就从你当前是 s ，那么就直接通过这个转移概率就直接决定了你下一个状态会是什么。
- 但是对于 MDP，它的中间多了一层这个行为 a ，就是说在你当前这个状态的时候，你首先要决定的是采取某一种行为。然后因为你有一定的不确定性，当你当前状态决定你当前采取的行为过后，你到未来的状态其实也是一个概率分布。所以你采取行为以及你决定，然后你可能有有多大的概率到达某一个未来状态，以及另外有多大概率到达另外一个状态。所以在这个当前状态跟未来状态转移过程中这里多了一层决策性，这是 MDP 跟之前的马尔可夫过程很不同的一个地方。在马尔科夫决策过程中，行为是由 agent 决定，所以多了一个 component，agent 会采取行为来决定未来的状态转移。

- 我们如何寻找最佳的 policy，方法有哪些？

答：本质来说，当我们取得最佳的价值函数过后，我们通过对这个 Q 函数进行极大化，然后得到最佳的价值。然后，我们直接在这个 Q 函数上面取一个让这个 action 最大化的值，然后我们就可以直接提取出它的最佳的 policy。

具体方法：

1. 穷举法（一般不使用）：假设我们有有限多个状态、有限多个行为可能性，那么每个状态我们可以采取这个 A 种行为的策略，那么总共就是 $|A|^{|S|}$ 个可能的 policy。我们可以把这个穷举一遍，然后算出每种策略的 value function，然后对比一下可以得到最佳策略。但是效率极低。
2. Policy iteration：一种迭代方法，有两部分组成，下面两个步骤一直在迭代进行，最终收敛：（有些类似于 ML 中 EM 算法（期望-最大化算法））
 - 第一个步骤是 policy evaluation，即当前我们在优化这个 policy π ，所以在优化过程中得到一个最新的这个 policy。
 - 第二个步骤是 policy improvement，即取得价值函数后，进一步推算出它的 Q 函数。得到 Q 函数过后，那我们就直接去取它的极大化。
3. Value iteration：我们一直去迭代 Bellman Optimality Equation，到了最后，它能逐渐趋向于最佳的策略，这是 value iteration 算法的精髓，就是我们去为了得到最佳的 v^* ，对于每个状态它的 v^* 这个值，我们直接把这个 Bellman Optimality Equation 进行迭代，迭代了很多次之后它就会收敛到最佳的 policy 以及其对应的状态，这里面是没有 policy function 的。

2.6 Something About Interview

- 高冷的面试官：请问马尔可夫过程是什么？马尔可夫决策过程又是什么？其中马尔可夫最重要的性质是什么呢？

答：马尔可夫过程是一个二元组 $\langle S, P \rangle$, S 为状态的集合, P 为状态转移概率矩阵；而马尔可夫决策过程是一个五元组 $\langle S, P, A, R, \gamma \rangle$ ，其中 R 表示为从 S 到 S' 能够获得的奖励期望， γ 为折扣因子， A 为动作集合。马尔可夫最重要的性质是下一个状态只与当前状态有关，与之前的状态无关，也就是 $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \dots, S_t]$

- 高冷的面试官：请问我们一般怎么求解马尔可夫决策过程？

答：我们直接求解马尔可夫决策过程可以直接求解贝尔曼等式（动态规划方程），即 $V(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s)V(s')$ ，特别地，矩阵形式： $V = R + \gamma PV$ 。但是贝尔曼等式很难求解且计算复杂度较高，所以可以使用动态规划，蒙特卡洛，时间差分等方法求解。

- 高冷的面试官：请问如果数据流不满足马尔科夫性怎么办？应该如何处理？

答：如果不满足马尔科夫性，即下一个状态与之前的状态也有关，若还仅仅用当前的状态来进行求解决策过程，势必导致决策的泛化能力变差。为了解决这个问题，可以利用 RNN 对历史信息建模，获得包含历史信息的状态表征。表征过程可以使用注意力机制等手段。最后在表征状态空间求解马尔可夫决策过程问题。

- 高冷的面试官：请分别写出基于状态值函数的贝尔曼方程以及基于动作值的贝尔曼方程。

答：

- 基于状态值函数的贝尔曼方程: $v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r(s,a) + \gamma v_\pi(s')]$
- 基于动作值的贝尔曼方程: $q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) [r(s',a) + \gamma v_\pi(s')]$
- 高冷的面试官：请问最佳价值函数 (optimal value function) v^* 和最佳策略 (optimal policy) π^* 为什么等价呢？

答：最佳价值函数的定义为: $v^*(s) = \max_\pi v^\pi(s)$ 即我们去搜索一种 policy π 来让每个状态的价值最大。 v^* 就是到达每一个状态，它的值的极大化情况。在这种极大化情况上面，我们得到的策略就可以说它是最佳策略 (optimal policy)，即 $\pi^*(s) = \arg \max_\pi v^\pi(s)$. Optimal policy 使得每个状态的价值函数都取得最大值。所以如果我们可以得到一个 optimal value function，就可以说某一个 MDP 的环境被解。在这种情况下，它的最佳的价值函数是一致的，就它达到的这个上限的值是一致的，但这里可能有多个最佳的 policy，就是说多个 policy 可以取得相同的最佳价值。

- 高冷的面试官：能不能手写一下第 n 步的值函数更新公式呀？另外，当 n 越来越大时，值函数的期望和方差分别变大还是变小呢？

答：n 越大，方差越大，期望偏差越小。值函数的更新公式？话不多说，公式如下：

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[\sum_{i=1}^n \gamma^{i-1} R_{t+i} + \gamma^n \max_a Q(S', a) - Q(S, A) \right]$$

References

- 强化学习基础 David Silver 笔记
- Reinforcement Learning: An Introduction (second edition)
- David Silver 强化学习公开课中文讲解及实践
- UCL Course on RL(David Silver)
- Derivation of Bellman's Equation
- 深入浅出强化学习：原理入门

第 3 章 Tabular Methods

本章我们通过最简单的表格型的方法 (tabular methods) 来讲解如何使用 value-based 方法去求解强化学习。

3.1 MDP

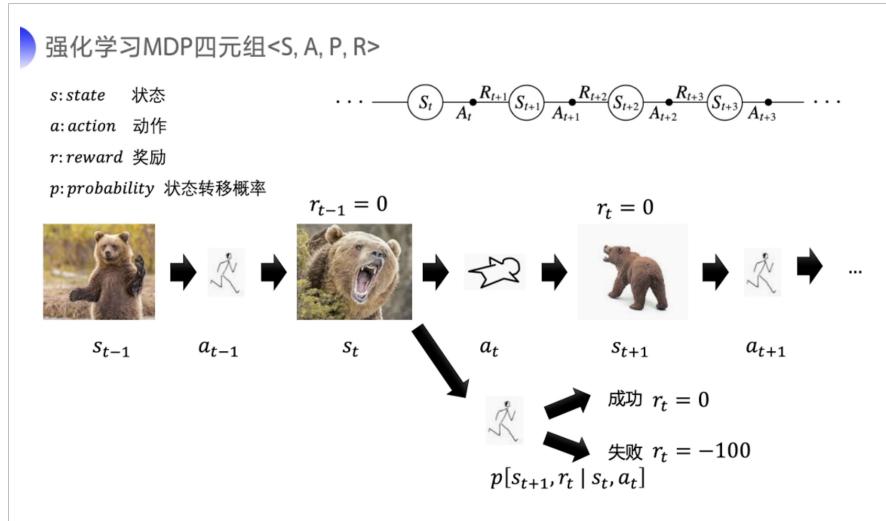


图 3.1

强化学习的三个重要的要素：状态、动作和奖励。强化学习智能体跟环境一步一步交互的，就是我先观察一下状态，然后再输入动作。再观察一下状态，再输出动作，拿到这些 reward。它是一个跟时间相关的序列决策的问题。

举个例子，在 $t-1$ 时刻，我看到了熊对我招手，那我下意识的可能输出的动作就是赶紧跑路。熊看到了有人跑了，可能就觉得发现猎物，开始发动攻击。而在 t 时刻的话，我如果选择装死的动作，可能熊咬了咬我，摔了几下就发现就觉得挺无趣的，可能会走开。这个时候，我再跑路的话可能就跑路成功了，就是这样子的一个序列决策的过程。

当然在输出每一个动作之前，你可以选择不同的动作。比如说在 t 时刻，我选择跑路的时候，熊已经追上来了，如果说 t 时刻，我没有选择装死，而我是选择跑路的话，这个时候熊已经追上了，那这个时候，其实我有两种情况转移到不同的状态去，就我有一定的概率可以逃跑成功，也有很大的概率我会逃跑失败。那我们就用状态转移概率 $p[s_{t+1}, r_t | s_t, a_t]$ 来表述说在 s_t 的状态选择了 a_t 的动作的时候，转移到 s_{t+1} ，而且拿到 r_t 的概率是多少。

这样子的一个状态转移概率是具有马尔可夫性质的（系统下一时刻的状态仅由当前时刻的状态决定，不依赖于以往任何状态）。因为这个状态转移概率，它是下一时刻的状态是取决于当前的状态，它和之前的 s_{t-1} 和 s_{t-2} 都没有什么关系。然后再加上这个过程也取决于智能体跟环境交互的这个 a_t ，所以有一个决策的一个过程在里面。我们就称这样的一个过程为马尔可夫决策过程 (Markov Decision Process, MDP)。

MDP 就是序列决策这样一个经典的表达方式。MDP 也是强化学习里面一个非常基本的学习框架。状态、动作、状态转移概率和奖励 (S, A, P, R) ，这四个合集就构成了强化学习 MDP 的四元组，后面也可能会再加个衰减因子构成五元组。

3.1.1 Model-based

如上图所示，我们把这些可能的动作和可能的状态转移的关系画成一个树状图。它们之间的关系就是从 s_t 到 a_t ，再到 s_{t+1} ，再到 a_{t+1} ，再到 s_{t+2} 这样子的一个过程。

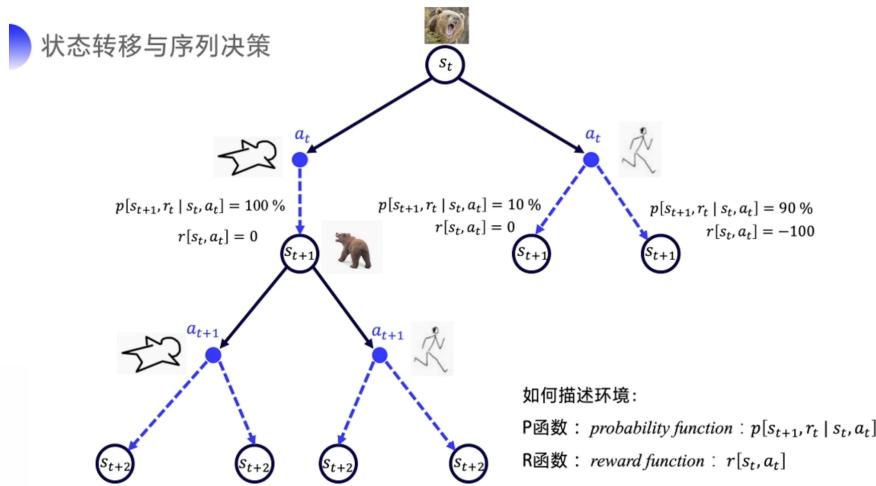


图 3.2

我们去跟环境交互，只能走完整的一条通路。这里面产生了一系列的一个决策的过程，就是我们跟环境交互产生了一个经验。我们会使用 P 函数 (probability function) 和 R 函数 (reward function) 来去描述环境。P 函数就是状态转移的概率，P 函数实际上反映的是环境的一个随机性。

当我们知道 P 函数和 R 函数时，我们就说这个 MDP 是已知的，可以通过 policy iteration 和 value iteration 来找最佳的策略。

比如，在熊发怒的情况下，我如果选择装死，假设熊看到人装死就一定会走的话，我们就称在里面的 state 转移概率就是 100%。但如果说在熊发怒的情况下，我选择跑路而导致可能跑成功以及跑失败，出现这两种情况。那我们就可以用概率去表达一下说转移到其中一种情况的概率大概 10%，另外一种情况的概率大概是 90% 会跑失败。

如果知道这些状态转移概率和奖励函数的话，我们就说这个环境是已知的，因为我是用这两个函数去描述环境的。如果是已知的话，我们其实可以用动态规划去计算说，如果要逃脱熊，那么能够逃脱熊概率最大的最优策略是什么。很多强化学习的经典算法都是 model-free 的，就是环境是未知的。

3.1.2 Model-free

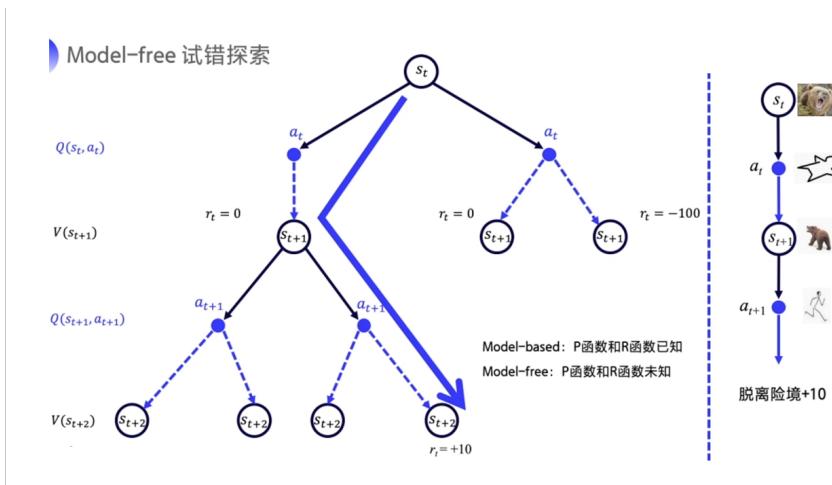


图 3.3

因为现实世界中人类第一次遇到熊之前，我们根本不知道能不能跑得过熊，所以刚刚那个 10%、90%

的概率也就是虚构出来的概率。熊到底在什么时候会往什么方向去转变的话，我们经常是不知道的。

我们是处在一个未知的环境里的，也就是这一系列的决策的 P 函数和 R 函数是未知的，这就是 model-based 跟 model-free 的一个最大的区别。

强化学习就是可以用来解决用完全未知的和随机的环境。强化学习要像人类一样去学习，人类学习的话就是一条路一条路地去尝试一下，先走一条路，看看结果到底是什么。多试几次，只要能活命的。我们可以慢慢地了解哪个状态会更好，

- 我们用价值函数 $V(s)$ 来代表这个状态是好的还是坏的。
- 用 Q 函数来判断说在什么状态下做什么动作能够拿到最大奖励，用 Q 函数来表示这个状态-动作值。

3.1.3 Model-based vs. Model-free

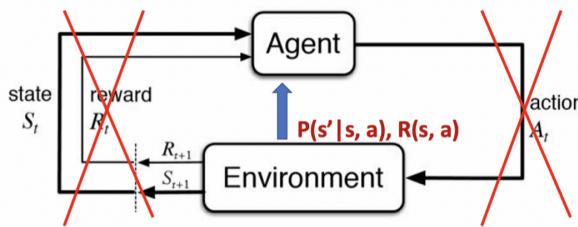


图 3.4

Policy iteration 和 value iteration 都需要得到环境的转移和奖励函数，所以在这个过程中，agent 没有跟环境进行交互。在很多实际的问题中，MDP 的模型有可能是未知的，也有可能模型太大了，不能进行迭代的计算。比如 Atari 游戏、围棋、控制直升飞机、股票交易等问题，这些问题的状态转移太复杂了。

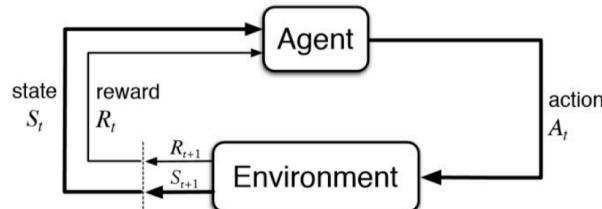


图 3.5

在这种情况下，我们使用 model-free 强化学习的方法来解。Model-free 没有获取环境的状态转移和奖励函数，我们让 agent 跟环境进行交互，采集到很多的轨迹数据，agent 从轨迹中获取信息来改进策略，从而获得更多的奖励。

3.2 Q-table

接下来介绍下 Q 函数。在多次尝试和熊打交道之后，人类就可以对熊的不同的状态去做出判断，我们可以用状态动作价值来表达说在某个状态下，为什么动作 1 会比动作 2 好，因为动作 1 的价值比动作 2 要高，这个价值就叫 Q 函数。

如果 Q 表格是一张已经训练好的表格的话，那这一张表格就像是一本生活手册。我们就知道在熊发怒的时候，装死的价值会高一点。在熊离开的时候，我们可能偷偷逃跑的会比较容易获救。

这张表格里面 Q 函数的意义就是我选择了这个动作之后，最后能不能成功，就是我要去计算在这个状态下，我选择了这个动作，后续能够一共拿到多少总收益。如果可以预估未来的总收益的大小，我们当然知道在当前的这个状态下选择哪个动作，价值更高。我选择某个动作是因为我未来可以拿到的那个

Q表格：状态动作价值：一本生活手册：取得成功的知识

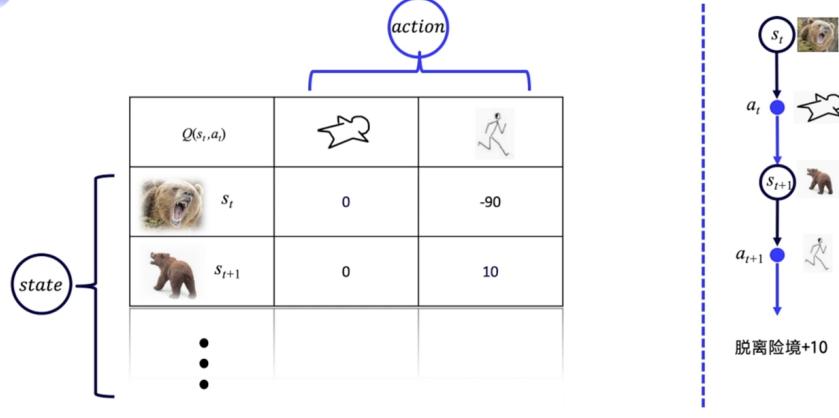


图 3.6

价值会更高一点。所以强化学习的目标导向性很强，环境给出的 reward 是一个非常重要的反馈，它就是根据环境的 reward 来去做选择。

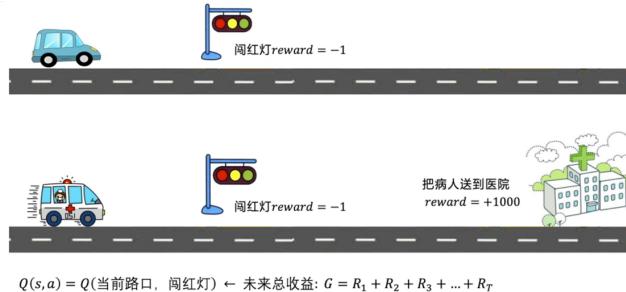


图 3.7

Q: 为什么可以用未来的总收益来评价当前这个动作是好是坏？

A: 举个例子，假设一辆车在路上，当前是红灯，我们直接走的收益就很低，因为违反交通规则，这就是当前的单步收益。可是如果我们这是一辆救护车，我们正在运送病人，把病人快速送达医院的收益非常的高，而且越快你的收益越大。在这种情况下，我们很可能应该要闯红灯，因为未来的远期收益太高了。这也是为什么强化学习需要去学习远期的收益，因为在现实世界中奖励往往是延迟的。所以我们一般会从当前状态开始，把后续有可能会收到所有收益加起来计算当前动作的 Q 的价值，让 Q 的价值可以真正地代表当前这个状态下，动作的真正的价值。

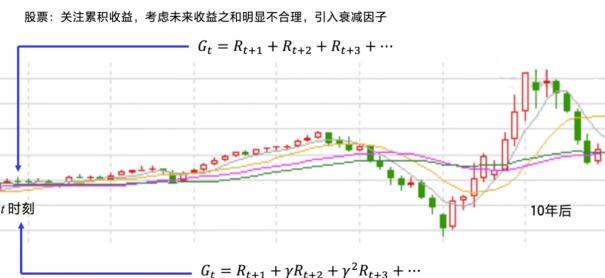


图 3.8

但有的时候把目光放得太长远不好，因为如果事情很快就结束的话，你考虑到最后一步的收益无可厚非。如果是一个持续的没有尽头的任务，即持续式任务 (Continuing Task)，你把未来的收益全部相加，作

为当前的状态价值就很不合理。

股票的例子就很典型了，我们要关注的是累积的收益。可是如果说十年之后才有一次大涨大跌，你显然不会把十年后的收益也作为当前动作的考虑因素。那我们会怎么办呢，有句俗话说得好，对远一点的东西，我们就当做近视，就不需要看得太清楚，我们可以引入这个衰减因子 γ 来去计算这个未来总收益， $\gamma \in [0, 1]$ ，越往后 γ^n 就会越小，也就是说越后面的收益对当前价值的影响就会越小。

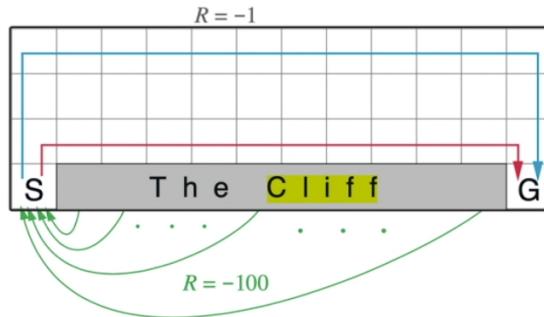


图 3.9

举个例子来看看计算出来的是什么效果。如图 3.9 所示，这是一个悬崖问题（快速到达目的地），这个问题是需要智能体从出发点 S 出发，到达目的地 G，同时避免掉进悬崖（cliff），每走一步都有 -1 的惩罚，掉进悬崖的话就会有 -100 分的惩罚，但游戏不会结束，它会被直接拖回起点，游戏继续，直到到达目的地结束游戏。为了到达目的地，我们可以沿着蓝线和红线走。

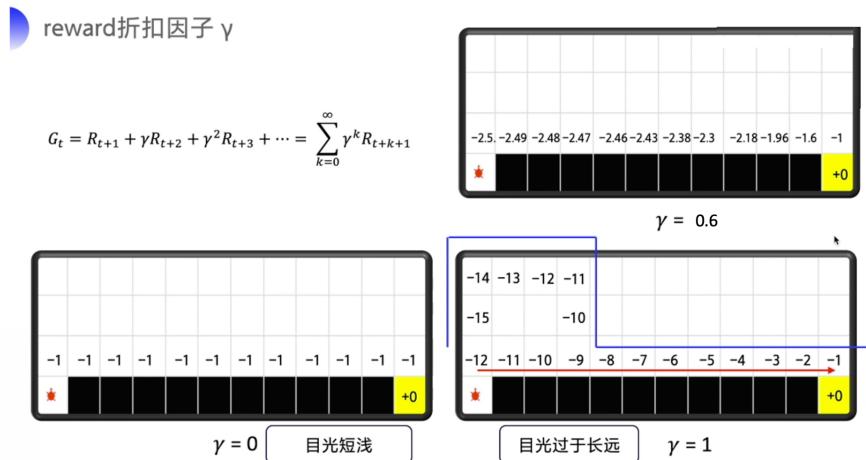


图 3.10

在这个环境当中，我们怎么去计算状态动作价值（未来的总收益）。如图 3.10 所示，

- 如果 $\gamma = 0$ ，假设我走一条路，并从这个状态出发，在这里选择是向上，这里选择向右。如果 $\gamma = 0$ ，用这个公式去计算的话，它相当于考虑的就是一个单步的收益。我们可以认为它是一个目光短浅的计算的方法。
- 如果 $\gamma = 1$ ，那就等于是说把后续所有的收益都全部加起来。在这里悬崖问题，你每走一步都会拿到一个 -1 分的 reward，只有到了终点之后，它才会停止。如果 $\gamma = 1$ 的话，我们用这个公式去计算，就这里是 -1。然后这里的话，未来的总收益就是 $-1 + -1 = -2$ 。
- 如果 $\gamma = 0.6$ ，就是目光没有放得那么的长远，计算出来是这个样子的。利用 $G_t = R_{t+1} + \gamma G_{t+1}$ 这个公式从后往前推。

$$\begin{aligned}
 G_7 &= R + \gamma G_8 = -1 + 0.6 * (-2.176) = -2.3056 \approx -2.3 \\
 G_8 &= R + \gamma G_9 = -1 + 0.6 * (-1.96) = -2.176 \approx -2.18 \\
 G_9 &= R + \gamma G_{10} = -1 + 0.6 * (-1.6) = -1.96 \\
 G_{10} &= R + \gamma G_{11} = -1 + 0.6 * (-1) = -1.6 \\
 G_{12} &= R + \gamma G_{13} = -1 + 0.6 * 0 = -1 \\
 G_{13} &= 0
 \end{aligned} \tag{3.1}$$

这里的计算是我们选择了一条路，计算出这条路径上每一个状态动作的价值。我们可以看一下右下角这个图，如果说我走的不是红色的路，而是蓝色的路，那我算出来的 Q 值可能是这样。那我们就知道，当小乌龟在 -12 这个点的时候，往右边走是 -11，往上走是 -15，它自然就知道往右走的价值更大，小乌龟就会往右走。

状态	上	下	左	右
坐标 (1, 1)	0	0	0	0
坐标 (1, 2)	0	0	0	0
坐标 (1, 3)	0	0	0	0
坐标 (1, 4)	0	0	0	0
坐标 (1, 5)	0	0	0	0
坐标 (1, 6)	0	0	0	0
...

图 3.11

类似于上图，最后我们要求解的就是一张 Q 表格，

- 它的行数是所有的状态数量，一般可以用坐标来表示表示格子的状态，也可以用 1、2、3、4、5、6、7 来表示不同的位置。
- Q 表格的列表示上下左右四个动作。

最开始这张 Q 表格会全部初始化为零，然后 agent 会不断地去和环境交互得到不同的轨迹，当交互的次数足够多的时候，我们就可以估算出每一个状态下，每个行动的平均总收益去更新这个 Q 表格。怎么去更新 Q 表格就是接下来要引入的强化概念。

强化就是我们可以用下一个状态的价值来更新当前状态的价值，其实就是强化学习里面 bootstrapping 的概念。在强化学习里面，你可以每走一步更新一下 Q 表格，然后用下一个状态的 Q 值来更新这个状态的 Q 值，这种单步更新的方法叫做时序差分。

3.3 Model-free Prediction

在没法获取 MDP 的模型情况下，我们可以通过以下两种方法来估计某个给定策略的价值：

- Monte Carlo policy evaluation
- Temporal Difference(TD) learning

3.3.1 Monte-Carlo Policy Evaluation

- 蒙特卡罗 (Monte-Carlo, MC) 方法是基于采样的方法，给定策略 π ，我们让 agent 跟环境进行交互，就会得到很多轨迹。每个轨迹都有对应的 return：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \tag{3.2}$$

- 我们把每个轨迹的 return 进行平均，就可以知道某一个策略下面对应状态的价值，即 $v^\pi(s) = \mathbb{E}_{\tau \sim \pi}[G_t | s_t = s]$ 。
- MC 仿真：我们可以采样大量的轨迹，计算所有轨迹的真实回报，然后进行平均。
- MC 是用 [经验平均回报 \(empirical mean return\)](#) 的方法来估计。
- MC 方法不需要 MDP 的转移函数和奖励函数，并且不需要像动态规划那样用 bootstrapping 的方法。
- MC 的局限性：只能用在有终止的 MDP。

接下来，我们总结下 MC 算法。为了得到评估 $v(s)$ ，我们进行了如下的步骤：

1. 在每个回合中，如果在时间步 t 状态 s 被访问了，那么

- 状态 s 的访问数 $N(s)$ 增加 1， $N(s) \leftarrow N(s) + 1$ 。
- 状态 s 的总的回报 $S(s)$ 增加 G_t ， $S(s) \leftarrow S(s) + G_t$ 。

2. 状态 s 的价值可以通过 return 的平均来估计，即 $v(s) = S(s)/N(s)$ 。

根据大数定律，只要我们得到足够多的轨迹，就可以趋近这个策略对应的价值函数。也就是说，当 $N(s) \rightarrow \infty$ 时， $v(s) \rightarrow v^\pi(s)$ 。

假设现在有样本 x_1, x_2, \dots ，我们可以把经验均值 (empirical mean) 转换成 [增量均值 \(incremental mean\)](#) 的形式，如下式所示：

$$\begin{aligned} \mu_t &= \frac{1}{t} \sum_{j=1}^t x_j \\ &= \frac{1}{t} \left(x_t + \sum_{j=1}^{t-1} x_j \right) \\ &= \frac{1}{t} (x_t + (t-1)\mu_{t-1}) \\ &= \frac{1}{t} (x_t + t\mu_{t-1} - \mu_{t-1}) \\ &= \mu_{t-1} + \frac{1}{t} (x_t - \mu_{t-1}) \end{aligned} \tag{3.3}$$

通过这种转换，我们就可以把上一时刻的平均值跟现在时刻的平均值建立联系，即：

$$\mu_t = \mu_{t-1} + \frac{1}{t} (x_t - \mu_{t-1}) \tag{3.4}$$

其中：

- $x_t - \mu_{t-1}$ 是残差
- $\frac{1}{t}$ 类似于学习率 (learning rate)

当我们得到 x_t ，就可以用上一时刻的值来更新现在的值。

我们可以把 Monte-Carlo 更新的方法写成 incremental MC 的方法：我们采集数据，得到一个新的轨迹 $(S_1, A_1, R_1, \dots, S_t)$ 。对于这个轨迹，我们采用增量的方法进行更新，如下式所示：

$$\begin{aligned} N(S_t) &\leftarrow N(S_t) + 1 \\ v(S_t) &\leftarrow v(S_t) + \frac{1}{N(S_t)} (G_t - v(S_t)) \end{aligned} \tag{3.5}$$

如下式所示，我们可以直接把 $\frac{1}{N(S_t)}$ 变成 α (学习率)， α 代表着更新的速率有多快，我们可以进行设置。

$$v(S_t) \leftarrow v(S_t) + \alpha (G_t - v(S_t)) \tag{3.6}$$

我们再来看一下 DP 和 MC 方法的差异。

- 动态规划也是常用的估计价值函数的方法。在动态规划里面，我们使用了 bootstrapping 的思想。bootstrapping 的意思就是我们基于之前估计的量来估计一个量。

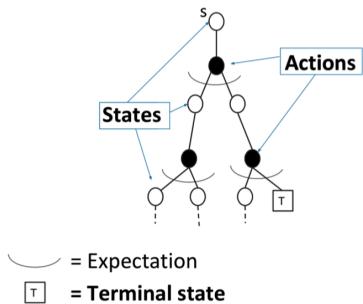


图 3.12

- DP 就是用 Bellman expectation backup，就是通过上一时刻的值 $v_{i-1}(s')$ 来更新当前时刻 $v_i(s)$ 这个值，如下式所示：

$$v_i(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a | s) \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' | s, a) v_{i-1}(s') \right) \quad (3.7)$$

不停迭代，最后可以收敛。Bellman expectation backup 就有两层加和，内部加和和外部加和，算了两次 expectation，得到了一个更新。

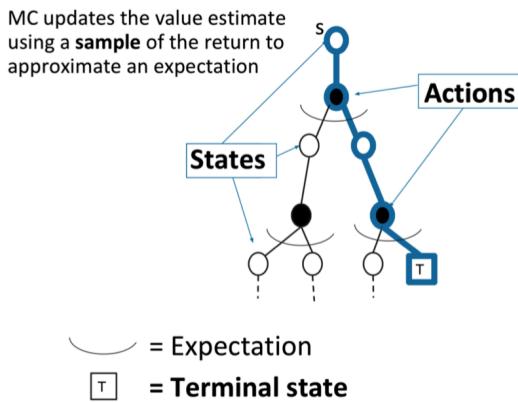


图 3.13

MC 是通过一个回合的 empirical mean return (实际得到的收益) 来进行更新，如下式所示：

$$v(S_t) \leftarrow v(S_t) + \alpha (G_{i,t} - v(S_t)) \quad (3.8)$$

对应树上面蓝色的轨迹，我们得到的是一个实际的轨迹，实际的轨迹上的状态已经是决定的，采取的行为都是决定的。MC 得到的是一条轨迹，这条轨迹表现出来就是这个蓝色的从起始到最后终止状态的轨迹。现在只是更新这个轨迹上的所有状态，跟这个轨迹没有关系的状态都没有更新。

MD 相比 DP 是有一些优势的：

- MC 可以在不知道环境的情况下 work，而 DP 是 model-based。
- MC 只需要更新一条轨迹的状态，而 DP 则是需要更新所有的状态。状态数量很多的时候（比如一百万个，两百万个），DP 这样去迭代的话，速度是非常慢的。这也是 sample-based 的方法 MC 相对于 DP 的优势。

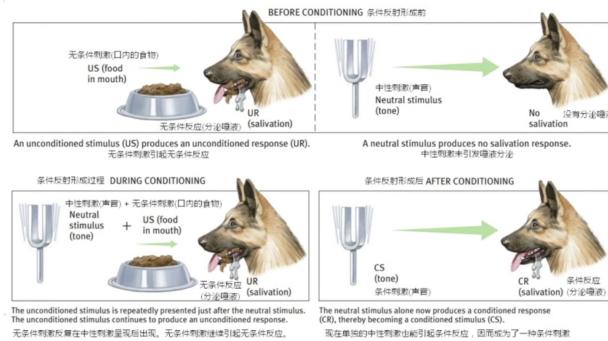


图 3.14 强化概念：巴普洛夫的条件反射实验

3.3.2 Temporal Difference

为了让大家更好地理解时序差分 (Temporal Difference, TD) 这种更新方法，这边给出它的物理意义。我们先理解一下巴普洛夫的条件反射实验，这个实验讲的是小狗会对盆里面的食物无条件产生刺激，分泌唾液。一开始小狗对于铃声这种中性刺激是没有反应的，可是我们把这个铃声和食物结合起来，每次先给它响一下铃，再给它喂食物，多次重复之后，当铃声响起的时候，小狗也会开始流口水。盆里的肉可以认为是强化学习里面那个延迟的 reward，声音的刺激可以认为是有 reward 的那个状态之前的一个状态。多次重复实验之后，最后的这个 reward 会强化小狗对于这个声音的条件反射，它会让小狗知道这个声音代表着有食物，这个声音对于小狗来说也就有了价值，它听到这个声音也会流口水。

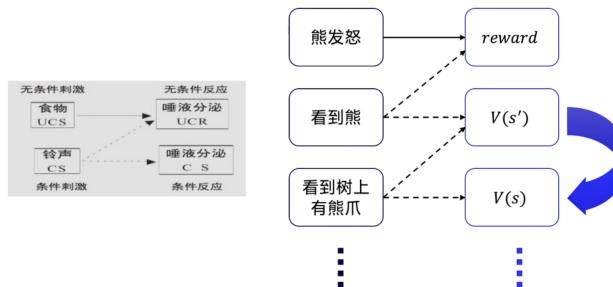


图 3.15

巴普洛夫效应揭示的是中性刺激 (铃声) 跟无条件刺激 (食物) 紧紧挨着反复出现的时候，条件刺激也可以引起无条件刺激引起的唾液分泌，然后形成这个条件刺激。

这种中性刺激跟无条件刺激在时间上面的结合，我们就称之为强化。强化的次数越多，条件反射就会越巩固。小狗本来不觉得铃声有价值的，经过强化之后，小狗就会慢慢地意识到铃声也是有价值的，它可能带来食物。更重要是一种条件反射巩固之后，我们再用另外一种新的刺激和条件反射去结合，还可以形成第二级条件反射，同样地还可以形成第三级条件反射。

在人的身上是可以建立多级的条件反射的，举个例子，比如说一般我们遇到熊都是这样一个顺序：看到树上有熊爪，然后看到熊之后，突然熊发怒，扑过来了。经历这个过程之后，我们可能最开始看到熊才会瑟瑟发抖，后面就是看到树上有熊爪就已经有害怕的感觉了。也就是说在不断的重复试验之后，下一个状态的价值，它是可以不断地去强化影响上一个状态的价值的。

为了让大家更加直观感受下一个状态影响上一个状态（状态价值迭代），我们推荐这个网站：Temporal Difference Learning Gridworld Demo。

- 我们先初始化一下，然后开始时序差分的更新过程。
- 在训练的过程中，你会看到这个小黄球在不断地试错，在探索当中会先迅速地发现有奖励的地方。最开始的时候，只是这些有奖励的格子才有价值。当不断地重复走这些路线的时候，这些有价值的格子

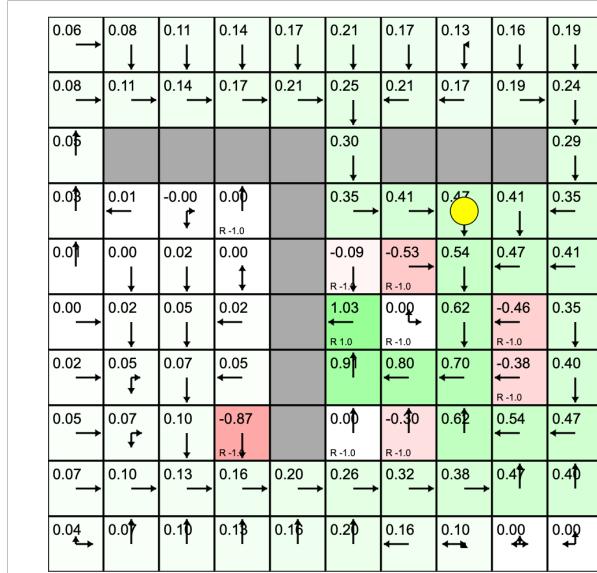


图 3.16

可以去慢慢地影响它附近的格子的价值。

- 反复训练之后，这些有奖励的格子周围的格子的状态就会慢慢地被强化。强化就是当它收敛到最后一个最优的状态了，这些价值最终收敛到一个最优的情况之后，那个小黄球就会自动地知道，就是我一直往价值高的地方走，就能够走到能够拿到奖励的地方。

下面开始正式介绍 TD 方法。

- TD 是介于 MC 和 DP 之间的方法。
- TD 是 model-free 的，不需要 MDP 的转移矩阵和奖励函数。
- TD 可以从不完整的 episode 中学习，结合了 bootstrapping 的思想。
- 接下来，总结下 TD 算法。
- 目的：对于某个给定的策略 π ，在线 (online) 地算出它的价值函数 v^π ，即一步一步地 (step-by-step) 算。
- 最简单的算法是 TD(0)，每往前走一步，就做一步 bootstrapping，用得到的估计回报 (estimated return) $R_{t+1} + \gamma v(S_{t+1})$ 来更新上一时刻的值 $v(S_t)$ 。

$$v(S_t) \leftarrow v(S_t) + \alpha (R_{t+1} + \gamma v(S_{t+1}) - v(S_t)) \quad (3.9)$$

- 估计回报 $R_{t+1} + \gamma v(S_{t+1})$ 被称为 TD target，TD target 是带衰减的未来收益的总和。TD target 由两部分组成：
 - 走了某一步后得到的实际奖励： R_{t+1} ，
 - 我们利用了 bootstrapping 的方法，通过之前的估计来估计 $v(S_{t+1})$ ，然后加了一个折扣系数，即 $\gamma v(S_{t+1})$ ，具体过程如下式所示：

$$\begin{aligned} v(s) &= \mathbb{E}[G_t | s_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t = s] \\ &= \mathbb{E}[R_{t+1} | s_t = s] + \gamma \mathbb{E}[R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots | s_t = s] \quad (3.10) \\ &= R(s) + \gamma \mathbb{E}[G_{t+1} | s_t = s] \\ &= R(s) + \gamma \mathbb{E}[v(s_{t+1}) | s_t = s] \end{aligned}$$

- TD 目标是估计有两个原因：它对期望值进行采样，并且使用当前估计 V 而不是真实 v_π 。
- TD error $\delta = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$ 。

- 可以类比于 Incremental Monte-Carlo 的方法，给定一个回合 i ，我们可以更新 $v(S_t)$ 来逼近真实的回报 G_t ，具体更新公式如下：

$$v(S_t) \leftarrow v(S_t) + \alpha (R_{t+1} + \gamma v(S_{t+1}) - v(S_t)) \quad (3.11)$$

式 (3.11) 体现了强化这个概念。

我们对比下 MC 和 TD：在 MC 里面 $G_{i,t}$ 是实际得到的值（可以看成 target），因为它已经把一条轨迹跑完了，可以算每个状态实际的 return。TD 没有等轨迹结束，往前走了一步，就可以更新价值函数。

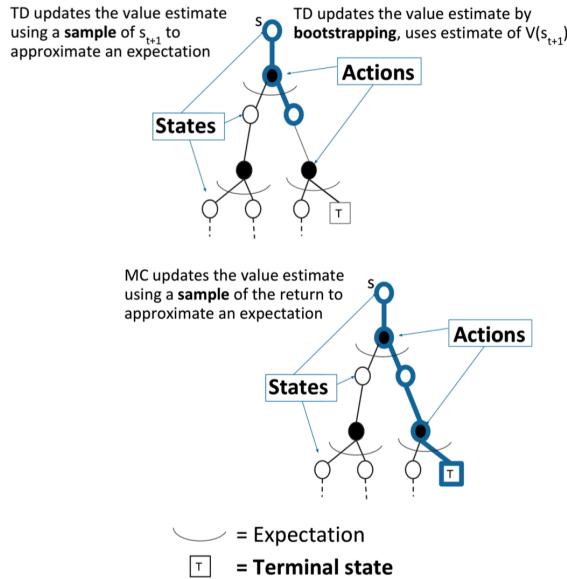


图 3.17 TD 相比 MC 的优势

如图 3.17 所示，TD 只执行了一步，状态的值就更新。MC 全部走完了之后，到了终止状态之后，再更新它的值。

接下来，进一步比较下 TD 和 MC。

- TD 可以在线学习 (online learning)，每走一步就可以更新，效率高。MC 必须等游戏结束才可以学习。
- TD 可以从不完整序列上进行学习。MC 只能从完整的序列上进行学习。
- TD 可以在连续的环境下（没有终止）进行学习。MC 只能在有终止的情况下学习。
- TD 利用了马尔可夫性质，在马尔可夫环境下有更高的学习效率。MC 没有假设环境具有马尔可夫性质，利用采样的价值来估计某一个状态的价值，在不是马尔可夫的环境下更加有效。

举个例子来解释 TD 和 MC 的区别，

TD 是指在不清楚马尔可夫状态转移概率的情况下，以采样的方式得到不完整的状态序列，估计某状态在该状态序列完整后可能得到的收益，并通过不断地采样持续更新价值。MC 则需要经历完整的状态序列后，再来更新状态的真实价值。

例如，你想获得开车去公司的时间，每天上班开车的经历就是一次采样。假设今天在路口 A 遇到了堵车，TD 会在路口 A 就开始更新预计到达路口 B、路口 C ……，以及到达公司的时间；而 MC 并不会立即更新时间，而是在到达公司后，再修改到达每个路口和公司的时间。

TD 能够在知道结果之前就开始学习，相比 MC，其更快速、灵活。

我们可以把 TD 进行进一步的推广。之前是只往前走一步，即 one-step TD，TD(0)。

我们可以调整步数，变成 n-step TD。比如 TD(2)，即往前走两步，然后利用两步得到的 return，使用 bootstrapping 来更新状态的价值。

这样就可以通过 step 来调整这个算法需要多少的实际奖励和 bootstrapping。

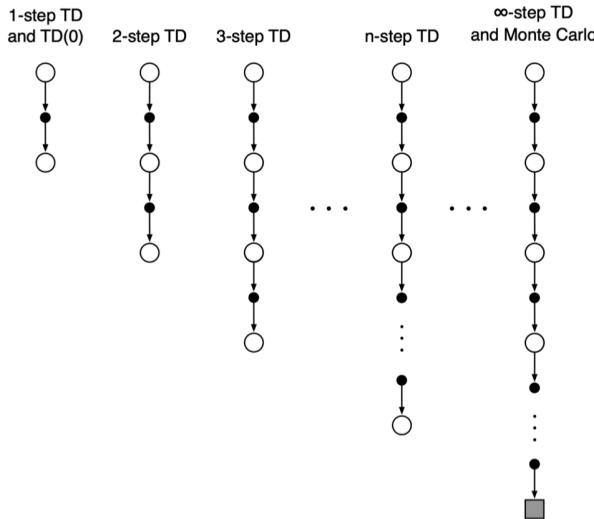


图 3.18 n-step TD

$$\begin{aligned}
 n = 1(TD) \quad G_t^{(1)} &= R_{t+1} + \gamma v(S_{t+1}) \\
 n = 2 \quad \quad \quad G_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 v(S_{t+2}) \\
 &\vdots \\
 n = \infty(MC) \quad G_t^{\infty} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T
 \end{aligned}$$

图 3.19

如上图所示，通过调整步数，可以进行一个 MC 和 TD 之间的 trade-off，如果 $n = \infty$ ，即整个游戏结束过后，再进行更新，TD 就变成了 MC。

n-step 的 TD target 如下式所示：

$$G_t^n = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n}) \quad (3.12)$$

得到 TD target 之后，我们用增量学习 (incremental learning) 的方法来更新状态的价值：

$$v(S_t) \leftarrow v(S_t) + \alpha (G_t^n - v(S_t)) \quad (3.13)$$

3.3.3 Bootstrapping and Sampling for DP, MC and TD

Bootstrapping：更新时使用了估计：

- MC 没用 bootstrapping，因为它是根据实际的 return 来更新。
- DP 用了 bootstrapping。
- TD 用了 bootstrapping。

Sampling：更新时通过采样得到一个期望：

- MC 是纯 sampling 的方法。
- DP 没有用 sampling，它是直接用 Bellman expectation equation 来更新状态价值的。
- TD 用了 sampling。TD target 由两部分组成，一部分是 sampling，一部分是 bootstrapping。

如图 3.20 所示，DP 是直接算 expectation，把它所有相关的状态都进行加和。

如图 3.21 所示，MC 在当前状态下，采一个支路，在一个 path 上进行更新，更新这个 path 上的所有状态。

如图 3.22 所示，TD 是从当前状态开始，往前走了一步，关注的是非常局部的步骤。

$$v(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma v(S_{t+1})]$$

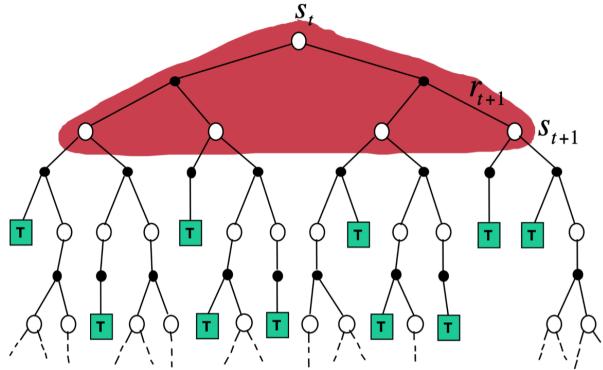


图 3.20 Unified View: Dynamic Programming Backup

$$v(S_t) \leftarrow v(S_t) + \alpha(G_t - v(S_t))$$

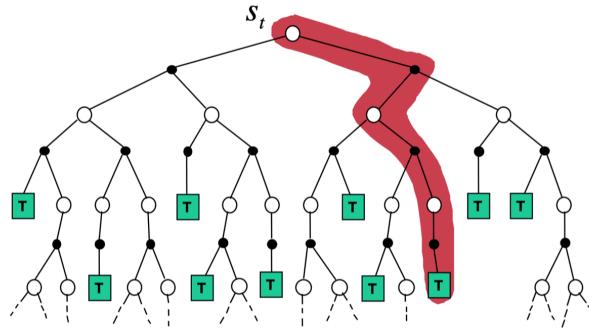


图 3.21 Unified View: Monte-Carlo Backup

如图 3.23 所示, 如果 TD 需要更广度的 update, 就变成了 DP (因为 DP 是把所有状态都考虑进去来进行更新)。如果 TD 需要更深度的 update, 就变成了 MC。右下角是穷举的方法 (exhaustive search), 穷举的方法既需要很深度的信息, 又需要很广度的信息。

3.4 Model-free Control

Q: 当我们不知道 MDP 模型情况下, 如何优化价值函数, 得到最佳的策略?

A: 我们可以把 policy iteration 进行一个广义的推广, 使它能够兼容 MC 和 TD 的方法, 即 [Generalized Policy Iteration\(GPI\) with MC and TD](#)。

Policy iteration 由两个步骤组成:

1. 根据给定的当前的 policy π 来估计价值函数;
2. 得到估计的价值函数后, 通过 greedy 的方法来改进策略, 如下式所示。

$$\pi' = \text{greedy}(v_\pi) \quad (3.14)$$

这两个步骤是一个互相迭代的过程。

$$q_{\pi_i}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' | s, a) v_{\pi_i}(s') \quad (3.15)$$

$$\pi_{i+1}(s) = \arg \max_a q_{\pi_i}(s, a) \quad (3.16)$$

$$TD(0) : v(S_t) \leftarrow v(S_t) + \alpha(R_{t+1} + \gamma v(s_{t+1}) - v(S_t))$$

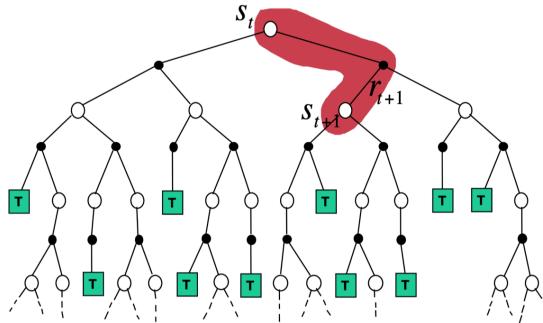


图 3.22 Unified View: Temporal-Difference Backup

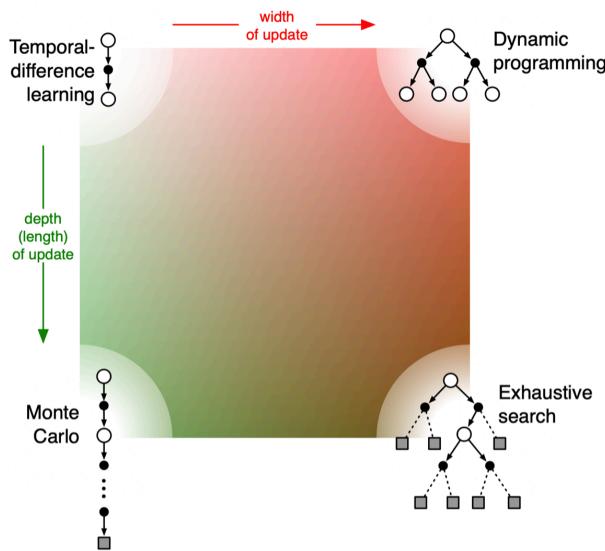


图 3.23 Unified View of Reinforcement Learning

我们可以计算出策略 π 的状态-价值函数如式 (3.15) 所示，并且我们根据式 (3.16) 来计算针对 $s \in \mathcal{S}$ 的新策略 π_{t+1} 。但得到一个状态-价值函数过后，我们并不知道它的奖励函数 $R(s, a)$ 和状态转移 $P(s'|s, a)$ ，所以就没法估计它的 Q 函数。所以这里有一个问题：当我们不知道奖励函数和状态转移时，如何进行策略的优化。

如图 3.25 所示，针对上述情况，我们引入了广义的 policy iteration 的方法。

我们对 policy evaluation 部分进行修改：用 MC 的方法代替 DP 的方法去估计 Q 函数。

当得到 Q 函数后，就可以通过 greedy 的方法去改进它。

如图 3.26 所示，这是 MC 估计 Q 函数的算法。

- 假设每一个 episode 都有一个 exploring start，exploring start 保证所有的状态和动作都在无限步的执行后能被采样到，这样才能很好地去估计。
- 算法通过 MC 的方法产生了很多的轨迹，每个轨迹都可以算出它的价值。然后，我们可以通过 average 的方法去估计 Q 函数。Q 函数可以看成一个 Q-table，通过采样的方法把表格的每个单元的值都填上，然后我们使用 policy improvement 来选取更好的策略。
- 算法核心：如何用 MC 方法来填 Q-table。

如图 3.27 所示，为了确保 MC 方法能够有足够的探索，我们使用了 ε -greedy exploration。

ε -greedy 的意思是说，我们有 $1 - \varepsilon$ 的概率会按照 Q-function 来决定 action，通常 ε 就设一个很小的

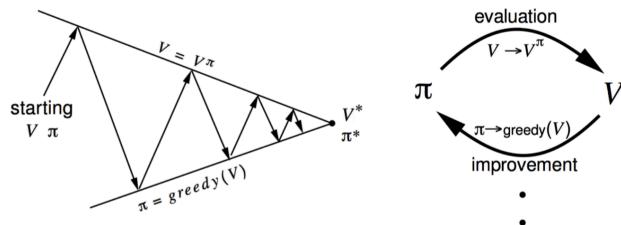
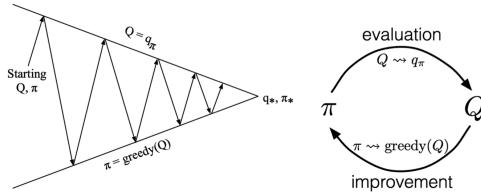


图 3.24 Policy Iteration

Generalized Policy Iteration with Action-Value Function

Monte Carlo version of policy iteration



- ① Policy evaluation: Monte-Carlo policy evaluation $Q = q_\pi$
- ② Policy improvement: Greedy policy improvement?

$$\pi(s) = \arg \max_a q(s, a)$$

图 3.25

值, $1 - \varepsilon$ 可能是 90%, 也就是 90% 的概率会按照 Q-function 来决定 action, 但是你有 10% 的机率是随机的。通常在实现上 ε 会随着时间递减。在最开始的时候。因为还不知道那个 action 是比较好的, 所以你会花比较大的力气在做 exploration。接下来随着训练的次数越来越多。已经比较确定说哪一个 Q 是比较好的。你就会减少你的 exploration, 你会把 ε 的值变小, 主要根据 Q-function 来决定你的 action, 比较少做 random, 这是 ε -greedy。

如图 3.28 所示, 当我们使用 MC 和 ε -greedy 探索这个形式的时候, 我们可以确保价值函数是单调的, 改进的。

带 ε -greedy 探索的 MC 算法的伪代码如图 3.29 所示。

与 MC 相比, TD 有如下几个优势:

- 低方差。

Monte Carlo with Exploring Starts

- ① One assumption to obtain the guarantee of convergence in PI:
Episode has exploring starts
- ② Exploring starts can ensure all actions are selected infinitely often

```
Monte Carlo ES (Exploring Starts), for estimating  $\pi \approx \pi_*$ 

Initialize:
   $\pi(s) \in \mathcal{A}(s)$  (arbitrarily), for all  $s \in \mathcal{S}$ 
   $Q(s, a) \in \mathbb{R}$  (arbitrarily), for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
   $Returns(s, a) \leftarrow$  empty list, for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 

Loop forever (for each episode):
  Choose  $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$  randomly such that all pairs have probability > 0
  Generate an episode from  $S_0, A_0$ , following  $\pi$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
   $G \leftarrow 0$ 
  Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
     $G \leftarrow \gamma G + R_{t+1}$ 
    Unless the pair  $S_t, A_t$  appears in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$ :
      Append  $G$  to  $Returns(S_t, A_t)$ 
       $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$ 
       $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ 
```

图 3.26

Monte Carlo with ϵ -Greedy Exploration

- ① Trade-off between exploration and exploitation (we will talk about this in later lecture)
- ② ϵ -Greedy Exploration: Ensuring continual exploration
 - ① All actions are tried with non-zero probability
 - ② With probability $1 - \epsilon$ choose the greedy action
 - ③ With probability ϵ choose an action at random

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

图 3.27

Monte Carlo with ϵ -Greedy Exploration

- ① Policy improvement theorem: For any ϵ -greedy policy π , the ϵ -greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) \geq v_\pi(s)$

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \\ &= \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \max_a q_\pi(s, a) \\ &\geq \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) + (1 - \epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}|}}{1 - \epsilon} q_\pi(s, a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) = v_\pi(s) \end{aligned}$$

Therefore, $v_{\pi'}(s) \geq v_\pi(s)$ from the policy improvement theorem

图 3.28

- 能够在线学习。
- 能够从不完整的序列学习。

所以我们可以把 TD 也放到 control loop 里面去估计 Q-table, 再采取这个 ϵ -greedy improvement。这样就可以在 episode 没结束的时候来更新已经采集到的状态价值。

偏差 (bias): 描述的是预测值 (估计值) 的期望与真实值之间的差距。偏差越大, 越偏离真实数据, 如图 3.30 第二行所示。
方差 (variance): 描述的是预测值的变化范围, 离散程度, 也就是离其期望值的距离。方差越大, 数据的分布越分散, 如图 3.30 右列所示。

Algorithm 1

```

1: Initialize  $Q(S, A) = 0, N(S, A) = 0, \epsilon = 1, k = 1$ 
2:  $\pi_k = \epsilon$ -greedy( $Q$ )
3: loop
4:   Sample  $k$ -th episode  $(S_1, A_1, R_2, \dots, S_T) \sim \pi_k$ 
5:   for each state  $S_t$  and action  $A_t$  in the episode do
6:      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$ 
7:      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$ 
8:   end for
9:    $k \leftarrow k + 1, \epsilon \leftarrow 1/k$ 
10:   $\pi_k = \epsilon$ -greedy( $Q$ )
11: end loop

```

图 3.29 Monte Carlo with ϵ -Greedy Exploration

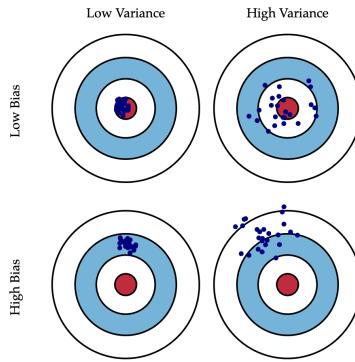


图 3.30

3.4.1 Sarsa: On-policy TD Control

Recall: TD Prediction

- ① An episode consists of an alternating sequence of states and state-action pairs:

$$\dots \rightarrow (S_t, A_t) \xrightarrow{R_{t+1}(S_{t+1})} (S_{t+1}, A_{t+1}) \xrightarrow{R_{t+2}(S_{t+2})} (S_{t+2}, A_{t+2}) \xrightarrow{R_{t+3}(S_{t+3})} (S_{t+3}, A_{t+3}) \dots$$

- ② TD(0) method for estimating the value function $V(S)$

```

 $A_t \leftarrow \text{action given by } \pi \text{ for } S$ 
Take action  $A_t$ , observe  $R_{t+1}$  and  $S_{t+1}$ 
 $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ 

```

- ③ How about estimating action value function $Q(S, A)$?

图 3.31

如图 3.31 所示，TD 是给定了一个策略，然后我们去估计它的价值函数。接着我们要考虑怎么用 TD 这个框架来估计 Q-function。

Sarsa 所作出的改变很简单，就是将原本我们 TD 更新 V 的过程，变成了更新 Q，如下式所示：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (3.17)$$

这个公式就是说可以拿下一步的 Q 值 $Q(S_{t+1}, A_{t+1})$ 来更新我这一步的 Q 值 $Q(S_t, A_t)$ 。

Sarsa 是直接估计 Q-table，得到 Q-table 后，就可以更新策略。

为了理解这个公式，如图 3.32 所示，我们先把 $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ 当作是一个目标值，就是 $Q(S_t, A_t)$ 想要去逼近的一个目标值。 $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ 就是 TD target。

我们想要计算的就是 $Q(S_t, A_t)$ 。因为最开始 Q 值都是随机初始化或者是初始化为零，它需要不断地去逼近它理想中真实的 Q 值 (TD target)， $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$ 就是 TD error。

也就是说，我们拿 $Q(S_t, A_t)$ 来逼近 G_t ，那 $Q(S_{t+1}, A_{t+1})$ 其实就是近似 G_{t+1} 。我就可以用 $Q(S_{t+1}, A_{t+1})$ 近似 G_{t+1} ，然后把 $R_{t+1} + Q(S_{t+1}, A_{t+1})$ 当成目标值。

$Q(S_t, A_t)$ 就是要逼近这个目标值，我们用软更新的方式来逼近。软更新的方式就是每次我只更新一点点， α 类似于学习率。最终的话，Q 值都是可以慢慢地逼近到真实的 target 值。这样我们的更新公式只需要用到当前时刻的 S_t, A_t ，还有拿到的 $R_{t+1}, S_{t+1}, A_{t+1}$ 。

该算法由于每次更新值函数需要知道当前的状态 (state)、当前的动作 (action)、奖励 (reward)、下一步的状态 (state)、下一步的动作 (action)，即 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 这几个值，由此得名 Sarsa 算法。

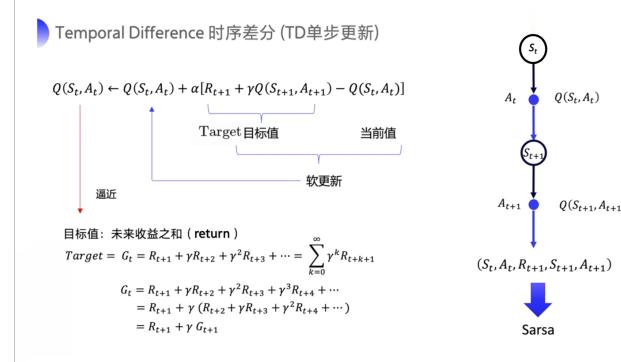


图 3.32

它走了一步之后，拿到了 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 之后，就可以做一次更新。

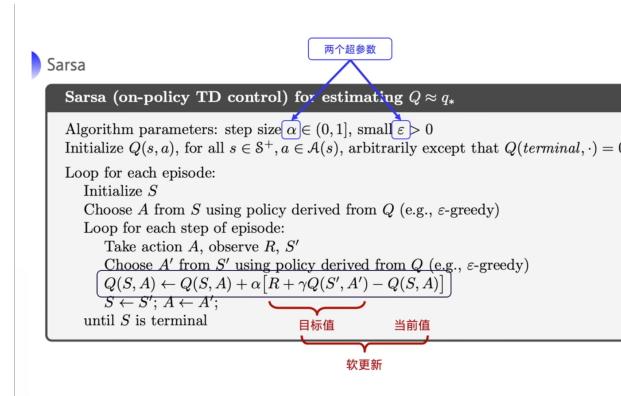


图 3.33

如图 3.33 所示，我们直接看这个框框里面的更新公式，和之前的公式是一样的。 S' 就是 S_{t+1} 。我们就是拿下一步的 Q 值 $Q(S', A')$ 来更新这一步的 Q 值 $Q(S, A)$ ，不断地强化每一个 Q。

Consider the following n -step Q-returns for $n = 1, 2, \infty$

$$\begin{aligned} n = 1(\text{Sarsa}) \quad q_t^{(1)} &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \\ n = 2 \quad q_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A_{t+2}) \\ &\vdots \\ n = \infty(\text{MC}) \quad q_t^{\infty} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T \end{aligned}$$

图 3.34 n-step Sarsa

如图 3.34 所示，Sarsa 属于单步更新法，也就是说每执行一个动作，就会更新一次价值和策略。如果不进行单步更新，而是采取 n 步更新或者回合更新，即在执行 n 步之后再来更新价值和策略，这样就得到了 n 步 Sarsa(n -step Sarsa)。

比如 2-step Sarsa，就是执行两步后再来更新 Q 的值。

具体来说，对于 Sarsa，在 t 时刻其价值的计算公式为

$$q_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \quad (3.18)$$

而对于 n 步 Sarsa，它的 n 步 Q 收获为

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \quad (3.19)$$

如果给 $q_t^{(n)}$ 加上衰减因子 λ 并进行求和，即可得到 Sarsa(λ) 的 Q 收获：

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)} \quad (3.20)$$

因此， n 步 Sarsa(λ) 的更新策略可以表示为

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (q_t^\lambda - Q(S_t, A_t)) \quad (3.21)$$

总的来说，Sarsa 和 Sarsa(λ) 的差别主要体现在价值的更新上。

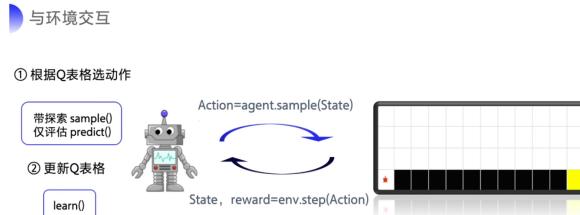


图 3.35

我们看看用代码去怎么去实现。了解单步更新的一个基本公式之后，代码实现就很简单了。右边是环境，左边是 agent。我们每次跟环境交互一次之后呢，就可以 learn 一下，向环境输出 action，然后从环境当中拿到 state 和 reward。Agent 主要实现两个方法：

- 根据 Q 表格去选择动作，输出 action。
- 拿到 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 这几个值去更新我们的 Q 表格。

3.4.2 Q-learning: Off-policy TD Control



图 3.36 Off-policy

Sarsa 是一种 on-policy 策略。Sarsa 优化的是它实际执行的策略，它直接拿下一步会执行的 action 来去优化 Q 表格，所以 on-policy 在学习的过程中，只存在一种策略，它用一种策略去做 action 的选取，也用一种策略去做优化。所以 Sarsa 知道它下一步的动作有可能会跑到悬崖那边去，所以它就会在优化它自己的策略的时候，会尽可能的离悬崖远一点。这样子就会保证说，它下一步哪怕是有随机动作，它也还是在安全区域内。

如图 3.36 所示，而 off-policy 在学习的过程中，有两种不同的策略：

- 第一个策略是我们需要去学习的策略，即 **target policy(目标策略)**，一般用 π 来表示，Target policy 就像是在后方指挥战术的一个军师，它可以根据自己的经验来学习最优的策略，不需要去和环境交互。
- 另外一个策略是探索环境的策略，即 **behavior policy(行为策略)**，一般用 μ 来表示。 μ 可以大胆地去探索到所有可能的轨迹，采集轨迹，采集数据，然后把采集到的数据喂给 target policy 去学习。而

且喂给目标策略的数据中并不需要 A_{t+1} ，而 Sarsa 是要有 A_{t+1} 的。Behavior policy 像是一个战士，可以在环境里面探索所有的动作、轨迹和经验，然后把这些经验交给目标策略去学习。比如目标策略优化的时候，Q-learning 不会管你下一步去往哪里探索，它就只选收益最大的策略。

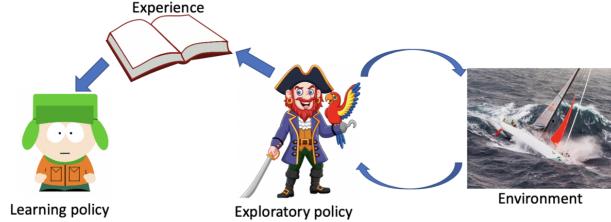


图 3.37

再举个例子。如图 3.37 所示，比如环境是一个波涛汹涌的大海，但 learning policy 太胆小了，没法直接跟环境去学习，所以我们有了 exploratory policy，exploratory policy 是一个不畏风浪的海盗，他非常激进，可以在环境中探索。他有很多经验，可以把这些经验写成稿子，然后喂给这个 learning policy。Learning policy 可以通过这个稿子来进行学习。

在 off-policy learning 的过程中，我们这些轨迹都是 behavior policy 跟环境交互产生的，产生这些轨迹后，我们使用这些轨迹来更新 target policy π 。

Off-policy learning 有很多好处：

- 我们可以利用 exploratory policy 来学到一个最佳的策略，学习效率高；
- 可以让我们学习其他 agent 的行为，模仿学习，学习人或者其他 agent 产生的轨迹；
- 重用老的策略产生的轨迹。探索过程需要很多计算资源，这样的话，可以节省资源。

Q-learning 有两种 policy：behavior policy 和 target policy。

Target policy π 直接在 Q-table 上取 greedy，就取它下一步能得到的所有状态，如式 (3.22) 所示：

$$\pi(S_{t+1}) = \arg \max_{a'} Q(S_{t+1}, a') \quad (3.22)$$

Behavior policy μ 可以是一个随机的 policy，但我们采取 ε -greedy，让 behavior policy 不至于是完全随机的，它是基于 Q-table 逐渐改进的。

我们可以构造 Q-learning target，Q-learning 的 next action 都是通过 arg max 操作来选出来的，于是我们可以代入 arg max 操作，可以得到下式：

$$\begin{aligned} R_{t+1} + \gamma Q(S_{t+1}, A') &= R_{t+1} + \gamma Q(S_{t+1}, \arg \max Q(S_{t+1}, a')) \\ &= R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') \end{aligned} \quad (3.23)$$

接着我们可以把 Q-learning 更新写成增量学习的形式，TD target 就变成 max 的值，即

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (3.24)$$

我们再通过对比的方式来进一步理解 Q-learning。Q-learning 是 off-policy 的时序差分学习方法，Sarsa 是 on-policy 的时序差分学习方法。

- Sarsa 在更新 Q 表格的时候，它用到的 A' 。我要获取下一个 Q 值的时候， A' 是下一个 step 一定会执行的 action。这个 action 有可能是 ε -greedy 方法采样出来的值，也有可能是 max Q 对应的 action，也有可能是随机动作，但这是它实际执行的那个动作。
- 但是 Q-learning 在更新 Q 表格的时候，它用到这个的 Q 值 $Q(S', a)$ 对应的那个 action，它不一定是下一个 step 会执行的实际的 action，因为你下一个实际会执行的那个 action 可能会探索。
- Q-learning 默认的 next action 不是通过 behavior policy 来选取的，Q-learning 直接看 Q-table，取它的 max 的这个值，它是默认 A' 为最优策略选的动作，所以 Q-learning 在学习的时候，不需要传入 A' ，即 A_{t+1} 的值。

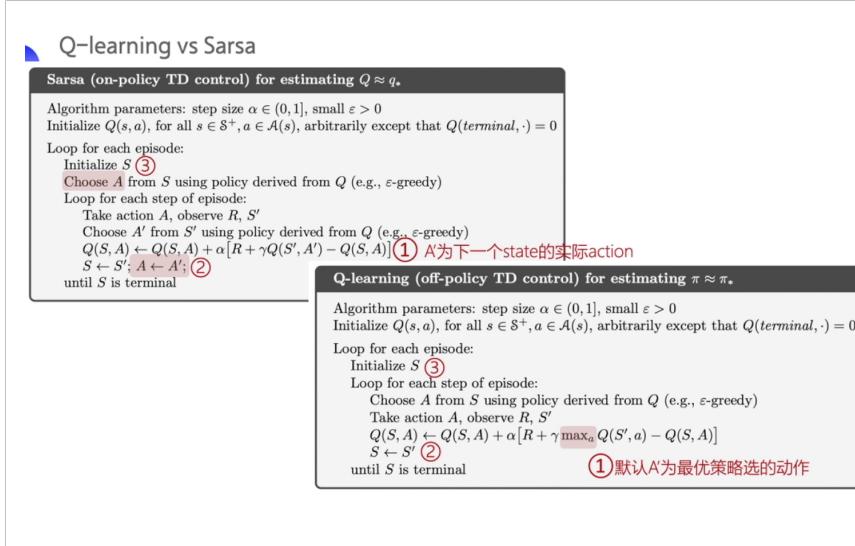


图 3.38

事实上，Q-learning 算法被提出的时间更早，Sarsa 算法是 Q-learning 算法的改进。

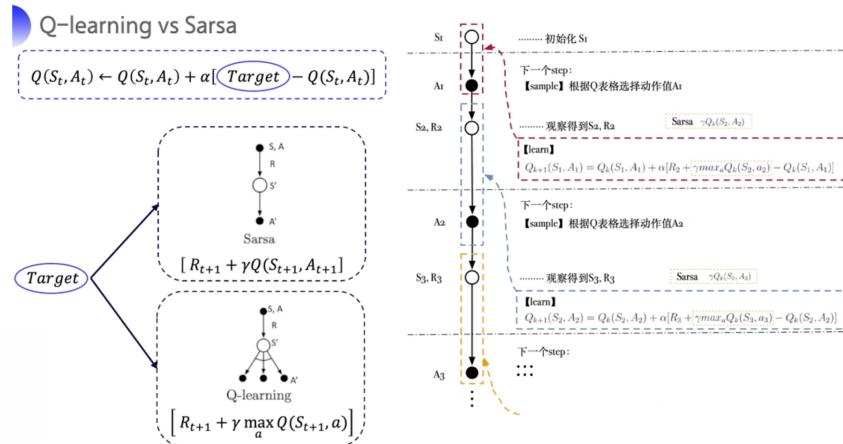


图 3.39

Sarsa 和 Q-learning 的更新公式都是一样的，区别只在 target 计算的这一部分，

- Sarsa 是 $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ ；
- Q-learning 是 $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ 。

Sarsa 是用自己的策略产生了 S, A, R, S', A' 这一条轨迹。然后拿着 $Q(S_{t+1}, A_{t+1})$ 去更新原本的 Q 值 $Q(S_t, A_t)$ 。

但是 Q-learning 并不需要知道我实际上选择哪一个 action，它默认下一个动作就是 Q 最大的那个动作。Q-learning 知道实际上 behavior policy 可能会有 10% 的概率去选择别的动作，但 Q-learning 并不担心受到探索的影响，它默认了就按照最优的策略来去优化目标策略，所以它可以更大胆地去寻找最优的路径，它会表现得比 Sarsa 大胆非常多。

对 Q-learning 进行逐步地拆解的话，跟 Sarsa 唯一一点不一样就是并不需要提前知道 A_2 ，我就能更新 $Q(S_1, A_1)$ 。在训练一个 episode 这个流程图当中，Q-learning 在 learn 之前它也不需要去拿到 next action A' ，它只需要前面四个 (S, A, R, S') ，这跟 Sarsa 很不一样。

3.5 On-policy vs. Off-policy

总结一下 on-policy 和 off-policy 的区别。

- Sarsa 是一个典型的 on-policy 策略，它只用了一个 policy π ，它不仅使用策略 π 学习，还使用策略 π 与环境交互产生经验。如果 policy 采用 ϵ -greedy 算法的话，它需要兼顾探索，为了兼顾探索和利用，它训练的时候会显得有点胆小。它在解决悬崖问题的时候，会尽可能地离悬崖边上远远的，确保说哪怕自己不小心探索了一点，也还是在安全区域内。此外，因为采用的是 ϵ -greedy 算法，策略会不断改变 (ϵ 会不断变小)，所以策略不稳定。
- Q-learning 是一个典型的 off-policy 的策略，它有两种策略：target policy 和 behavior policy。它分离了目标策略跟行为策略。Q-learning 就可以大胆地用 behavior policy 去探索得到的经验轨迹来优化目标策略，从而更有可能去探索到最优的策略。Behavior policy 可以采用 ϵ -greedy 算法，但 target policy 采用的是 greedy 算法，直接根据 behavior policy 采集到的数据来采用最优策略，所以 Q-learning 不需要兼顾探索。
- 比较 Q-learning 和 Sarsa 的更新公式可以发现，Sarsa 并没有选取最大值的 max 操作，因此，
 - Q-learning 是一个非常激进的算法，希望每一步都获得最大的利益；
 - 而 Sarsa 则相对非常保守，会选择一条相对安全的迭代路线。

3.6 Summary

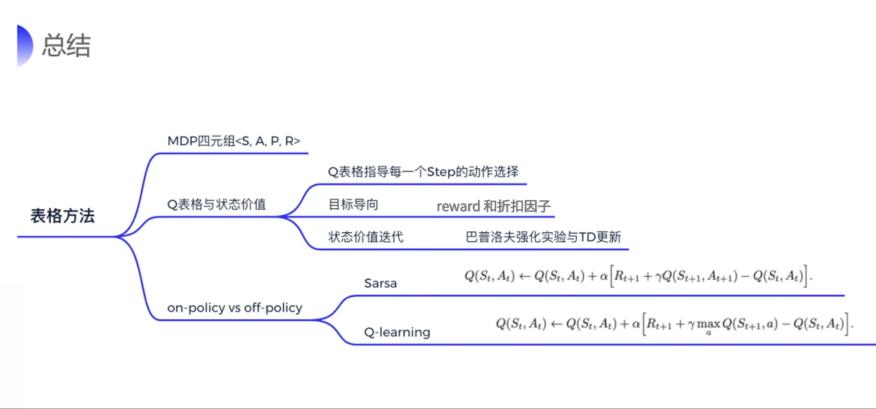


图 3.40

总结如图 3.40 所示。

3.7 Keywords

- P 函数和 R 函数：P 函数反应的是状态转移的概率，即反应的环境的随机性，R 函数就是 Reward function。但是我们通常处于一个未知的环境（即 P 函数和 R 函数是未知的）。
- Q 表格型表示方法：表示形式是一种表格形式，其中横坐标为 action (agent) 的行为，纵坐标是环境的 state，其对应着每一个时刻 agent 和环境的情况，并通过对对应的 reward 反馈去做选择。一般情况下，Q 表格是一个已经训练好的表格，不过，我们也可以每进行一步，就更新一下 Q 表格，然后用下一个状态的 Q 值来更新这个状态的 Q 值（即时序差分方法）。
- 时序差分 (Temporal Difference)：一种 Q 函数 (Q 值) 的更新方式，也就是可以拿下一步的 Q 值 $Q(S_{t+1}, A_{t+1})$ 来更新我这一步的 Q 值 $Q(S_t, A_t)$ 。完整的计算公式如下： $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
- SARSA 算法：一种更新前一时刻状态的单步更新的强化学习算法，也是一种 on-policy 策略。该算法由于每次更新值函数需要知道前一步的状态 (state)，前一步的动作 (action)、奖励 (reward)、当前

状态 (state)、将要执行的动作 (action)，即 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 这几个值，所以被称为 SARSA 算法。agent 每进行一次循环，都会用 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 对于前一步的 Q 值 (函数) 进行一次更新。

3.8 Questions

- 构成强化学习 MDP 的四元组有哪些变量？

答：状态、动作、状态转移概率和奖励，分别对应 (S, A, P, R)，后面有可能会加上个衰减因子构成五元组。

- 基于以上的描述所构成的强化学习的“学习”流程。

答：强化学习要像人类一样去学习了，人类学习的话就是一条路一条路的去尝试一下，先走一条路，我看看结果到底是什么。多试几次，只要能一直走下去的，我们其实可以慢慢的了解哪个状态会更好。我们用价值函数 $V(s)$ 来代表这个状态是好的还是坏的。然后用这个 Q 函数来判断说在什么状态下做什么动作能够拿到最大奖励，我们用 Q 函数来表示这个状态-动作值。

- 基于 SARSA 算法的 agent 的学习过程。

答：我们现在有环境，有 agent。每交互一次以后，我们的 agent 会向环境输出 action，接着环境会反馈给 agent 当前时刻的 state 和 reward。那么 agent 此时会实现两个方法：

- 使用已经训练好的 Q 表格，对应环境反馈的 state 和 reward 选取对应的 action 进行输出。
- 我们已经拥有了 $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ 这几个值，并直接使用 A_{t+1} 去更新我们的 Q 表格。

- Q-learning 和 Sarsa 的区别？

答：Sarsa 算法是 Q-learning 算法的改进。（这句话出自「神经网络与深度学习」的第 342 页）（可参考 SARSA 「on-line q-learning using connectionist systems」的 abstract 部分）

- 首先，Q-learning 是 off-policy 的时序差分学习方法，Sarsa 是 on-policy 的时序差分学习方法。
- 其次，Sarsa 在更新 Q 表格的时候，它用到的 A' 。我要获取下一个 Q 值的时候， A' 是下一个 step 一定会执行的 action。这个 action 有可能是 ϵ -greedy 方法 sample 出来的值，也有可能是 max Q 对应的 action，也有可能是随机动作。但是就是它实实在在执行了的那个动作。
- 但是 Q-learning 在更新 Q 表格的时候，它用到这个的 Q 值 $Q(S', a')$ 对应的那个 action，它不一定是下一个 step 会执行的实际的 action，因为你下一个实际会执行的那个 action 可能会探索。Q-learning 默认的 action 不是通过 behavior policy 来选取的，它是默认 A' 为最优策略选的动作，所以 Q-learning 在学习的时候，不需要传入 A' ，即 a_{t+1} 的值。
- 更新公式的对比（区别只在 target 计算这一部分）：
 - Sarsa 的公式： $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ ；
 - Q-learning 的公式： $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$

Sarsa 实际上都是用自己的策略产生了 S,A,R,S',A' 这一条轨迹。然后拿着 $Q(S_{t+1}, A_{t+1})$ 去更新原本的 Q 值 $Q(S_t, A_t)$ 。但是 Q-learning 并不需要知道，我实际上选择哪一个 action，它默认下一个动作就是 Q 最大的那个动作。所以基于此，Sarsa 的 action 通常会更加“保守”、“胆小”，而对应的 Q-Learning 的 action 会更加“莽撞”、“激进”。

- On-policy 和 off-policy 的区别？

答：

- Sarsa 就是一个典型的 on-policy 策略，它只用一个 π ，为了兼顾探索和利用，所以它训练的时候会显得有点胆小怕事。它在解决悬崖问题的时候，会尽可能地离悬崖边上远远的，确保说哪怕自己不小心探索了一点，也还是在安全区域内不至于跳进悬崖。
- Q-learning 是一个比较典型的 off-policy 的策略，它有目标策略 target policy，一般用 π 来表示。然后还有行为策略 behavior policy，用 μ 来表示。它分离了目标策略跟行为策略。Q-learning 就可以大胆地用 behavior policy 去探索得到的经验轨迹来去优化我的目标策略。这样子我更有

可能去探索到最优的策略。

3. 比较 Q-learning 和 Sarsa 的更新公式可以发现，Sarsa 并没有选取最大值的 max 操作。因此，Q-learning 是一个非常激进的算法，希望每一步都获得最大的利益；而 Sarsa 则相对非常保守，会选择一条相对安全的迭代路线。

3.9 Something About Interview

- 高冷的面试官：同学，你能否简述 on-policy 和 off-policy 的区别？

答：off-policy 和 on-policy 的根本区别在于生成样本的 policy 和参数更新时的 policy 是否相同。对于 on-policy，行为策略和要优化的策略是一个策略，更新了策略后，就用该策略的最新版本对于数据进行采样；对于 off-policy，使用任意的一个行为策略来对于数据进行采样，并利用其更新目标策略。如果举例来说，Q-learning 在计算下一状态的预期收益时使用了 max 操作，直接选择最优动作，而当前 policy 并不一定能选择到最优的 action，因此这里生成样本的 policy 和学习时的 policy 不同，所以 Q-learning 为 off-policy 算法；相对应的 SARAS 则是基于当前的 policy 直接执行一次动作选择，然后用这个样本更新当前的 policy，因此生成样本的 policy 和学习时的 policy 相同，所以 SARAS 算法为 on-policy 算法。

- 高冷的面试官：小同学，能否讲一下 Q-Learning，最好可以写出其 $Q(s, a)$ 的更新公式。另外，它是 on-policy 还是 off-policy，为什么？

答：Q-learning 是通过计算最优动作值函数来求策略的一种时序差分的学习方法，其更新公式为：

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

其是 off-policy 的，由于是 Q 更新使用了下一个时刻的最大值，所以我们只关心哪个动作使得 $Q(s_{t+1}, a)$ 取得最大值，而实际到底采取了哪个动作（行为策略），并不关心。这表明优化策略并没有用到行为策略的数据，所以说它是 off-policy 的。

- 高冷的面试官：小朋友，能否讲一下 SARSA，最好可以写出其 $Q(s, a)$ 的更新公式。另外，它是 on-policy 还是 off-policy，为什么？

答：SARSA 可以算是 Q-learning 的改进（这句话出自「神经网络与深度学习」的第 342 页）（可参考 SARSA 「on-line q-learning using connectionist systems」的 abstract 部分），其更新公式为：

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \gamma Q(s', a') - Q(s, a)]$$

其为 on-policy 的，SARSA 必须执行两次动作得到 (s, a, r, s', a') 才可以更新一次；而且 a' 是在特定策略 π 的指导下执行的动作，因此估计出来的 $Q(s, a)$ 是在该策略 π 之下的 Q-value，样本生成用的 π 和估计的 π 是同一个，因此是 on-policy。

- 高冷的面试官：请问 value-based 和 policy-based 的区别是什么？

答：

1. 生成 policy 上的差异：前者随机，后者确定。Value-Based 中的 action-value 估计值最终会收敛到对应的 true values（通常是不同的有限数，可以转化为 0 到 1 之间的概率），因此通常会获得一个确定的策略（deterministic policy）；而 Policy-Based 不会收敛到一个确定性的值，另外他们会趋向于生成 optimal stochastic policy。如果 optimal policy 是 deterministic 的，那么 optimal action 对应的性能函数将远大于 suboptimal actions 对应的性能函数，性能函数的大小代表了概率的大小。
2. 动作空间是否连续，前者离散，后者连续。Value-Based，对于连续动作空间问题，虽然可以将动作空间离散化处理，但离散间距的选取不易确定。过大的离散间距会导致算法取不到最优 action，会在这附近徘徊，过小的离散间距会使得 action 的维度增大，会和高维度动作空间一样导致维度灾难，影响算法的速度；而 Policy-Based 适用于连续的动作空间，在连续的动作空间中，可以不用计算每个动作的概率，而是通过 Gaussian distribution（正态分布）选择 action。

3. value-based, 例如 Q-learning, 是通过求解最优值函数间接的求解最优策略; policy-based, 例如 REINFORCE, Monte-Carlo Policy Gradient, 等方法直接将策略参数化, 通过策略搜索, 策略梯度或者进化方法来更新策略的参数以最大化回报。基于值函数的方法不易扩展到连续动作空间, 并且当同时采用非线性近似、自举和离策略时会有收敛性问题。策略梯度具有良好的收敛性证明。
4. 补充: 对于值迭代和策略迭代: 策略迭代。它有两个循环, 一个是在策略估计的时候, 为了求当前策略的值函数需要迭代很多次。另外一个是外面的大循环, 就是策略评估, 策略提升这个循环。值迭代算法则是一步到位, 直接估计最优值函数, 因此没有策略提升环节。
- 高冷的面试官: 请简述以下时序差分 (Temporal Difference, TD) 算法。
答: TD 算法是使用广义策略迭代来更新 Q 函数的方法, 核心使用了自举 (bootstrapping), 即值函数的更新使用了下一个状态的值函数来估计当前状态的值。也就是使用下一步的 Q 值 $Q(S_{t+1}, A_{t+1})$ 来更新我这一步的 Q 值 $Q(S_t, A_t)$ 。完整的计算公式如下:
- $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})]$$
- 高冷的面试官: 请问蒙特卡洛方法 (Monte Carlo Algorithm, MC) 和时序差分 (Temporal Difference, TD) 算法是无偏估计吗? 另外谁的方法更大呢? 为什么呢?
答: 蒙特卡洛方法 (MC) 是无偏估计, 时序差分 (TD) 是有偏估计; MC 的方差较大, TD 的方差较小, 原因在于 TD 中使用了自举 (bootstrapping) 的方法, 实现了基于平滑的效果, 导致估计的值函数的方差更小。
 - 高冷的面试官: 能否简单说下动态规划、蒙特卡洛和时序差分的异同点?
答:
 - 相同点: 都用于进行值函数的描述与更新, 并且所有方法都是基于对未来事件的展望来计算一个回溯值。
 - 不同点: 蒙特卡洛和 TD 算法隶属于 model-free, 而动态规划属于 model-based; TD 算法和蒙特卡洛的方法, 因为都是基于 model-free 的方法, 因而对于后续状态的获知也都是基于试验的方法; TD 算法和动态规划的策略评估, 都能基于当前状态的下一步预测情况来得到对于当前状态的值函数的更新。另外, TD 算法不需要等到实验结束后才能进行当前状态的值函数的计算与更新, 而蒙特卡洛的方法需要试验交互, 产生一整条的马尔科夫链并直到最终状态才能进行更新。TD 算法和动态规划的策略评估不同之处为 model-free 和 model-based 这一点, 动态规划可以凭借已知转移概率就能推断出来后续的状态情况, 而 TD 只能借助试验才能知道。
 - 蒙特卡洛方法和 TD 方法的不同在于, 蒙特卡洛方法进行完整的采样来获取了长期的回报值, 因而在价值估计上会有着更小的偏差, 但是也正因为收集了完整的信息, 所以价值的方差会更大, 原因在于毕竟基于试验的采样得到, 和真实的分布还是有差距, 不充足的交互导致的较大方差。而 TD 算法与其相反, 因为只考虑了前一步的回报值其他都是基于之前的估计值, 因而估计具有偏差但方差较小。
 - 三者的联系: 对于 $TD(\lambda)$ 方法, 如果 $\lambda = 0$, 那么此时等价于 TD, 即只考虑下一个状态; 如果 $\lambda = 1$, 等价于 MC, 即考虑 $T - 1$ 个后续状态即到整个 episode 序列结束。

3.10 Solve CliffWalking with Q-learning

强化学习在运动规划方面也有很大的应用前景, 具体包括路径规划与决策, 群体派单等等, 本次项目就将单体运动规划抽象并简化, 让大家初步认识到强化学习在这方面的应用。在运动规划方面, 其实已有很多适用于强化学习的仿真环境, 小到迷宫, 大到贴近真实的自动驾驶环境CARLA, 对这块感兴趣的童鞋可以再多搜集一点。本项目采用 gym 开发的 [CliffWalking-v0](#) 环境, 在上面实现一个简单的 Q-learning 入门 demo。

3.10.1 CliffWalking-v0 环境简介

首先对该环境做一个简介，该环境中文名称叫悬崖寻路问题（CliffWalking），是指在一个 4×12 的网格中，智能体以网格的左下角位置为起点，以网格的下角位置为终点，目标是移动智能体到达终点位置，智能体每次可以在上、下、左、右这 4 个方向中移动一步，每移动一步会得到 -1 单位的奖励。



图 3.41

如图 3.41 所示，红色部分表示悬崖，数字代表智能体能够观测到的位置信息，即 observation，总共会有 0-47 等 48 个不同的值，智能体再移动中会有以下限制：

- 智能体不能移出网格，如果智能体想执行某个动作移出网格，那么这一步智能体不会移动，但是这个操作依然会得到 -1 单位的奖励
- 如果智能体“掉入悬崖”，会立即回到起点位置，并得到 -100 单位的奖励
- 当智能体移动到终点时，该回合结束，该回合总奖励为各步奖励之和

实际的仿真界面如图 3.42 所示：

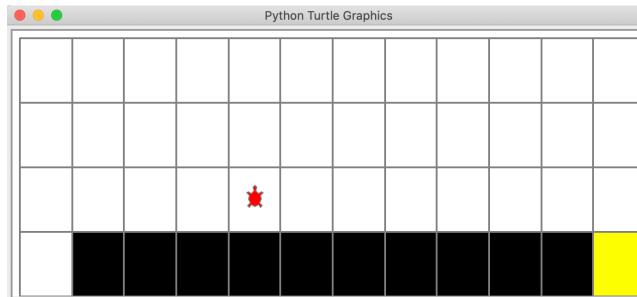


图 3.42

由于从起点到终点最少需要 13 步，每步得到 -1 的 reward，因此最佳训练算法下，每个 episode 下 reward 总和应该为 -13。所以我们的目标也是要通过 RL 训练出一个模型，使得该模型能在测试中一个 episode 的 reward 能够接近于 -13 左右。

3.10.2 RL 基本训练接口

以下是强化学习算法的基本接口，也就是一个完整的上层训练模式，首先是初始化环境和智能体，然后每个 episode 中，首先 agent 选择 action 给到环境，然后环境反馈出下一个状态和 reward，然后 agent 开始更新或者学习，如此多个 episode 之后 agent 开始收敛并保存模型。其中可以通过可视化 reward 随每个 episode 的变化来查看训练的效果。另外由于强化学习的不稳定性，在收敛的状态下也可能会有起伏的情况，此时可以使用滑动平均的 reward 让曲线更加平滑便于分析。

```
''' 初始化环境 '''
env = gym.make("CliffWalking-v0") # 0 up, 1 right, 2 down, 3 left
```

```

env = CliffWalkingWapper(env)
agent = QLearning(
    obs_dim=env.observation_space.n,
    action_dim=env.action_space.n,
    learning_rate=cfg.policy_lr,
    gamma=cfg.gamma,
    epsilon_start=cfg.epsilon_start,epsilon_end=cfg.epsilon_end,epsilon_decay=cfg.epsilon_decay
)
render = False # 是否打开GUI画面
rewards = [] # 记录所有episode的reward
MA_rewards = [] # 记录滑动平均的reward
steps = []# 记录所有episode的steps
for i_episode in range(1,cfg.max_episodes+1):
    ep_reward = 0 # 记录每个episode的reward
    ep_steps = 0 # 记录每个episode走了多少step
    obs = env.reset() # 重置环境，重新开一局（即开始新的一个episode）
    while True:
        action = agent.sample(obs) # 根据算法选择一个动作
        next_obs, reward, done, _ = env.step(action) # 与环境进行一个交互
        # 训练 Q-learning 算法
        agent.learn(obs, action, reward, next_obs, done) # 不需要下一步的action
        obs = next_obs # 存储上一个观察值
        ep_reward += reward
        ep_steps += 1 # 计算step数
        if render:
            env.render() # 渲染新的一帧图形
        if done:
            break
    steps.append(ep_steps)
    rewards.append(ep_reward)
    # 计算滑动平均的reward
    if i_episode == 1:
        MA_rewards.append(ep_reward)
    else:
        MA_rewards.append(
            0.9*MA_rewards[-1]+0.1*ep_reward)
    print('Episode %s: steps = %s , reward = %.1f, explore = %.2f' % (i_episode, ep_steps,
        ep_reward,agent.epsilon))
agent.save() # 训练结束，保存模型

```

3.10.3 任务要求

基于以上目标，本次任务即使训练并绘制 reward 以及滑动平均后的 reward 随 episode 的变化曲线图并记录超参数写成报告，示例如图 3.43 所示。

3.10.4 主要代码清单

main.py：保存强化学习基本接口，以及相应的超参数，可使用 argparse

model.py：保存神经网络，比如全连接网络

agent.py：保存算法模型，主要包含 predict(预测动作) 和 learn 两个函数

params.py：保存一些参数，比如训练参数等

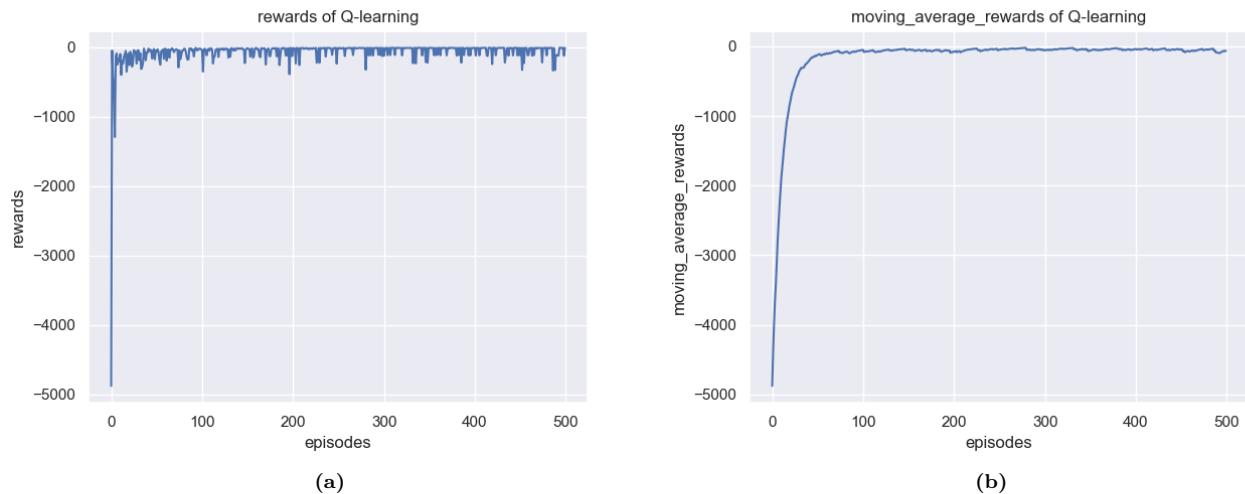


图 3.43

plot.py: 保存相关绘制函数

参考代码

3.10.5 备注

- 注意 ϵ -greedy 策略的使用，以及相应的参数 ϵ 如何衰减
- 训练模型和测试模型的时候选择动作有一些不同，训练时采取 ϵ -greedy 策略，而测试时直接选取 Q 值最大对应的动作，所以算法在动作选择的时候会包括 sample(训练时的动作采样) 和 predict(测试时的动作选择)
- Q 值最大对应的动作可能不止一个，此时可以随机选择一个输出结果

References

- 百度强化学习
- 强化学习基础 David Silver 笔记
- Intro to Reinforcement Learning (强化学习纲要)
- Reinforcement Learning: An Introduction (second edition)
- 百面深度学习
- 神经网络与深度学习
- 机器学习
- Understanding the Bias-Variance Tradeoff

第 4 章 Policy Gradient

4.1 Policy Gradient

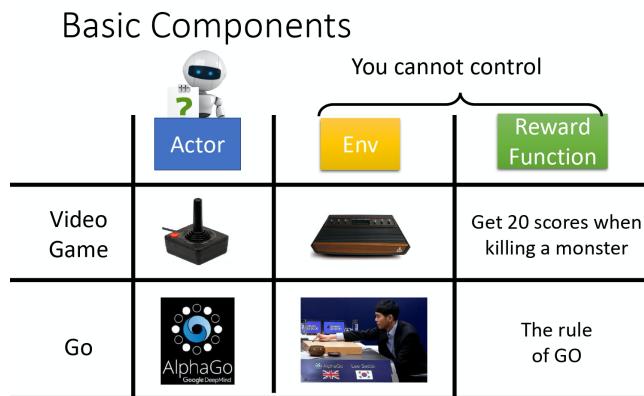


图 4.1

在强化学习中有 3 个组成部分：演员 (actor)、环境 (environment) 和 奖励函数 (reward function)。

让机器玩视频游戏时，

- 演员做的事情就是去操控游戏的摇杆，比如说向左、向右、开火等操作；
- 环境就是游戏的主机，负责控制游戏的画面，负责控制怪物要怎么移动，你现在要看到什么画面等等；
- 奖励函数就是当你做什么事情，发生什么状况的时候，你可以得到多少分数，比如说杀一只怪兽得到 20 分等等。

同样的概念用在围棋上也是一样的，

- 演员就是 Alpha Go，它要决定下哪一个位置；
- 环境就是对手；
- 奖励函数就是按照围棋的规则，赢就是得一分，输就是负一分。

在强化学习里面，环境跟奖励函数不是你可以控制的，环境跟奖励函数是在开始学习之前，就已经事先给定的。你唯一能做的事情是调整演员里面的策略 (policy)，使得演员可以得到最大的奖励。演员里面会有一个策略，这个策略决定了演员的行为。策略就是给一个外界的输入，然后它会输出演员现在应该要执行的行为。

- 策略一般写成 π 。假设你是用深度学习的技术来做强化学习的话，策略就是一个网络。网络里面就有一堆参数，我们用 θ 来代表 π 的参数。
- 网络的输入就是现在机器看到的东西，如果让机器打电玩的话，机器看到的东西就是游戏的画面。机器看到什么东西，会影响你现在训练到底好不好训练。举例来说，在玩游戏的时候，也许你觉得游戏的画面前后是相关的，也许你觉得你应该让你的策略，看从游戏初始到现在这个时间点，所有画面的总和。你可能会觉得你要用到 RNN 来处理它，不过这样子会比较难处理。要让你的机器，你的策略看到什么样的画面，这个是你自己决定的。让你知道说给机器看到什么样的游戏画面，可能是比较有效的。
- 输出的就是机器要采取什么样的行为。

上图就是具体的例子，

- 策略就是一个网络；

- 输入就是游戏的画面，它通常是由像素 (pixels) 所组成的；

Policy of Actor

- Policy π is a network with parameter θ
 - Input: the observation of machine represented as a vector or a matrix
 - Output: each action corresponds to a neuron in output layer

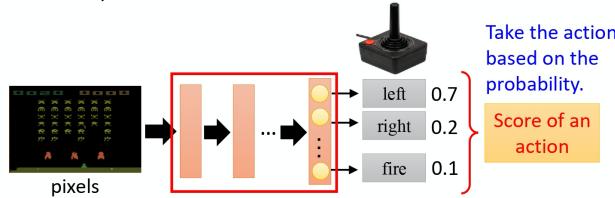


图 4.2

- 输出就是看看说有哪些选项是你可以去执行的，输出层就有几个神经元。
- 假设你现在可以做的行为有 3 个，输出层就是有 3 个神经元。每个神经元对应到一个可以采取的行为。
- 输入一个东西后，网络就会给每一个可以采取的行为一个分数。你可以把这个分数当作是概率。演员就是看这个概率的分布，根据这个概率的分布来决定它要采取的行为。比如说 70% 会向左走，20% 向右走，10% 开火等等。概率分布不同，演员采取的行为就会不一样。

Example: Playing Video Game

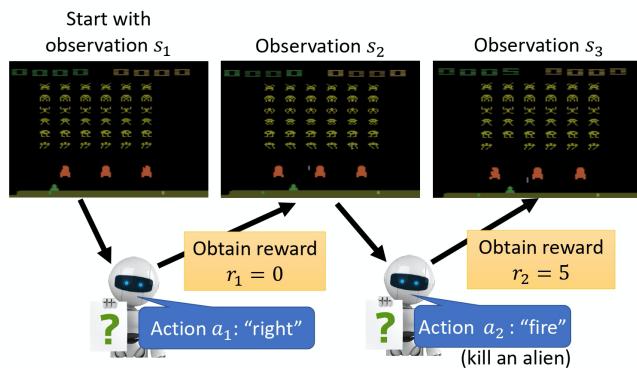


图 4.3

接下来用一个例子来说明演员是怎么样跟环境互动的。

首先演员会看到一个游戏画面，我们用 s_1 来表示游戏初始的画面。接下来演员看到这个游戏的初始画面以后，根据它内部的网络，根据它内部的策略来决定一个动作。假设它现在决定的动作是向右，它决定完动作以后，它就会得到一个奖励，代表它采取这个动作以后得到的分数。

我们把一开始的初始画面记作 s_1 ，把第一次执行的动作记作 a_1 ，把第一次执行动作完以后得到的奖励记作 r_1 。不同的书会有不同的定义，有人会觉得说这边应该要叫做 r_2 ，这个都可以，你自己看得懂就好。演员决定一个行为以后，就会看到一个新的游戏画面，这边是 s_2 。然后把这个 s_2 输入给演员，这个演员决定要开火，然后它可能杀了一只怪，就得到五分。这个过程就反复地持续下去，直到今天走到某一个时间点执行某一个动作，得到奖励之后，这个环境决定这个游戏结束了。比如说，如果在这个游戏里面，你是控制绿色的船去杀怪，如果你被杀死的话，游戏就结束，或是你把所有的怪都清空，游戏就结束了。

- 一场游戏叫做一个回合 (episode) 或者试验 (trial)。

Example: Playing Video Game

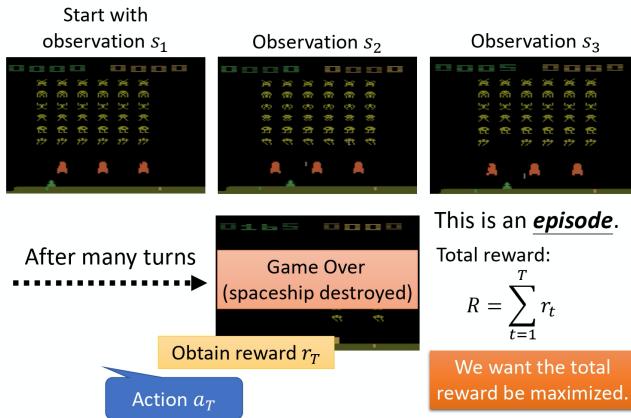


图 4.4

- 把这场游戏里面所有得到的奖励都加起来，就是总奖励 (total reward)，我们称其为回报 (return)，用 R 来表示它。
- 演员要想办法去最大化它可以得到的奖励。

Actor, Environment, Reward

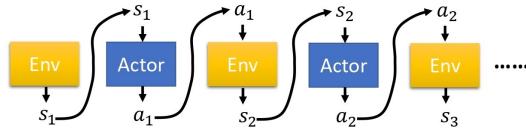


图 4.5

首先，环境是一个函数，游戏的主机也可以把它看作是一个函数，虽然它不一定是神经网络，可能是基于规则的 (rule-based) 规则，但你可以把它看作是一个函数。这个函数一开始就先吐出一个状态，也就是游戏的画面，接下来你的演员看到这个游戏画面 s_1 以后，它吐出 a_1 ，然后环境把 a_1 当作它的输入，然后它再吐出 s_2 ，吐出新的游戏画面。演员看到新的游戏画面，再采取新的行为 a_2 ，然后环境再看到 a_2 ，再吐出 s_3 。这个过程会一直持续下去，直到环境觉得说应该要停止为止。

在一场游戏里面，我们把环境输出的 s 跟演员输出的行为 a ，把 s 跟 a 全部串起来，叫做一个 Trajectory(轨迹)，如下式所示。

$$\text{Trajectory } \tau = \{s_1, a_1, s_2, a_2, \dots, s_t, a_t\}$$

你可以计算每一个轨迹发生的概率。假设现在演员的参数已经被给定了话，就是 θ 。根据 θ ，你其实可以计算某一个轨迹发生的概率，你可以计算某一个回合里面发生这样子状况的概率。

$$\begin{aligned} p_\theta(\tau) &= p(s_1)p_\theta(a_1|s_1)p(s_2|s_1, a_1)p_\theta(a_2|s_2)p(s_3|s_2, a_2)\dots \\ &= p(s_1)\prod_{t=1}^T p_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t) \end{aligned}$$

怎么算呢，如上式所示。在假设演员的参数就是 θ 的情况下，某一个轨迹 τ 的概率就是这样算的，你先算环境输出 s_1 的概率，再计算根据 s_1 执行 a_1 的概率，这是由你策略里面的网络参数 θ 所决定的，它是一个概率，因为你的策略的网络的输出是一个分布，演员是根据这个分布去做采样，决定现在实际上要采取的动作是哪一个。接下来环境根据 a_1 跟 s_1 产生 s_2 ，因为 s_2 跟 s_1 还是有关系的，下一个游戏画面

跟前一个游戏画面通常还是有关系的，至少要是连续的，所以给定前一个游戏画面 s_1 和现在演员采取的行为 a_1 ，就会产生 s_2 。

这件事情可能是概率，也可能不是概率，这个取决于环境，就是主机它内部设定是怎样。看今天这个主机在决定，要输出什么样的游戏画面的时候，有没有概率。因为如果没有概率的话，这个游戏的每次的行为都一样，你只要找到一条路径就可以过关了，这样感觉是蛮无聊的。所以游戏里面通常还是有一些概率的，你做同样的行为，给同样的前一个画面，下次产生的画面不见得是一样的。过程就反复继续下去，你就可以计算一个轨迹 s_1, a_1, s_2, a_2 出现的概率有多大。

这个概率取决于两部分，

- 环境的行为，环境的函数内部的参数或内部的规则长什么样子。 $p(s_{t+1}|s_t, a_t)$ 这一项代表的是环境，环境这一项通常你是无法控制它的，因为那个是人家写好的，你不能控制它。
- Agent 的行为。你能控制的是 $p_\theta(a_t|s_t)$ 。给定一个 s_t ，演员要采取什么样的 a_t 会取决于演员的参数 θ ，所以这部分是演员可以自己控制的。随着演员的行为不同，每个同样的轨迹，它就会有不同的出现的概率。

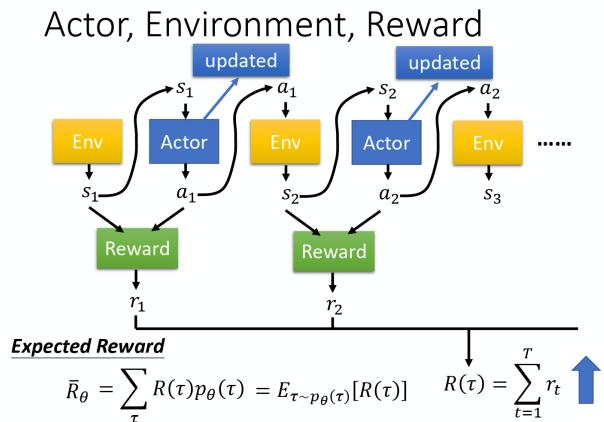


图 4.6

在强化学习里面，除了环境跟演员以外，还有奖励函数 (reward function)。

奖励函数根据在某一个状态采取的某一个动作决定说现在这个行为可以得到多少的分数。它是一个函数，给它 s_1, a_1 ，它告诉你得到 r_1 。给它 s_2, a_2 ，它告诉你得到 r_2 。把所有的 r 都加起来，我们就得到了 $R(\tau)$ ，代表某一个轨迹 τ 的奖励。

在某一场游戏里面，某一个回合里面，我们会得到 R 。我们要做的事情就是调整演员内部的参数 θ ，使得 R 的值越大越好。但实际上奖励不只是一个标量，奖励其实是一个随机变量。 R 其实是一个随机变量，因为演员在给定同样的状态会做什么样的行为，这件事情是有随机性的。环境在给定同样的观测要采取什么样的动作，要产生什么样的观测，本身也是有随机性的，所以 R 是一个随机变量。你能够计算的是 R 的期望值。你能够计算的是说，在给定某一组参数 θ 的情况下，我们会得到的 R_θ 的期望值是多少。

$$\bar{R}_\theta = \sum_{\tau} R(\tau) p_\theta(\tau)$$

这个期望值的算法如上式所示。我们要穷举所有可能的轨迹 τ ，每一个轨迹 τ 都有一个概率。

比如 θ 是一个很强的模型，它都不会死。因为 θ 很强，所以：

- 如果有一个回合 θ 很快就死掉了，因为这种情况很少会发生，所以该回合对应的轨迹 τ 的概率就很小；
- 如果有一个回合 θ 都一直没有死，因为这种情况很可能发生，所以该回合对应的轨迹 τ 的概率就很大。

你可以根据 θ 算出某一个轨迹 τ 出现的概率，接下来计算这个 τ 的总奖励是多少。总奖励使用这个 τ 出现的概率进行加权，对所有的 τ 进行求和，就是期望值。给定一个参数，你会得到的期望值。

$$\bar{R}_\theta = \sum_{\tau} R(\tau) p_\theta(\tau) = E_{\tau \sim p_\theta(\tau)} [R(\tau)]$$

我们还可以写成上式那样，从 $p_\theta(\tau)$ 这个分布采样一个轨迹 τ ，然后计算 $R(\tau)$ 的期望值，就是你的期望的奖励。我们要做的事情就是最大化期望奖励。

Policy Gradient $\bar{R}_\theta = \sum_{\tau} R(\tau) p_\theta(\tau) \quad \nabla \bar{R}_\theta = ?$

$$\begin{aligned} \nabla \bar{R}_\theta &= \sum_{\tau} R(\tau) \nabla p_\theta(\tau) = \sum_{\tau} R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\ &\quad R(\tau) \text{ do not have to be differentiable} \\ &\quad \text{It can even be a black box.} \\ &= \boxed{\sum_{\tau} R(\tau) p_\theta(\tau)} \nabla \log p_\theta(\tau) \quad \boxed{\begin{aligned} \nabla f(x) &= \\ f(x) \nabla \log f(x) \end{aligned}} \\ &= E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

图 4.7

怎么最大化期望奖励呢？我们用的是梯度上升 (gradient ascent)，因为要让它越大越好，所以是梯度上升。梯度上升在更新参数的时候要加。要进行梯度上升，我们先要计算期望的奖励 (expected reward) \bar{R} 的梯度。我们对 \bar{R} 取一个梯度，这里面只有 $p_\theta(\tau)$ 是跟 θ 有关，所以梯度就放在 $p_\theta(\tau)$ 这个地方。 $R(\tau)$ 这个奖励函数不需要是可微分的 (differentiable)，这个不影响我们解接下来的问题。举例来说，如果是在 GAN 里面， $R(\tau)$ 其实是一个 discriminator，它就算是没有办法微分，也无所谓，你还是可以做接下来的运算。

取梯度之后，我们背一个公式：

$$\nabla f(x) = f(x) \nabla \log f(x)$$

我们可以对 $\nabla p_\theta(\tau)$ 使用这个公式，然后会得到 $\nabla p_\theta(\tau) = p_\theta(\tau) \nabla \log p_\theta(\tau)$ 。

接下来，分子分母，上下同乘 $p_\theta(\tau)$ ，然后我们可以得到下式：

$$\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} = \log p_\theta(\tau)$$

如下式所示，对 τ 进行求和，把 $R(\tau)$ 和 $\log p_\theta(\tau)$ 这两项使用 $p_\theta(\tau)$ 进行加权，既然使用 $p_\theta(\tau)$ 进行加权，它们就可以被写成期望的形式。也就是你从 $p_\theta(\tau)$ 这个分布里面采样 τ 出来，去计算 $R(\tau)$ 乘上 $\nabla \log p_\theta(\tau)$ ，然后把它对所有可能的 τ 进行求和，就是这个期望的值 (expected value)。

$$\begin{aligned} \nabla \bar{R}_\theta &= \sum_{\tau} R(\tau) \nabla p_\theta(\tau) \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \nabla \log p_\theta(\tau) \\ &= E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \end{aligned}$$

实际上这个期望值没有办法算，所以你是用采样的方式来采样一大堆的 τ 。你采样 N 笔 τ ，然后你去计算每一笔的这些值，然后把它全部加起来，就可以得到梯度。你就可以去更新参数，你就可以去更新你

的 agent，如下式所示：

$$\begin{aligned} E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

下面给出 $\nabla \log p_\theta(\tau)$ 的具体计算过程，如下式所示。

$$\begin{aligned} \nabla \log p_\theta(\tau) &= \nabla \left(\log p(s_1) + \sum_{t=1}^T \log p_\theta(a_t | s_t) + \sum_{t=1}^T \log p(s_{t+1} | s_t, a_t) \right) \\ &= \nabla \log p(s_1) + \nabla \sum_{t=1}^T \log p_\theta(a_t | s_t) + \nabla \sum_{t=1}^T \log p(s_{t+1} | s_t, a_t) \\ &= \nabla \sum_{t=1}^T \log p_\theta(a_t | s_t) \\ &= \sum_{t=1}^T \nabla \log p_\theta(a_t | s_t) \end{aligned}$$

注意， $p(s_1)$ 和 $p(s_{t+1} | s_t, a_t)$ 来自于环境， $p_\theta(a_t | s_t)$ 是来自于 agent。 $p(s_1)$ 和 $p(s_{t+1} | s_t, a_t)$ 由环境决定，所以与 θ 无关，因此 $\nabla \log p(s_1) = 0$ ， $\nabla \sum_{t=1}^T \log p(s_{t+1} | s_t, a_t) = 0$ 。

$$\begin{aligned} \nabla \bar{R}_\theta &= \sum_{\tau} R(\tau) \nabla p_\theta(\tau) \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\ &= \sum_{\tau} R(\tau) p_\theta(\tau) \nabla \log p_\theta(\tau) \\ &= E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \\ &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

我们可以直观地来理解上面这个式子，也就是在你采样到的数据里面，你采样到在某一个状态 s_t 要执行某一个动作 a_t ，这个 s_t 跟 a_t 它是在整个轨迹 τ 的里面的某一个状态和动作的对。

假设你在 s_t 执行 a_t ，最后发现 τ 的奖励是正的，那你就要增加这一项的概率，你就要增加在 s_t 执行 a_t 的概率。反之，在 s_t 执行 a_t 会导致 τ 的奖励变成负的，你就要减少这一项的概率。

这个怎么实现呢？你用梯度上升来更新你的参数，你原来有一个参数 θ ，把你的 θ 加上你的梯度这一项，那当然前面要有个学习率，学习率也是要调整的，你可用 Adam、RMSProp 等方法对其进行调整。

我们可以套下面这个公式来把梯度计算出来：

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)$$

实际上，要套上面这个公式，首先你要先收集一大堆的 s 跟 a 的对 (pair)，你还要知道这些 s 跟 a 在跟环境互动的时候，你会得到多少的奖励。这些资料怎么收集呢？你要拿你的 agent，它的参数是 θ ，去跟环境做互动，也就是拿你已经训练好的 agent 先去跟环境玩一下，先去跟那个游戏互动一下，互动完以后，你就会得到一大堆游戏的纪录，你会记录说，今天先玩了第一场，在第一场游戏里面，我们在状态 s_1 采取动作 a_1 ，在状态 s_2 采取动作 a_2 。

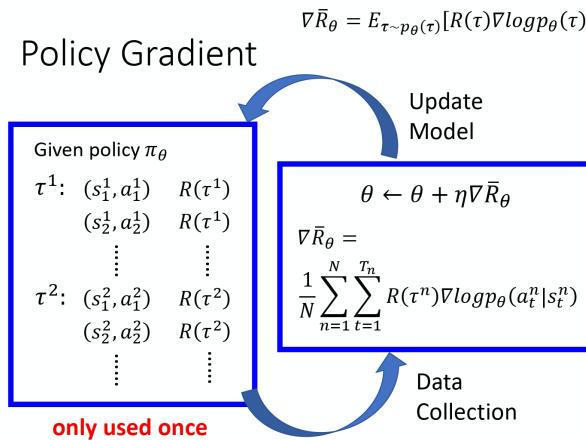


图 4.8

玩游戏的时候是有随机性的，所以 agent 本身是有随机性的，在同样状态 s_1 ，不是每次都会采取 a_1 ，所以你要记录下来。在状态 s_1^1 采取 a_1^1 ，在状态 s_2^1 采取 a_2^1 。整场游戏结束以后，得到的分数是 $R(\tau^1)$ 。你会采样到另外一笔数据，也就是另外一场游戏。在另外一场游戏里面，你在状态 s_1^2 采取 a_1^2 ，在状态 s_2^2 采取 a_2^2 ，然后你采样到的就是 τ^2 ，得到的奖励是 $R(\tau^2)$ 。

你就可以把采样到的东西代到这个梯度的式子里面，把梯度算出来。也就是把这边的每一个 s 跟 a 的对拿进来，算一下它的对数概率 (log probability)。你计算一下在某一个状态采取某一个动作的对数概率，然后对它取梯度，然后这个梯度前面会乘一个权重，权重就是这场游戏的奖励。有了这些以后，你就会去更新你的模型。

更新完你的模型以后。你要重新去收集数据，再更新模型。注意，一般 **policy gradient(PG)** 采样的数据就只会用一次。你把这些数据采样起来，然后拿去更新参数，这些数据就丢掉了。接着再重新采样数据，才能够去更新参数，等一下我们会解决这个问题。

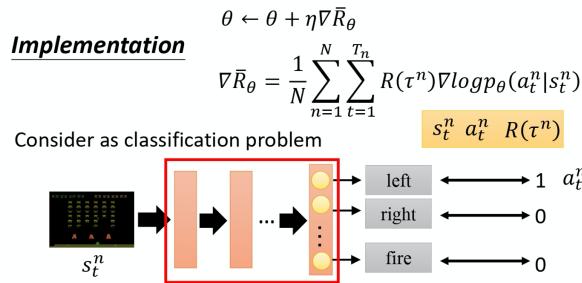


图 4.9

接下来讲一些实现细节。

我们可以把它想成一个分类的问题，在分类里面就是输入一个图像，然后输出决定说是 10 个类里面的哪一个。在做分类时，我们要收集一堆训练数据，要有输入跟输出的对。

在实现的时候，你就把状态当作是分类器的输入。你就当在做图像分类的问题，只是现在的类不是说图像里面有什么东西，而是说看到这张图像我们要采取什么样的行为，每一个行为就是一个类。比如说第一个类叫做向左，第二个类叫做向右，第三个类叫做开火。

在做分类的问题时，要有输入和正确的输出，要有训练数据。而这些训练数据是从采样的过程来的。假设在采样的过程里面，在某一个状态，你采样到你要采取动作 a ，你就把这个动作 a 当作是你的 ground truth。你在这个状态，你采样到要向左。本来向左这件事概率不一定是最高的，因为你是采样，它不一定

概率最高。假设你采样到向左，在训练的时候，你告诉机器说，调整网络的参数，如果看到这个状态，你就向左。在一般的分类问题里面，其实你在实现分类的时候，你的目标函数都会写成最小化交叉熵 (cross entropy)，其实最小化交叉熵就是最大化对数似然 (log likelihood)。

$$\begin{aligned} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \log p_\theta(a_t^n | s_t^n) &\xrightarrow{\text{TF, pyTorch ...}} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \nabla \log p_\theta(a_t^n | s_t^n) \\ \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \log p_\theta(a_t^n | s_t^n) &\xrightarrow{} \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

图 4.10

做分类的时候，目标函数就是最大化或最小化的对象，因为我们现在是最大化似然 (likelihood)，所以其实是最大化，你要最大化的对象，如下式所示：

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \log p_\theta(a_t^n | s_t^n)$$

像这种损失函数，你可在 TensorFlow 里调用现成的函数，它就会自动帮你算，然后你就可以把梯度计算出来。这是一般的分类问题，RL 唯一不同的地方是 loss 前面乘上一个权重：整场游戏得到的总奖励 R，它并不是在状态 s 采取动作 a 的时候得到的奖励，如下式所示：

$$\frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \log p_\theta(a_t^n | s_t^n)$$

你要把你的每一笔训练数据，都使用这个 R 进行加权。然后你用 TensorFlow 或 PyTorch 去帮你算梯度就结束了，跟一般分类差不多。

4.2 Tips

这边有一些在实现的时候，你也许用得上的 tip。

4.2.1 Tip 1: Add a Baseline

Tip 1: Add a Baseline

$$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta \quad \boxed{\text{It is possible that } R(\tau^n) \text{ is always positive.}}$$

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n) \quad b \approx E[R(\tau)]$$

图 4.11

第一个 tip 是 add 一个 baseline。如果给定状态 s 采取动作 a 会给你整场游戏正的奖励，就要增加它的概率。如果状态 s 执行动作 a，整场游戏得到负的奖励，就要减少这一项的概率。

但在很多游戏里面，奖励总是正的，就是说最低都是 0。比如说打乒乓球游戏，你的分数就是介于 0 到 21 分之间，所以 R 总是正的。假设你直接套用这个式子，在训练的时候告诉模型说，不管是什么动作你都应该要把它的概率提升。在理想上，这么做并不一定会有问题。因为虽然说 R 总是正的，但它正的量总是有大有小，你在玩乒乓球那个游戏里面，得到的奖励总是正的，但它是介于 0 到 21 分之间，有时候你采取某些动作可能是得到 0 分，采取某些动作可能是得到 20 分。

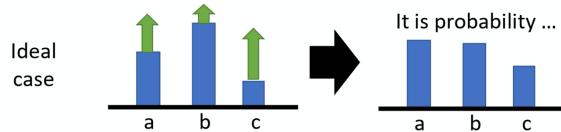


图 4.12

假设你在某一个状态有 3 个动作 a/b/c 可以执行。根据下式，

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n)$$

你要把这 3 项的概率，对数概率都拉高。但是它们前面权重的 R 是不一样的。 R 是有大有小的，权重小的，它上升的就少，权重多的，它上升的大一点。因为这个对数概率是一个概率，所以动作 a、b、c 的对数概率的和要是 0。所以上升少的，在做完归一化 (normalize) 以后，它其实就是下降的，上升的多的，才会上升。

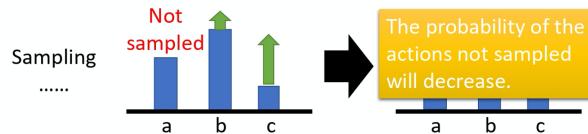


图 4.13

这是一个理想上的状况，但是实际上，我们是在做采样就本来这边应该是一个期望 (expectation)，对所有可能的 s 跟 a 的对进行求和。但你真正在学的时候不可能是这么做的，你只是采样了少量的 s 跟 a 的对而已。因为我们做的是采样，有一些动作可能从来都没有采样到。在某一个状态，虽然可以执行的动作有 a/b/c，但你可能只采样到动作 b，你可能只采样到动作 c，你没有采样到动作 a。但现在所有动作的奖励都是正的，所以根据这个式子，它的每一项的概率都应该要上升。你会遇到的问题是，因为 a 没有被采样到，其它动作的概率如果都要上升，a 的概率就下降。所以 a 不一定是一个不好的动作，它只是没被采样到。但只是因为它没被采样到，它的概率就会下降，这个显然是有问题的，要怎么解决这个问题呢？你会希望你的奖励不要总是正的。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n) \quad b \approx E[R(\tau)]$$

图 4.14

为了解决奖励总是正的这个问题，你可以把奖励减掉一项叫做 b ，这项 b 叫做 baseline。你减掉这项 b 以后，就可以让 $R(\tau^n) - b$ 这一项有正有负。所以如果得到的总奖励 $R(\tau^n)$ 大于 b 的话，就让它的概率上升。如果这个总奖励小于 b ，就算它是正的，正的很小也是不好的，你就要让这一项的概率下降。如果 $R(\tau^n) < b$ ，你就要让这个状态采取这个动作的分数下降。这个 b 怎么设呢？一个最简单的做法就是：你把 τ^n 的值取期望，算一下 τ^n 的平均值，即：

$$b \approx E[R(\tau)]$$

这是其中一种做法，你可以想想看有没有其它的做法。

所以在实现训练的时候，你会不断地把 $R(\tau)$ 的分数记录下来然后你会不断地去计算 $R(\tau)$ 的平均值，你会把这个平均值，当作你的 b 来用。这样就可以让你在训练的时候， $\nabla \log p_\theta(a_t^n | s_t^n)$ 乘上前面这一项，是有正有负的，这个是第一个 tip。

4.2.2 Tip 2: Assign Suitable Credit

第二个 tip：给每一个动作合适的分数 (credit)。

如果我们看下面这个式子的话，

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (R(\tau^n) - b) \nabla \log p_\theta(a_t^n | s_t^n)$$

我们原来会做的事情是，在某一个状态，假设你执行了某一个动作 a ，它得到的奖励，它前面乘上的这一项 $R(\tau^n) - b$ 。

只要在同一个回合里面，在同一场游戏里面，所有的状态跟动作的对都会使用同样的奖励项 (term) 进行加权，这件事情显然是不公平的，因为在同一场游戏里面也许有些动作是好的，有些动作是不好的。假设整场游戏的结果是好的，并不代表这个游戏里面每一个行为都是对的。若是整场游戏结果不好，但不代表游戏里面的所有行为都是错的。所以我们希望可以给每一个不同的动作前面都乘上不同的权重。每一个动作的不同权重，它反映了每一个动作到底是好还是不好。

$\times 3$	$\times -2$	$\times -2$
(s_a, a_1)	(s_b, a_2)	(s_c, a_3)
+5	+0	-2
$R = +3$		

图 4.15

举个例子，假设这个游戏都很短，只有 3 4 个互动，在 s_a 执行 a_1 得到 5 分。在 s_b 执行 a_2 得到 0 分。在 s_c 执行 a_3 得到 -2 分。整场游戏下来，你得到 +3 分，那你得到 +3 分代表在 s_b 执行动作 a_2 是好的吗？并不见得代表 s_b 执行 a_2 是好的。因为这个正的分数，主要来自于在 s_a 执行了 a_1 ，跟在 s_b 执行 a_2 是没有关系的，也许在 s_b 执行 a_2 反而是不好的，因为它导致你接下来会进入 s_c ，执行 a_3 被扣分，所以整场游戏得到的结果是好的，并不代表每一个行为都是对的。

如果按照我们刚才的讲法，整场游戏得到的分数是 3 分，那到时候在训练的时候，每一个状态跟动作的对，都会被乘上 +3。在理想的状况下，这个问题，如果你采样够多就可以被解决。因为假设你采样够多，在 s_b 执行 a_2 的这件事情，被采样到很多。就某一场游戏，在 s_b 执行 a_2 ，你会得到 +3 分。但在另外一场游戏，在 s_b 执行 a_2 ，你却得到了 -7 分，为什么会得到 -7 分呢？因为在 s_b 执行 a_2 之前，你在 s_a 执行 a_2 得到 -5 分，-5 分这件事可能也不是在 s_b 执行 a_2 的错，这两件事情，可能是没有关系的，因为它先发生了，这件事才发生，所以它们是没有关系的。

在 s_b 执行 a_2 可能造成的问题只有会在接下来 -2 分，而跟前面的 -5 分没有关系的。但是假设我们今天采样到这项的次数够多，把所有发生这件事情的情况的分数通通都集合起来，那可能不是一个问题。但现在的问题就是，我们采样的次数是不够多的。在采样的次数不够多的情况下，你要给每一个状态跟动作对合理的分数，你要让大家知道它合理的贡献。怎么给它一个合理的贡献呢？

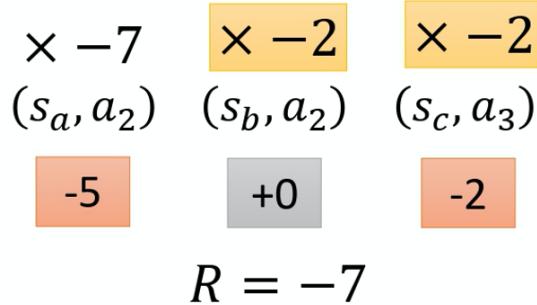


图 4.16

一个做法是计算这个对的奖励的时候，不把整场游戏得到的奖励全部加起来，只计算从这个动作执行以后所得到的奖励。因为这场游戏在执行这个动作之前发生的事情是跟执行这个动作是没有关系的，所以在执行这个动作之前得到多少奖励都不能算是这个动作的功劳。跟这个动作有关的东西，只有在执行这个动作以后发生的所有的奖励把它加起来，才是这个动作真正的贡献。所以在这个例子里面，在 s_b 执行 a_2 这件事情，也许它真正会导致你得到的分数应该是 -2 分而不是 +3 分，因为前面的 +5 分并不是执行 a_2 的功劳。实际上执行 a_2 以后，到游戏结束前，你只有被扣 2 分而已，所以它应该是 -2。那一样的道理，今天执行 a_2 实际上不应该是扣 7 分，因为前面扣 5 分，跟在 s_b 执行 a_2 是没有关系的。在 s_b 执行 a_2 ，只会让你被扣两分而已，所以也许在 s_b 执行 a_2 ，你真正会导致的结果只有扣两分而已。如果要把它写成式子的话是什么样子呢？如下式所示：

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (\cancel{R(\cancel{-7})} - b) \nabla \log p_\theta(a_t^n | s_t^n)$$

↗

$$\sum_{t'=t}^{T_n} r_{t'}^n$$

图 4.17

本来的权重是整场游戏的奖励的总和，现在改成从某个时间 t 开始，假设这个动作是在 t 这个时间点所执行的，从 t 这个时间点一直到游戏结束所有奖励的总和，才真的代表这个动作是好的还是不好的。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (\cancel{R(\cancel{-7})} - b) \nabla \log p_\theta(a_t^n | s_t^n)$$

↗

$$\sum_{t'=t}^{T_n} r_{t'}^n \rightarrow \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$$

Add discount factor $\gamma \in [0,1]$

Can be state-dependent

图 4.18

接下来再更进一步，我们把未来的奖励做一个折扣 (discount)，由此得到的回报被称为 [Discounted Return\(折扣回报\)](#)。为什么要把未来的奖励做一个折扣呢？因为虽然在某一个时间点，执行某一个动作，

会影响接下来所有的结果，有可能在某一个时间点执行的动作，接下来得到的奖励都是这个动作的功劳。但在比较真实的情况下，如果时间拖得越长，影响力就越小。比如说在第二个时间点执行某一个动作，那我在第三个时间点得到的奖励可能是在第二个时间点执行某个动作的功劳，但是在 100 个时间点之后又得到奖励，那可能就不是在第二个时间点执行某一个动作得到的功劳。所以我们实际上在做的时候，你会在 R 前面乘上一个 discount factor γ , $\gamma \in [0, 1]$ ，一般会设个 0.9 或 0.99, $\gamma = 0$: 只关心即时奖励, $\gamma = 1$: 未来奖励等同于即时奖励。

如果时间点 t' 越大，它前面就乘上越多次的 γ ，就代表说现在在某一个状态 s_t ，执行某一个动作 a_t 的时候，它真正的分数是在执行这个动作之后所有奖励的总和，而且你还要乘上 γ 。

举一个例子，你就想成说，这是游戏的第 1、2、3、4 回合，假设你在游戏的第二回合的某一个 s_t 执行 a_t 得到 +1 分，在 s_{t+1} 执行 a_{t+1} 得到 +3 分，在 s_{t+2} 执行 a_{t+2} 得到 -5 分，然后第二回合结束。 a_t 的分数应该是：

$$1 + \gamma \times 3 + \gamma^2 \times -5$$

实际上就是这么实现的， b 可以是取决于状态 (state-dependent) 的，事实上 b 它通常是一个网络估计出来的，它是一个网络的输出。

Advantage Function $A^\theta(s_t, a_t)$	How good it is if we take a_t other than other actions at s_t . Estimated by "critic" (later)
--	--

图 4.19

把 $R - b$ 这一项合起来，我们统称为**优势函数 (advantage function)**，用 A 来代表优势函数。优势函数取决于 s 和 a，我们就是要计算的是在某一个状态 s 采取某一个动作 a 的时候，优势函数有多大。

在算优势函数时，你要计算 $\sum_{t'=t}^{T_n} r_t^n$ ，你需要有一个互动的结果。你需要有一个模型去跟环境做互动，你才知道接下来得到的奖励会有多少。优势函数 $A^\theta(s_t, a_t)$ 的上标是 θ ， θ 就是代表说是用 θ 这个模型跟环境去做互动，然后你才计算出这一项。从时间 t 开始到游戏结束为止，所有 r 的加和减掉 b，这个就叫优势函数。

优势函数的意义就是，假设我们在某一个状态 s_t 执行某一个动作 a_t ，相较于其他可能的动作，它有多好。它在意的不是一个绝对的好，而是相对的好，即**相对优势 (relative advantage)**。因为会减掉一个 b ，减掉一个 baseline，所以这个东西是相对的好，不是绝对的好。 $A^\theta(s_t, a_t)$ 通常可以是由一个网络估计出来的，这个网络叫做 critic。

4.3 REINFORCE: Monte Carlo Policy Gradient

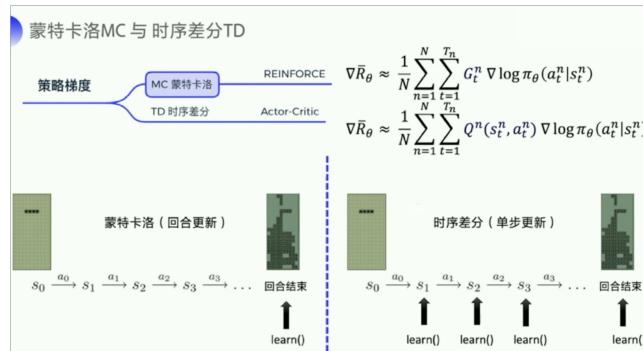


图 4.20

MC 可以理解为算法完成一个回合之后，再拿这个回合的数据来去 learn 一下，做一次更新。因为我们已经拿到了一整个回合的数据的话，也能够拿到每一个步骤的奖励，我们可以很方便地去计算每个步骤的未来总收益，就是我们的期望，就是我们的回报 G_t 。 G_t 是我们的未来总收益， G_t 代表是从这个步骤后面，我能拿到的收益之和是多少。 G_1 是说我从第一步开始，往后能够拿到多少的收益。 G_2 是说从第二步开始，往后一共能够拿到多少的收益。

相比 MC 还是一个回合更新一次这样子的方式，TD 就是每个步骤都更新一下。每走一步，我就更新下，这样的更新频率会更高一点。它拿的是 Q-function 来去近似地表示我的未来总收益 G_t 。

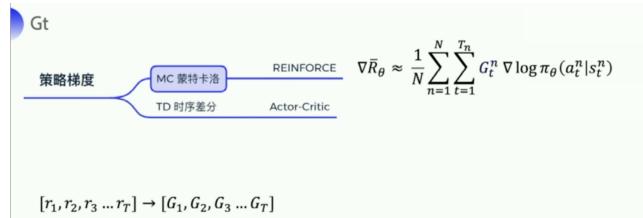


图 4.21

我们介绍下策略梯度最简单的也是最经典的一个算法 REINFORCE。REINFORCE 用的是回合更新的方式。它在代码上的处理上是先拿到每个步骤的奖励，然后计算每个步骤的未来总收益 G_t 是多少，然后拿每个 G_t 代入公式，去优化每一个动作的输出。所以编写代码时会有这样一个函数，输入每个步骤拿到的奖励，把这些奖励转成每一个步骤的未来总收益。因为未来总收益是这样计算的：

$$\begin{aligned} G_t &= \sum_{k=t+1}^T \gamma^{k-t-1} r_k \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned}$$

上一个步骤和下一个步骤的未来总收益可以有这样子的一个关系。所以在代码的计算上，我们就是从后往前推，一步一步地往前推，先算 G_T ，然后往前推，一直算到 G_1 。

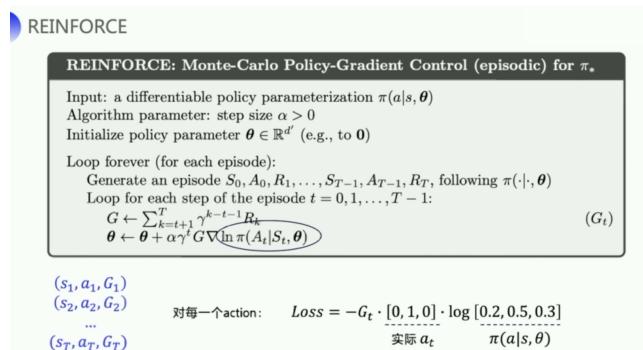


图 4.22

REINFORCE 的伪代码主要看最后四行，先产生一个回合的数据，比如 $(s_1, a_1, G_1), (s_2, a_2, G_2), \dots, (s_T, a_T, G_T)$ 。然后针对每个动作来计算梯度。在代码上计算时，我们要拿到神经网络的输出。神经网络会输出每个动作对应的概率值，然后我们还可以拿到实际的动作，把它转成 one-hot 向量乘一下，我们可以拿到 $\ln \pi(A_t | S_t, \theta)$ 。

独热编码 (one-hot Encoding) 通常用于处理类别间不具有大小关系的特征。例如血型，一共有 4 个取值 (A 型、B 型、AB 型、O 型)，独热编码会把血型变成一个 4 维稀疏向量，A 型血表示为 $(1, 0, 0, 0)$ ，B 型血表示为 $(0, 1, 0, 0)$ ，AB 型会表示为 $(0, 0, 1, 0)$ ，O 型血表示为 $(0, 0, 0, 1)$ 。

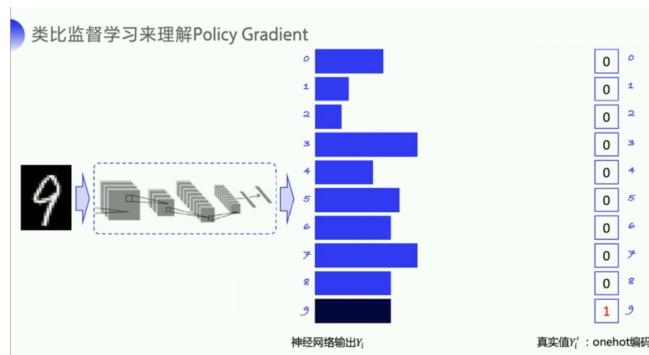


图 4.23

- 手写数字识别是一个经典的多分类问题，输入是一张手写数字的图片，经过神经网络输出的是各个类别中的一个概率。
- 目的是希望输出的这个概率的分布尽可能地去贴近真实值的概率分布。
- 因为真实值只有一个数字 9，你用这个 one-hot 向量的形式去给它编码的话，也可以把这个真实值理解为一个概率分布，9 的概率就是 1，其他的概率就是 0。
- 神经的网络输出一开始可能会比较平均，通过不断地迭代，训练优化之后，我会希望 9 输出的概率可以远高于其他数字输出的概率。

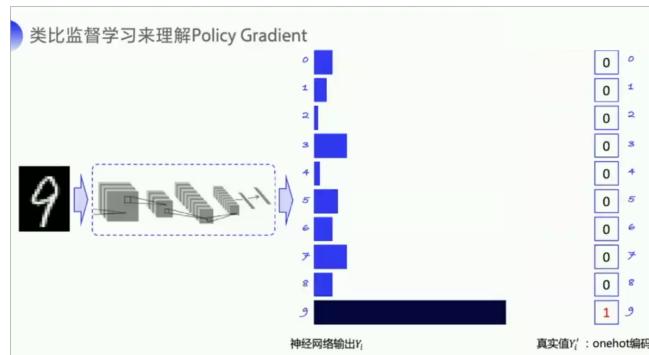


图 4.24

如上图所示，就是提高 9 对应的概率，降低其他数字对应的概率，让神经网络输出的概率能够更贴近这个真实值的概率分布。我们可以用交叉熵来表示两个概率分布之间的差距。

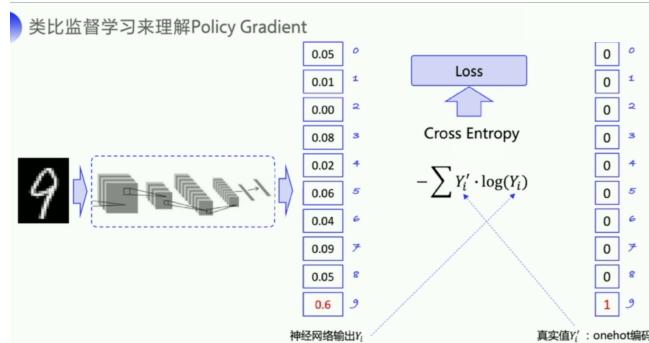


图 4.25

我们看一下它的优化流程，就是怎么让这个输出去逼近这个真实值。它的优化流程就是将图片作为输入传给神经网络，神经网络会判断这个图片属于哪一类数字，输出所有数字可能的概率，然后再计算这个交叉熵，就是神经网络的输出 Y_i 和真实的标签值 Y'_i 之间的距离 $-\sum Y'_i \cdot \log(Y_i)$ 。我们希望尽可能地缩

小这两个概率分布之间的差距，计算出来的交叉熵可以作为这个损失函数传给神经网络里面的优化器去优化，去自动去做神经网络的参数更新。

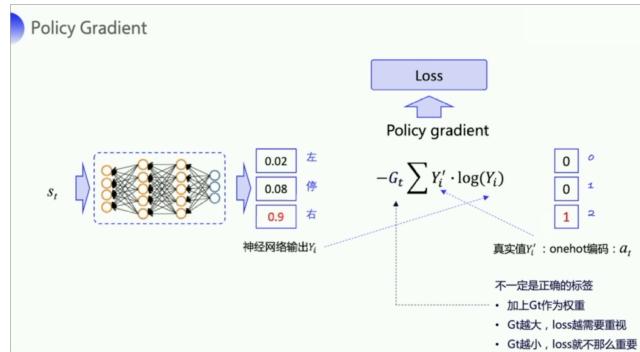


图 4.26

类似地，policy gradient 预测每一个状态下面应该要输出的这个行动的概率，就是输入状态 s_t ，然后输出动作的概率，比如 0.02, 0.08, 0.09。实际上输出给环境的动作是随机选了一个动作，比如说我选了右这个动作，它的 one-hot 向量就是 0, 0, 1。我们把神经网络的输出和实际动作带入交叉熵的公式就可以求出输出的概率和实际的动作之间的差距。

但这个实际的动作 a_t 只是我们输出的真实的动作，它并不一定是正确的动作，它不能像手写数字识别一样作为一个正确的标签来去指导神经网络朝着正确的方向去更新，所以我们需要乘以一个奖励回报 G_t 。这个奖励回报相当于是对这个真实动作的评价。

- 如果 G_t 越大，未来总收益越大，那就说明当前输出的这个真实动作就越好，这个 loss 就越需要重视。
- 如果 G_t 越小，那就说明做这个动作 a_t 并没有那么的好，loss 的权重就要小一点，优化力度就小一点。

通过这个和那个手写输入识别的一个对比，我们就知道为什么 loss 会构造成这个样子。

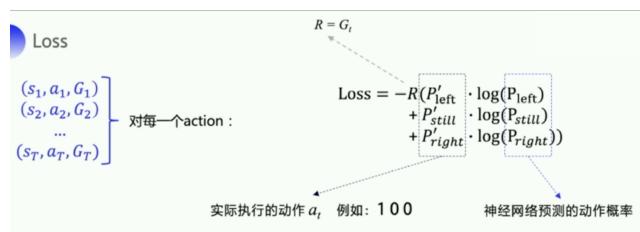


图 4.27

实际上我们在计算这个 loss 的时候，我们要拿到那个 $\ln \pi(A_t|S_t, \theta)$ 。我就拿实际执行的这个动作，先取个 one-hot 向量，然后再拿到神经网络预测的动作概率，这两个一相乘，我就可以拿到算法里面的那个 $\ln \pi(A_t|S_t, \theta)$ 。这个就是我们要构造的 loss。因为我们会拿到整个回合的所有轨迹，所以我们可以对这一条整条轨迹里面的每个动作都去计算一个 loss。把所有的 loss 加起来之后，我们再扔给 adam 的优化器去自动更新参数就好了。

上图是 REINFORCE 的流程图。首先我们需要一个 policy model 来输出动作概率，输出动作概率后，我们 `sample()` 函数去得到一个具体的动作，然后跟环境交互过后，我们可以得到一个回合的数据。拿到回合数据之后，我再去执行一下 `learn()` 函数，在 `learn()` 函数里面，我就可以拿这些数据去构造损失函数，扔给这个优化器去优化，去更新我的 policy model。

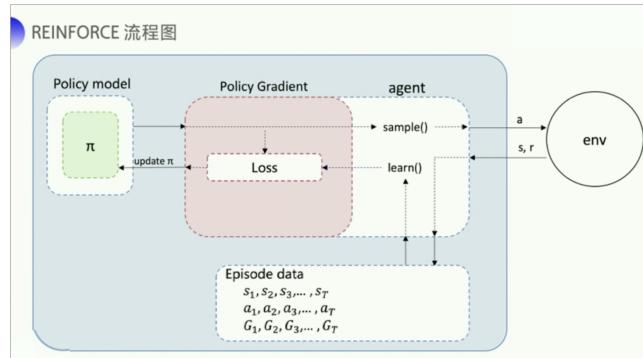


图 4.28

4.4 Keywords

- policy (策略): 每一个 actor 中会有对应的策略，这个策略决定了 actor 的行为。具体来说，Policy 就是给一个外界的输入，然后它会输出 actor 现在应该要执行的行为。一般地，我们将 policy 写成 π 。
- Return (回报): 一个回合 (Episode) 或者试验 (Trial) 所得到的所有 reward 的总和，也被人们称为 Total reward。一般地，我们用 R 来表示它。
- Trajectory: 一个试验中我们将 environment 输出的 s 跟 actor 输出的行为 a ，把这个 s 跟 a 全部串起来形成的集合，我们称为 Trajectory，即 $Trajectory \tau = \{s_1, a_1, s_2, a_2, \dots, s_t, a_t\}$ 。
- Reward function: 根据在某一个 state 采取的某一个 action 决定说现在这个行为可以得到多少的分数，它是一个 function。也就是给一个 s_1, a_1 ，它告诉你得到 r_1 。给它 s_2, a_2 ，它告诉你得到 r_2 。把所有的 r 都加起来，我们就得到了 $R(\tau)$ ，代表某一个 trajectory τ 的 reward。
- Expected reward: $\bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) = E_{\tau \sim p_\theta(\tau)}[R(\tau)]$ 。
- Reinforce: 基于策略梯度的强化学习的经典算法，其采用回合更新的模式。

4.5 Questions

- 如果我们想让机器人自己玩 video game，那么强化学习中三个组成 (actor、environment、reward function) 部分具体分别是什么？

答：actor 做的事情就是去操控游戏的摇杆，比如说向左、向右、开火等操作；environment 就是游戏的主机，负责控制游戏的画面负责控制说，怪物要怎么移动，你现在要看到什么画面等等；reward function 就是当你做什么事情，发生什么状况的时候，你可以得到多少分数，比如说杀一只怪兽得到 20 分等等。

- 在一个 process 中，一个具体的 trajectory s_1, a_1, s_2, a_2 出现的概率取决于什么？

答：1. 一部分是 environment 的行为，environment 的 function 它内部的参数或内部的规则长什么样子。 $p(s_{t+1}|s_t, a_t)$ 这一项代表的是 environment，environment 这一项通常你是无法控制它的，因为那个是人家写好的，或者已经客观存在的。

2. 另一部分是 agent 的行为，你能控制的是 $p_\theta(a_t|s_t)$ 。给定一个 s_t ，actor 要采取什么样的 a_t 会取决于你 actor 的参数 θ ，所以这部分是 actor 可以自己控制的。随着 actor 的行为不同，每个同样的 trajectory，它就会有不同的出现的概率。

- 当我们在计算 maximize expected reward 时，应该使用什么方法？

答：gradient ascent (梯度上升)，因为它越大越好，所以是 gradient ascent。Gradient ascent 在 update 参数的时候要加。要进行 gradient ascent，我们先要计算 expected reward \bar{R} 的 gradient。我们对 \bar{R} 取一个 gradient，这里面只有 $p_\theta(\tau)$ 是跟 θ 有关，所以 gradient 就放在 $p_\theta(\tau)$ 这个地方。

- 我们应该如何理解梯度策略的公式呢？

答：

$$\begin{aligned} E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] &\approx \frac{1}{N} \sum_{n=1}^N R(\tau^n) \nabla \log p_\theta(\tau^n) \\ &= \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} R(\tau^n) \nabla \log p_\theta(a_t^n | s_t^n) \end{aligned}$$

$p_\theta(\tau)$ 里面有两项， $p(s_{t+1}|s_t, a_t)$ 来自于 environment， $p_\theta(a_t|s_t)$ 是来自于 agent。 $p(s_{t+1}|s_t, a_t)$ 由环境决定从而与 θ 无关，因此 $\nabla \log p(s_{t+1}|s_t, a_t) = 0$ 。因此 $\nabla p_\theta(\tau) = \nabla \log p_\theta(a_t^n | s_t^n)$ 。公式的具体推导可见我们的教程。

具体来说：

- 假设你在 s_t 执行 a_t ，最后发现 τ 的 reward 是正的，那你就需要增加这一项的概率，即增加在 s_t 执行 a_t 的概率。
- 反之，在 s_t 执行 a_t 会导致 τ 的 reward 变成负的，你就要减少这一项的概率。

- 我们可以使用哪些方法来进行 gradient ascent 的计算？

答：用 gradient ascent 来 update 参数，对于原来的参数 θ ，可以将原始的 θ 加上更新的 gradient 这一项，再乘以一个 learning rate，learning rate 其实也是要调的，和神经网络一样，我们可以使用 Adam、RMSProp 等优化器对其进行调整。

- 我们进行基于梯度策略的优化时的小技巧有哪些？

答：

1. Add a baseline：为了防止所有的 reward 都大于 0，从而导致每一个 stage 和 action 的变换，会使得每一项的概率都会上升。所以通常为了解决这个问题，我们把 reward 减掉一项叫做 b，这项 b 叫做 baseline。你减掉这项 b 以后，就可以让 $R(\tau^n) - b$ 这一项，有正有负。所以如果得到的 total reward $R(\tau^n)$ 大于 b 的话，就让它的概率上升。如果这个 total reward 小于 b，就算它是正的，正的很小也是不好的，你就要让这一项的概率下降。如果 $R(\tau^n) < b$ ，你就要让这个 state 采取这个 action 的分数下降。这样也符合常理。但是使用 baseline 会让本来 reward 很大的“行为”的 reward 变小，降低更新速率。

2. Assign suitable credit：首先第一层，本来的 weight 是整场游戏的 reward 的总和。那现在改成从某个时间 t 开始，假设这个 action 是在 t 这个时间点所执行的，从 t 这个时间点，一直到游戏结束所有 reward 的总和，才真的代表这个 action 是好的还是不好的；接下来我们再进一步，我们把未来的 reward 做一个 discount，这里我们称由此得到的 reward 的和为 Discounted Return(折扣回报)。
3. 综合以上两种 tip，我们将其统称为 Advantage function，用 A 来代表 advantage function。Advantage function 是 dependent on s and a，我们就是要计算的是在某一个 state s 采取某一个 action a 的时候，advantage function 有多大。

4. Advantage function 的意义就是，假设我们在某一个 state s_t 执行某一个 action a_t ，相较于其他可能的 action，它有多好。它在意的不是一个绝对的好，而是相对的好，即相对优势 (relative advantage)。因为会减掉一个 b，减掉一个 baseline，所以这个东西是相对的好，不是绝对的好。 $A^\theta(s_t, a_t)$ 通常可以是由一个 network estimate 出来的，这个 network 叫做 critic。

- 对于梯度策略的两种方法，蒙特卡洛 (MC) 强化学习和时序差分 (TD) 强化学习两个方法有什么联系和区别？

答：

1. 两者的更新频率不同，蒙特卡洛强化学习方法是每一个 episode 更新一次，即需要经历完整的状态序列后再更新（比如我们的贪吃蛇游戏，贪吃蛇“死了”游戏结束后再更新），而对于时序差分强化学习方法是每一个 step 就更新一次，（比如我们的贪吃蛇游戏，贪吃蛇每移动一次（或几次）就进行更新）。相对来说，时序差分强化学习方法比蒙特卡洛强化学习方法更新的频率更快。

2. 时序差分强化学习能够在知道一个小 step 后就进行学习，相比于蒙特卡洛强化学习，其更加快

速、灵活。

3. 具体举例来说：假如我们要优化开车去公司的通勤时间。对于此问题，每一次通勤，我们将会到达不同的路口。对于时序差分（TD）强化学习，其会对于每一个经过的路口都会计算时间，例如在路口 A 就开始更新预计到达路口 B、路口 C ……，以及到达公司的时间；而对于蒙特卡洛（MC）强化学习，其不会每经过一个路口就更新时间，而是到达最终的目的地后，再修改每一个路口和公司对应的时间。

- 请详细描述 REINFORCE 的计算过程。

答：首先我们需要根据一个确定好的 policy model 来输出每一个可能的 action 的概率，对于所有的 action 的概率，我们使用 sample 方法（或者是随机的方法）去选择一个 action 与环境进行交互，同时环境就会给我们反馈一整个 episode 数据。对于此 episode 数据输入到 learn 函数中，并根据 episode 数据进行 loss function 的构造，通过 adam 等优化器的优化，再来更新我们的 policy model。

4.6 Something About Interview

- 高冷的面试官：同学来吧，给我手工推导一下策略梯度公式的计算过程。

答：首先我们目的是最大化 reward 函数，即调整 θ ，使得期望回报最大，可以用公式表示如下

$$J(\theta) = E_{\tau \sim p_{\theta}(\tau)} [\sum_t r(s_t, a_t)]$$

对于上面的式子， τ 表示从开始到结束的一条完整路径。通常，对于最大化问题，我们可以使用梯度上升算法来找到最大值，即

$$\theta^* = \theta + \alpha \nabla J(\theta)$$

所以我们仅仅需要计算（更新） $\nabla J(\theta)$ ，也就是计算回报函数 $J(\theta)$ 关于 θ 的梯度，也就是策略梯度，计算方法如下：

$$\nabla_{\theta} J(\theta) = \int \nabla_{\theta} p_{\theta}(\tau) r(\tau) d_{\tau} = \int p_{\theta} \nabla_{\theta} \log p_{\theta}(\tau) r(\tau) d_{\tau} = E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) r(\tau)]$$

接着我们继续讲上式展开，对于 $p_{\theta}(\tau)$ ，即 $p_{\theta}(\tau|\theta)$ ：

$$p_{\theta}(\tau|\theta) = p(s_1) \prod_{t=1}^T \pi_{\theta}(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

取对数后为：

$$\log p_{\theta}(\tau|\theta) = \log p(s_1) + \sum_{t=1}^T \log \pi_{\theta}(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

继续求导：

$$\nabla \log p_{\theta}(\tau|\theta) = \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)$$

带入第三个式子，可以将其化简为：

$$\begin{aligned} \nabla_{\theta} J(\theta) &= E_{\tau \sim p_{\theta}(\tau)} [\nabla_{\theta} \log p_{\theta}(\tau) r(\tau)] \\ &= E_{\tau \sim p_{\theta}} [(\nabla_{\theta} \log \pi_{\theta}(a_t|s_t)) (\sum_{t=1}^T r(s_t, a_t))] \\ &= \frac{1}{N} \sum_{i=1}^N [(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t})) (\sum_{t=1}^N r(s_{i,t}, a_{i,t}))] \end{aligned}$$

- 高冷的面试官：可以说一下你了解到的基于梯度策略的优化时的小技巧吗？

答：

1. Add a baseline: 为了防止所有的 reward 都大于 0，从而导致每一个 stage 和 action 的变换，会使得每一项的概率都会上升。所以通常为了解决这个问题，我们把 reward 减掉一项叫做 b，这项 b 叫做 baseline。你减掉这项 b 以后，就可以让 $R(\tau^n) - b$ 这一项，有正有负。所以如果得到的 total reward $R(\tau^n)$ 大于 b 的话，就让它的概率上升。如果这个 total reward 小于 b，就算它是正的，正的很小也是不好的，你就要让这一项的概率下降。如果 $R(\tau^n) < b$ ，你就要让这个 state 采取这个 action 的分数下降。这样也符合常理。但是使用 baseline 会让本来 reward 很大的“行为”的 reward 变小，降低更新速率。
2. Assign suitable credit: 首先第一层，本来的 weight 是整场游戏的 reward 的总和。那现在改成从某个时间 t 开始，假设这个 action 是在 t 这个时间点所执行的，从 t 这个时间点，一直到游戏结束所有 reward 的总和，才真的代表这个 action 是好的还是不好的；接下来我们再进一步，我们把未来的 reward 做一个 discount，这里我们称由此得到的 reward 的和为 Discounted Return(折扣回报)。
3. 综合以上两种 tip，我们将其统称为 Advantage function，用 A 来代表 advantage function。Advantage function 是 dependent on s and a，我们就是要计算的是在某一个 state s 采取某一个 action a 的时候，advantage function 有多大。

References

- Intro to Reinforcement Learning (强化学习纲要)
- 神经网络与深度学习
- 百面深度学习

第 5 章 PPO

5.1 From On-policy to Off-policy

在讲 PPO 之前，我们先讲一下 on-policy 和 off-policy 这两种训练方法的区别。在强化学习里面，我们要学习的就是一个 agent。

- 如果要学习的 agent 跟和环境互动的 agent 是同一个的话，这个叫做 [on-policy\(同策略\)](#)。
- 如果要学习的 agent 跟和环境互动的 agent 不是同一个的话，那这个叫做 [off-policy\(异策略\)](#)。

比较拟人化的讲法是如果要学习的那个 agent，一边跟环境互动，一边做学习这个叫 on-policy。如果它在旁边看别人玩，通过看别人玩来学习的话，这个叫做 off-policy。

为什么我们会想要考虑 off-policy？让我们来想想 policy gradient。Policy gradient 是 on-policy 的做法，因为在做 policy gradient 时，我们需要有一个 agent、一个 policy 和一个 actor。这个 actor 先去跟环境互动去搜集资料，搜集很多的 τ ，根据它搜集到的资料按照 policy gradient 的式子去更新 policy 的参数。所以 policy gradient 是一个 on-policy 的算法。

[近端策略优化 \(Proximal Policy Optimization, 简称 PPO\)](#) 是 policy gradient 的一个变形，它是现在 OpenAI 默认的强化学习算法。

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)] \quad (5.1)$$

问题是上面这个更新的式子中的 $E_{\tau \sim p_\theta(\tau)}$ 应该是你现在的 policy π_θ 所采样出来的轨迹 τ 做期望 (expectation)。一旦更新了参数，从 θ 变成 θ' ， $p_\theta(\tau)$ 这个概率就不对了，之前采样出来的数据就变的不能用了。所以 policy gradient 是一个会花很多时间来采样数据的算法，大多数时间都在采样数据，agent 去跟环境做互动以后，接下来就要更新参数。你只能更新参数一次。接下来你就要重新再去收集数据，然后才能再次更新参数。

这显然是非常花时间的，所以我们想要从 on-policy 变成 off-policy。这样做就可以用另外一个 policy，另外一个 actor θ' 去跟环境做互动 (θ' 被固定了)。用 θ' 收集到的数据去训练 θ 。假设我们可以用 θ' 收集到的数据去训练 θ ，意味着说我们可以把 θ' 收集到的数据用非常多次，我们可以执行梯度上升 (gradient ascent) 好几次，我们可以更新参数好几次，都只要用同一笔数据就好了。因为假设 θ 有能力学习另外一个 actor θ' 所采样出来的数据的话，那 θ' 就只要采样一次，也许采样多一点的数据，让 θ 去更新很多次，这样就会比较有效率。

5.1.1 Importance Sampling

具体怎么做呢？这边就需要介绍 [importance sampling\(重要性采样\)](#) 的概念。

假设你有一个函数 $f(x)$ ，你要计算从 p 这个分布采样 x ，再把 x 带到 f 里面，得到 $f(x)$ 。你要该怎么计算这个 $f(x)$ 的期望值？假设你不能对 p 这个分布做积分的话，那你可以从 p 这个分布去采样一些数据 x^i 。把 x^i 代到 $f(x)$ 里面，然后取它的平均值，就可以近似 $f(x)$ 的期望值。

现在有另外一个问题，我们没有办法从 p 这个分布里面采样数据。假设我们不能从 p 采样数据，只能从另外一个分布 q 去采样数据， q 可以是任何分布。我们不能够从 p 去采样数据，但可以从 q 去采样 x 。我们从 q 去采样 x^i 的话就不能直接套下面的式子：

$$E_{x \sim p}[f(x)] \approx \frac{1}{N} \sum_{i=1}^N f(x^i) \quad (5.2)$$

因为上式是假设你的 x 都是从 p 采样出来的。

所以做一个修正，修正是这样子的。期望值 $E_{x \sim p}[f(x)]$ 其实就是 $\int f(x)p(x)dx$ ，我们对其做如下的变换：

$$\int f(x)p(x)dx = \int f(x) \frac{p(x)}{q(x)} q(x)dx = E_{x \sim q}[f(x) \frac{p(x)}{q(x)}] \quad (5.3)$$

我们就可以写成对 q 里面所采样出来的 x 取期望值。我们从 q 里面采样 x , 然后再去计算 $f(x) \frac{p(x)}{q(x)}$, 去再取期望值。所以就算我们不能从 p 里面去采样数据, 只要能够从 q 里面去采样数据, 然后代入上式, 你就可以计算从 p 这个分布采样 x 代入 f 以后所算出来的期望值。

这边是从 q 做采样, 所以从 q 里采样出来的每一笔数据, 你需要乘上一个**重要性权重 (importance weight)** $\frac{p(x)}{q(x)}$ 来修正这两个分布的差异。 $q(x)$ 可以是任何分布, 唯一的限制就是 $q(x)$ 的概率是 0 的时候, $p(x)$ 的概率不为 0, 不然这样会没有定义。假设 $q(x)$ 的概率是 0 的时候, $p(x)$ 的概率也都是 0 的话, 那这样 $p(x)$ 除以 $q(x)$ 是有定义的。所以这个时候你就可以使用重要性采样这个技巧。你就可以从 p 做采样换成从 q 做采样。

重要性采样有一些问题。虽然理论上你可以把 p 换成任何的 q 。但是在实现上, p 和 q 不能差太多。差太多的话, 会有一些问题。什么样的问题呢?

$$E_{x \sim p}[f(x)] = E_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] \quad (5.4)$$

虽然上式成立 (上式左边是 $f(x)$ 的期望值, 它的分布是 p , 上式右边是 $f(x) \frac{p(x)}{q(x)}$ 的期望值, 它的分布是 q), 但如果不是算期望值, 而是算方差的话, $\text{Var}_{x \sim p}[f(x)]$ 和 $\text{Var}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$ 是不一样的。两个随机变量的平均值一样, 并不代表它的方差一样。

我们可以代一下方差的公式 $\text{Var}[X] = E[X^2] - (E[X])^2$, 然后得到下式:

$$\text{Var}_{x \sim p}[f(x)] = E_{x \sim p} [f(x)^2] - (E_{x \sim p}[f(x)])^2 \quad (5.5)$$

$$\begin{aligned} \text{Var}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] &= E_{x \sim q} \left[\left(f(x) \frac{p(x)}{q(x)} \right)^2 \right] - \left(E_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right] \right)^2 \\ &= E_{x \sim p} \left[f(x)^2 \frac{p(x)}{q(x)} \right] - (E_{x \sim p}[f(x)])^2 \end{aligned} \quad (5.6)$$

$\text{Var}_{x \sim p}[f(x)]$ 和 $\text{Var}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$ 的差别在第一项是不同的, $\text{Var}_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$ 的第一项多乘了 $\frac{p(x)}{q(x)}$, 如果 $\frac{p(x)}{q(x)}$ 差距很大的话, $f(x) \frac{p(x)}{q(x)}$ 的方差就会很大。所以理论上它们的期望值一样, 也就是说, 你只要对 p 这个分布采样够多次, q 这个分布采样够多, 你得到的结果会是一样的。但是如果你采样的次数不够多, 因为它们的方差差距是很大的, 所以你就有可能得到非常大的差别。

$$E_{x \sim p}[f(x)] = E_{x \sim q} \left[f(x) \frac{p(x)}{q(x)} \right]$$

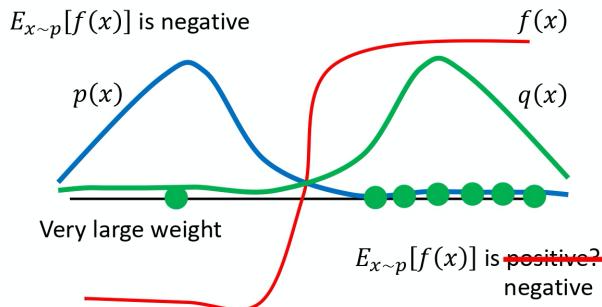


图 5.1 重要性采样的问题

举个例子, 当 $p(x)$ 和 $q(x)$ 差距很大的时候, 会发生什么样的问题。

假设蓝线是 $p(x)$ 的分布, 绿线是 $q(x)$ 的分布, 红线是 $f(x)$ 。如果我们要计算 $f(x)$ 的期望值, 从 $p(x)$ 这个分布做采样的话, 那显然 $E_{x \sim p}[f(x)]$ 是负的, 因为左边那块区域 $p(x)$ 的概率很高, 所以要采样的话, 都会采样到这个地方, 而 $f(x)$ 在这个区域是负的, 所以理论上这一项算出来会是负。

接下来我们改成从 $q(x)$ 这边做采样，因为 $q(x)$ 在右边这边的概率比较高，所以如果你采样的点不够的话，那你可能都只采样到右侧。如果你都只采样到右侧的话，你会发现说，算 $E_{x \sim q} [f(x) \frac{p(x)}{q(x)}]$ 这一项，搞不好还应该是正的。你这边采样到这些点，然后你去计算它们的 $f(x) \frac{p(x)}{q(x)}$ 都是正的。你采样到这些点都是正的。你取期望值以后也都是正的，这是因为你采样的次数不够多。假设你采样次数很少，你只能采样到右边这边。左边虽然概率很低，但也不是没有可能被采样到。假设你今天好不容易采样到左边的点，因为左边的点， $p(x)$ 和 $q(x)$ 是差很多的，这边 $p(x)$ 很大， $q(x)$ 很小。今天 $f(x)$ 好不容易终于采样到一个负的，这个负的就会被乘上一个非常大的权重，这样就可以平衡掉刚才那边一直采样到正的值的情况。最终你算出这一项的期望值，终究还是负的。但前提是你要采样够多次，这件事情才会发生。但有可能采样次数不够多， $E_{x \sim p}[f(x)]$ 跟 $E_{x \sim q} [f(x) \frac{p(x)}{q(x)}]$ 就有可能有很大的差距。这就是重要性采样的问题。

On-policy → Off-policy

$$\nabla \bar{R}_\theta = E_{\tau \sim p_\theta(\tau)} [R(\tau) \nabla \log p_\theta(\tau)]$$

- Use π_θ to collect data. When θ is updated, we have to sample training data again.
- Goal: Using the sample from $\pi_{\theta'}$ to train θ . θ' is fixed, so we can re-use the sample data.

$$\nabla \bar{R}_\theta = E_{\tau \sim p_{\theta'}(\tau)} \left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)} R(\tau) \nabla \log p_\theta(\tau) \right]$$

- Sample the data from θ' .
- Use the data to train θ many times.

<u>Importance Sampling</u>	$E_{x \sim p}[f(x)] = E_{x \sim q} [f(x) \frac{p(x)}{q(x)}]$
-----------------------------------	--

图 5.2

现在要做的事情就是把重要性采样用在 off-policy 的情况，把 on-policy 训练的算法改成 off-policy 训练的算法。

怎么改呢，之前我们是拿 θ 这个 policy 去跟环境做互动，采样出轨迹 τ ，然后计算 $R(\tau) \nabla \log p_\theta(\tau)$ 。现在我们不用 θ 去跟环境做互动，假设有另外一个 policy θ' ，它就是另外一个 actor。它的工作是去做示范 (demonstration)。 θ' 的工作是要去示范给 θ 看。它去跟环境做互动，告诉 θ 说，它跟环境做互动会发生什么事，借此来训练 θ 。我们要训练的是 θ ， θ' 只是负责做示范，负责跟环境做互动。

我们现在的 τ 是从 θ' 采样出来的，是拿 θ' 去跟环境做互动。所以采样出来的 τ 是从 θ' 采样出来的，这两个分布不一样。但没有关系，假设你本来是从 p 做采样，但你发现你不能从 p 做采样，所以我们不拿 θ 去跟环境做互动。你可以把 p 换 q ，然后在后面补上一个重要性权重。现在的状况就是一样，把 θ 换成 θ' 后，要补上一个重要性权重 $\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}$ 。这个重要性权重就是某一个轨迹 τ 用 θ 算出来的概率除以这个轨迹 τ 用 θ' 算出来的概率。这一项是很重要的，因为你要学习的是 actor θ 和 θ' 是不太一样的， θ' 会见到的情形跟 θ 见到的情形不见得是一样的，所以中间要做一个修正的项。

Q: 现在的数据是从 θ' 采样出来的，从 θ 换成 θ' 有什么好处呢？

A: 因为现在跟环境做互动是 θ' 而不是 θ 。所以采样出来的东西跟 θ 本身是没有关系的。所以你就可以让 θ' 做互动采样一大堆的数据， θ 可以更新参数很多次，一直到 θ 训练到一定的程度，更新很多次以后， θ' 再重新去做采样，这就是 on-policy 换成 off-policy 的妙用。

实际在做 policy gradient 的时候，我们并不是给整个轨迹 τ 都一样的分数，而是每一个状态-动作的对 (pair) 会分开来计算。实际上更新梯度的时候，如下式所示。

$$= E_{(s_t, a_t) \sim \pi_\theta} [A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n)] \quad (5.7)$$

我们用 θ 这个 actor 去采样出 s_t 跟 a_t ，采样出状态跟动作的对，我们会计算这个状态跟动作对的 advantage $A^\theta(s_t, a_t)$ ，就是它有多好。

On-policy → Off-policy

Gradient for update

$$\nabla f(x) = f(x) \nabla \log f(x)$$

$$\begin{aligned}
 &= E_{(s_t, a_t) \sim \pi_\theta} [A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n)] \\
 &\quad \boxed{A^{\theta'}(s_t, a_t)} \text{ This term is from sampled data.} \\
 &= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(s_t, a_t)}{p_{\theta'}(s_t, a_t)} A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \\
 &= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} \cancel{A^\theta(s_t, a_t)} \nabla \log p_\theta(a_t^n | s_t^n) \right] \\
 J^{\theta'}(\theta) &= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right] \text{ When to stop?}
 \end{aligned}$$

图 5.3

$A^\theta(s_t, a_t)$ 就是累积奖励减掉 bias，这一项就是估测出来的。它要估测的是，在状态 s_t 采取动作 a_t 是好的还是不好的。接下来后面会乘上 $\nabla \log p_\theta(a_t^n | s_t^n)$ ，也就是说如果 $A^\theta(s_t, a_t)$ 是正的，就要增加概率，如果是负的，就要减少概率。

我们通过重要性采样把 on-policy 变成 off-policy，从 θ 变成 θ' 。所以现在 s_t, a_t 是 θ' 跟环境互动以后所采样到的数据。但是拿来训练要调整参数是模型 θ 。因为 θ' 跟 θ 是不同的模型，所以你要做一个修正的项。这项修正的项，就是用重要性采样的技术，把 s_t, a_t 用 θ 采样出来的概率除掉 s_t, a_t 用 θ' 采样出来的概率。

$$= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(s_t, a_t)}{p_{\theta'}(s_t, a_t)} A^\theta(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.8)$$

$A^\theta(s_t, a_t)$ 有一个上标 θ ， θ 代表说这个是 actor θ 跟环境互动的时候所计算出来的 A。但是实际上从 θ 换到 θ' 的时候， $A^\theta(s_t, a_t)$ 应该改成 $A^{\theta'}(s_t, a_t)$ ，为什么？A 这一项是想要估测说现在在某一个状态采取某一个动作，接下来会得到累积奖励的值减掉 baseline。你怎么估 A 这一项，你就会看在状态 s_t ，采取动作 a_t ，接下来会得到的奖励的总和，再减掉 baseline。之前是 θ 在跟环境做互动，所以你观察到的是 θ 可以得到的奖励。但现在是 θ' 在跟环境做互动，所以你得到的这个 advantage，其实是根据 θ' 所估计出来的 advantage。但我们现在先不要管那么多，我们就假设这两项可能是差不多的。

接下来，我们可以拆解 $p_\theta(s_t, a_t)$ 和 $p_{\theta'}(s_t, a_t)$ ，即

$$\begin{aligned}
 p_\theta(s_t, a_t) &= p_\theta(a_t | s_t) p_\theta(s_t) \\
 p_{\theta'}(s_t, a_t) &= p_{\theta'}(a_t | s_t) p_{\theta'}(s_t)
 \end{aligned} \quad (5.9)$$

于是我们得到下式：

$$= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} \frac{p_\theta(s_t)}{p_{\theta'}(s_t)} A^{\theta'}(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.10)$$

这边需要做一件事情是，假设模型是 θ 的时候，你看到 s_t 的概率，跟模型是 θ' 的时候，你看到 s_t 的概率是差不多的，即 $p_\theta(s_t) = p_{\theta'}(s_t)$ 。因为它们是一样的，所以你可以把它删掉，即

$$= E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \nabla \log p_\theta(a_t^n | s_t^n) \right] \quad (5.11)$$

Q：为什么可以假设 $p_\theta(s_t)$ 和 $p_{\theta'}(s_t)$ 是差不多的？

A：因为你会看到什么状态往往跟你会采取什么样的动作是没有太大的关系的。比如说你玩不同的 Atari 的游戏，其实你看到的游戏画面都是差不多的，所以也许不同的 θ 对 s_t 是没有影响的。但更直觉的

理由就是 $p_\theta(s_t)$ 很难算，想想看这项要怎么算，这一项你还要说我有一个参数 θ ，然后拿 θ 去跟环境做互动，算 s_t 出现的概率，这个你很难算。尤其如果输入是图片的话，同样的 s_t 根本就不会出现第二次。你根本没有办法估这一项，所以干脆就无视这个问题。

但是 $p_\theta(a_t|s_t)$ 很好算。你手上有 θ 这个参数，它就是个网络。你就把 s_t 带进去， s_t 就是游戏画面，你把游戏画面带进去，它就会告诉你某一个状态的 a_t 概率是多少。我们有个 policy 的网络，把 s_t 带进去，它会告诉我们每一个 a_t 的概率是多少。所以 $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ 这一项，你只要知道 θ 和 θ' 的参数就可以算。

现在我们得到一个新的目标函数。

$$J^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} A^{\theta'}(s_t, a_t) \right] \quad (5.12)$$

式 (5.11) 是梯度，其实我们可以从梯度去反推原来的目标函数，我们可以用如下的公式来反推目标函数：

$$\nabla f(x) = f(x) \nabla \log f(x) \quad (5.13)$$

要注意一点，对 θ 求梯度时， $p_{\theta'}(a_t|s_t)$ 和 $A^{\theta'}(s_t, a_t)$ 都是常数。

所以实际上，当我们使用重要性采样的时候，要去优化的那一个目标函数就长这样子，我们把它写作 $J^{\theta'}(\theta)$ 。为什么写成 $J^{\theta'}(\theta)$ 呢，这个括号里面那个 θ 代表我们要去优化的那个参数。 θ' 是说我们拿 θ' 去做示范，就是现在真正在跟环境互动的是 θ' 。因为 θ 不跟环境做互动，是 θ' 在跟环境互动。

然后你用 θ' 去跟环境做互动，采样出 s_t 、 a_t 以后，你要去计算 s_t 跟 a_t 的 advantage，然后你再去把它乘上 $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ 。 $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ 是好算的， $A^{\theta'}(s_t, a_t)$ 可以从这个采样的结果里面去估测出来的，所以 $J^{\theta'}(\theta)$ 是可以算的。实际上在更新参数的时候，就是按照式 (5.11) 来更新参数。

5.2 PPO

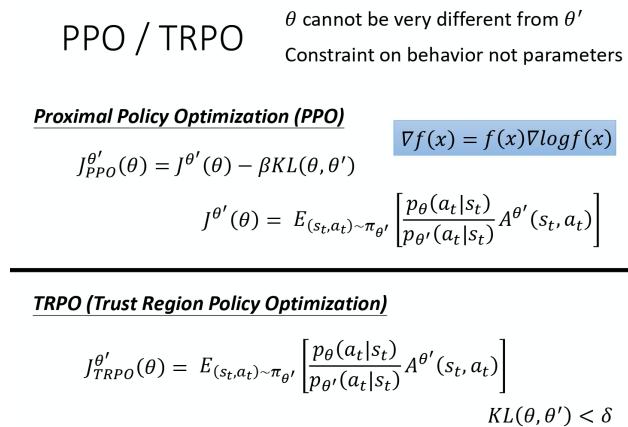


图 5.4

我们可以通过重要性采样把 on-policy 换成 off-policy，但重要性采样有一个问题：如果 $p_\theta(a_t|s_t)$ 跟 $p_{\theta'}(a_t|s_t)$ 差太多的话，这两个分布差太多的话，重要性采样的结果就会不好。怎么避免它差太多呢？这个就是 Proximal Policy Optimization (PPO) 在做的事情。

PPO 实际上做的事情就是这样，在 off-policy 的方法里要优化的是 $J^{\theta'}(\theta)$ 。但是这个目标函数又牵涉到重要性采样。在做重要性采样的时候， $p_\theta(a_t|s_t)$ 不能跟 $p_{\theta'}(a_t|s_t)$ 差太多。你做示范的模型不能够跟真正的模型差太多，差太多的话，重要性采样的结果就会不好。我们在训练的时候，多加一个约束 (constraint)。这个约束是 θ 跟 θ' 输出的动作的 KL 散度 (KL divergence)，简单来说，这一项的意思就是要衡量说 θ 跟 θ' 有多像。

然后我们希望在训练的过程中，学习出来的 θ 跟 θ' 越像越好。因为如果 θ 跟 θ' 不像的话，最后的结果就会不好。所以在 PPO 里面有两个式子，一方面是优化本来要优化的东西，但再加一个约束。这个约束就好像正则化 (regularization) 的项 (term) 一样，在做机器学习的时候不是有 L1/L2 的正则化。这一项也很像正则化，这样正则化做的事情就是希望最后学习出来的 θ 不要跟 θ' 太不一样。

PPO 有一个前身叫做[信任区域策略优化 \(Trust Region Policy Optimization, TRPO\)](#)，TRPO 的式子如下式所示：

$$J_{TRPO}^{\theta'}(\theta) = E_{(s_t, a_t) \sim \pi_{\theta'}} \left[\frac{p_\theta(a_t | s_t)}{p_{\theta'}(a_t | s_t)} A^{\theta'}(s_t, a_t) \right] \quad (5.14)$$

$$KL(\theta, \theta') < \delta$$

它与 PPO 不一样的地方是约束摆的位置不一样，PPO 是直接把约束放到你要优化的那个式子里面，然后你就可以用梯度上升的方法去最大化这个式子。但 TRPO 是把 KL 散度当作约束，它希望 θ 跟 θ' 的 KL 散度小于一个 δ 。如果你使用的是基于梯度的优化时，有约束是很难处理的。

TRPO 是很难处理的，因为它把 KL 散度约束当做一个额外的约束，没有放目标 (objective) 里面，所以它很难算。所以不想搬石头砸自己的脚的话，你就用 PPO 不要用 TRPO。看文献上的结果是，PPO 跟 TRPO 可能性能差不多，但 PPO 在实现上比 TRPO 容易的多。

Q: KL 散度到底指的是什么？

A:

这边我是直接把 KL 散度当做一个函数，输入是 θ 跟 θ' ，但我的意思并不是说把 θ 或 θ' 当做一个分布，算这两个分布之间的距离。所谓的 θ 跟 θ' 的距离并不是参数上的距离，而是行为 (behavior) 上的距离。

假设你有两个 actor，它们的参数分别为 θ 和 θ' ，所谓参数上的距离就是你算这两组参数有多像。这里讲的不是参数上的距离，而是它们行为上的距离。你先代进去一个状态 s ，它会对这个动作的空间输出一个分布。假设你有 3 个动作，3 个可能的动作就输出 3 个值。今天所指的距离是行为距离 (behavior distance)，也就是说，给定同样的状态，输出动作之间的差距。这两个动作的分布都是一个概率分布，所以就可以计算这两个概率分布的 KL 散度。把不同的状态输出的这两个分布的 KL 散度平均起来才是我这边所指的两个 actor 间的 KL 散度。

Q: 为什么不直接算 θ 和 θ' 之间的距离？算这个距离的话，甚至不要用 KL 散度算，L1 跟 L2 的范数 (norm) 也可以保证 θ 跟 θ' 很接近。

A: 在做强化学习的时候，之所以我们考虑的不是参数上的距离，而是动作上的距离，是因为很有可能对 actor 来说，参数的变化跟动作的变化不一定是完全一致的。有时候你参数小小变了一下，它可能输出的行为就差很多。或者是参数变很多，但输出的行为可能没什么改变。所以我们真正在意的是这个 actor 的行为上的差距，而不是它们参数上的差距。所以在做 PPO 的时候，所谓的 KL 散度并不是参数的距离，而是动作的距离。

5.2.1 PPO-Penalty

PPO 算法有两个主要的变种：PPO-Penalty 和 PPO-Clip。

我们来看一下 PPO1 的算法，即 PPO-Penalty。它先初始化一个 policy 的参数 θ^0 。然后在每一个迭代里面呢，你要用参数 θ^k ， θ^k 就是你在前一个训练的迭代得到的 actor 的参数，你用 θ^k 去跟环境做互动，采样到一大堆状态-动作的对。

然后你根据 θ^k 互动的结果，估测一下 $A^{\theta^k}(s_t, a_t)$ 。然后你就使用 PPO 的优化的公式。但跟原来的 policy gradient 不一样，原来的 policy gradient 只能更新一次参数，更新完以后，你就要重新采样数据。但是现在不用，你拿 θ^k 去跟环境做互动，采样到这组数据以后，你可以让 θ 更新很多次，想办法去最大化目标函数。这边 θ 更新很多次没有关系，因为我们已经有做重要性采样，所以这些经验，这些状态-动

PPO algorithm

- Initial policy parameters θ^0
- In each iteration
 - Using θ^k to interact with the environment to collect $\{s_t, a_t\}$ and compute advantage $A^{\theta^k}(s_t, a_t)$
 - Find θ optimizing $J_{PPO}(\theta)$

$$J^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)} A^{\theta^k}(s_t, a_t)$$

图 5.5

作的对是从 θ^k 采样出来的没有关系。 θ 可以更新很多次，它跟 θ^k 变得不太一样也没有关系，你还是可以照样训练 θ 。

$$J_{PPO}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta KL(\theta, \theta^k)$$

Update parameters several times

- If $KL(\theta, \theta^k) > KL_{max}$, increase β
- If $KL(\theta, \theta^k) < KL_{min}$, decrease β

Adaptive KL Penalty

图 5.6

在 PPO 的论文里面还有一个 adaptive KL divergence。这边会遇到一个问题就是 β 要设多少，它就跟正则化一样。正则化前面也要乘一个权重，所以这个 KL 散度前面也要乘一个权重，但 β 要设多少呢？所以有个动态调整 β 的方法。

在这个方法里面，你先设一个你可以接受的 KL 散度的最大值。假设优化完这个式子以后，你发现 KL 散度的项太大，那就代表说后面这个惩罚的项没有发挥作用，那就把 β 调大。另外，你设一个 KL 散度的最小值。如果优化完上面这个式子以后，你发现 KL 散度比最小值还要小，那代表后面这一项的效果太强了，你怕他只弄后面这一项，那 θ 跟 θ^k 都一样，这不是你要的，所以你要减少 β 。

所以 β 是可以动态调整的。这个叫做 adaptive KL penalty。

5.2.2 PPO-Clip

PPO algorithm

$$J_{PPO}^{\theta^k}(\theta) = J^{\theta^k}(\theta) - \beta KL(\theta, \theta^k)$$

$$J^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \frac{p_\theta(a_t | s_t)}{p_{\theta^k}(a_t | s_t)} A^{\theta^k}(s_t, a_t)$$

图 5.7

如果你觉得算 KL 散度很复杂，有一个 PPO2，PPO2 即 PPO-Clip。PPO2 要去最大化的目标函数如下式所示，它的式子里面就没有 KL 散度。

$$J_{PPo2}^{\theta^k}(\theta) \approx \sum_{(s_t, a_t)} \min \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} A^{\theta^k}(s_t, a_t), \right. \\ \left. \text{clip} \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}, 1 - \varepsilon, 1 + \varepsilon \right) A^{\theta^k}(s_t, a_t) \right) \quad (5.15)$$

这个式子看起来有点复杂，但实际实现就很简单。我们来看一下这个式子到底是什么意思。

- Min 这个操作符 (operator) 做的事情是第一项跟第二项里面选比较小的那个。
- 第二项前面有个 clip 函数，clip 函数的意思是说，
 - 在括号里面有三项，如果第一项小于第二项的话，那就输出 $1 - \varepsilon$ 。
 - 第一项如果大于第三项的话，那就输出 $1 + \varepsilon$ 。
- ε 是一个超参数，你要 tune 的，你可以设成 0.1 或设 0.2。

假设这边设 0.2 的话，如下式所示

$$\text{clip} \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}, 0.8, 1.2 \right) \quad (5.16)$$

如果 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 算出来小于 0.8，那就当作 0.8。如果算出来大于 1.2，那就当作 1.2。

我们先看看下面这项这个算出来到底是什么东西：

$$\text{clip} \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}, 1 - \varepsilon, 1 + \varepsilon \right) \quad (5.17)$$

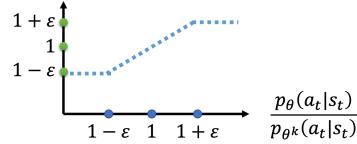


图 5.8

上图的横轴是 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ ，纵轴是 clip 函数的输出。

- 如果 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 大于 $1 + \varepsilon$ ，输出就是 $1 + \varepsilon$ 。
- 如果小于 $1 - \varepsilon$ ，它输出就是 $1 - \varepsilon$ 。
- 如果介于 $1 + \varepsilon$ 跟 $1 - \varepsilon$ 之间，就是输入等于输出。

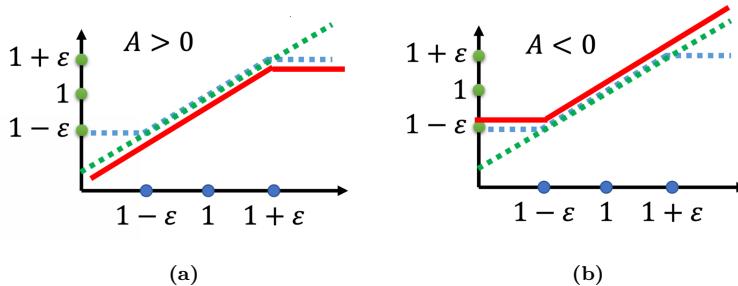


图 5.9

如图 5.9a 所示，

- $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 是绿色的线；
- $\text{clip} \left(\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}, 1 - \varepsilon, 1 + \varepsilon \right)$ 是蓝色的线；
- 在绿色的线跟蓝色的线中间，我们要取一个最小的。假设前面乘上的这个项 A，它是大于 0 的话，取最小的结果，就是红色的这一条线。

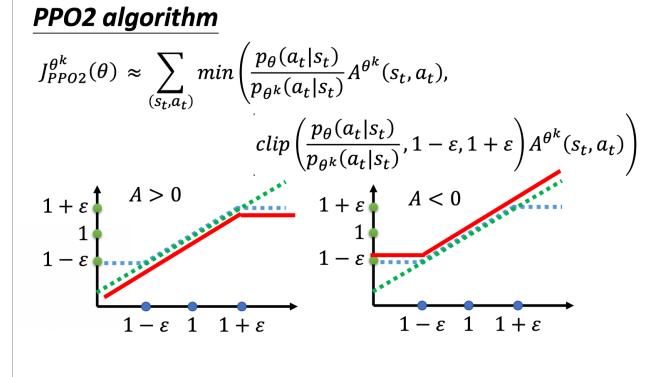


图 5.10

如图 5.9b 所示，如果 A 小于 0 的话，取最小的以后，就得到红色的这一条线。

虽然这个式子看起来有点复杂，实现起来是蛮简单的，因为这个式子想要做的事情就是希望 $p_\theta(a_t|s_t)$ 跟 $p_{\theta^k}(a_t|s_t)$ ，也就是你拿来做示范的模型跟你实际上学习的模型，在优化以后不要差距太大。

怎么让它做到不要差距太大呢？

- 如果 $A > 0$ ，也就是某一个状态-动作的对是好的，那我们希望增加这个状态-动作对的概率。也就是说，我们想要让 $p_\theta(a_t|s_t)$ 越大越好，但它跟 $p_{\theta^k}(a_t|s_t)$ 的比值不可以超过 $1 + \epsilon$ 。如果超过 $1 + \epsilon$ 的话，就没有 benefit 了。红色的线就是我们的目标函数，我们希望目标越大越好，我们希望 $p_\theta(a_t|s_t)$ 越大越好。但是 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)}$ 只要大过 $1 + \epsilon$ ，就没有 benefit 了。所以今天在训练的时候，当 $p_\theta(a_t|s_t)$ 被训练到 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} > 1 + \epsilon$ 时，它就会停止。假设 $p_\theta(a_t|s_t)$ 比 $p_{\theta^k}(a_t|s_t)$ 还要小，并且这个 advantage 是正的。因为这个动作是好的，我们当然希望这个动作被采取的概率越大越好，我们希望 $p_\theta(a_t|s_t)$ 越大越好。所以假设 $p_\theta(a_t|s_t)$ 还比 $p_{\theta^k}(a_t|s_t)$ 小，那就尽量把它挪大，但只要大到 $1 + \epsilon$ 就好。
- 如果 $A < 0$ ，也就是某一个状态-动作对是不好的，我们希望把 $p_\theta(a_t|s_t)$ 减小。如果 $p_\theta(a_t|s_t)$ 比 $p_{\theta^k}(a_t|s_t)$ 还大，那你就尽量把它压小，压到 $\frac{p_\theta(a_t|s_t)}{p_{\theta^k}(a_t|s_t)} = 1 - \epsilon$ 的时候就停了，就不要再压得更小。这样的好处就是，你不会让 $p_\theta(a_t|s_t)$ 跟 $p_{\theta^k}(a_t|s_t)$ 差距太大。要实现这个东西，很简单。

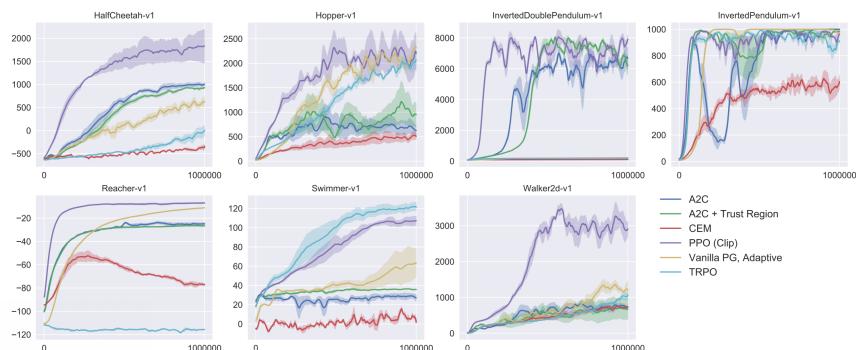


Figure 3: Comparison of several algorithms on several MuJoCo environments, training for one million timesteps.

图 5.11

上图是 PPO 跟其它方法的比较。Actor-Critic 和 A2C+Trust Region 方法是基于 actor-critic 的方法。PPO 是紫色线的方法，这边每张图就是某一个 RL 的任务，你会发现说在多数的情况 (cases) 里面，PPO 都是不错的，不是最好的，就是第二好的。

5.3 Keywords

- on-policy(同策略): 要 learn 的 agent 和环境互动的 agent 是同一个时, 对应的 policy。
- off-policy(异策略): 要 learn 的 agent 和环境互动的 agent 不是同一个时, 对应的 policy。
- important sampling (重要性采样): 使用另外一种数据分布, 来逼近所求分布的一种方法, 在强化学习中通常和蒙特卡罗方法结合使用, 公式如下: $\int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = E_{x \sim q}[f(x)\frac{p(x)}{q(x)}] = E_{x \sim p}[f(x)]$ 我们在已知 q 的分布后, 可以使用上述公式计算出从 p 这个 distribution sample x 代入 f 以后所算出来的期望值。
- Proximal Policy Optimization (PPO): 避免在使用 important sampling 时由于在 θ 下的 $p_\theta(a_t|s_t)$ 跟在 θ' 下的 $p_{\theta'}(a_t|s_t)$ 差太多, 导致 important sampling 结果偏差较大而采取的算法。具体来说就是在 training 的过程中增加一个 constrain, 这个 constrain 对应着 θ 跟 θ' output 的 action 的 KL divergence, 来衡量 θ 与 θ' 的相似程度。

5.4 Questions

- 基于 on-policy 的 policy gradient 有什么可改进之处? 或者说其效率较低的原因在于?
 - 经典 policy gradient 的大部分时间花在 sample data 处, 即当我们的 agent 与环境做了交互后, 我们就要进行 policy model 的更新。但是对于一个回合我们仅能更新 policy model 一次, 更新完后我们就要花时间去重新 collect data, 然后才能再次进行如上的更新。
 - 所以我们的可以自然而然地想到, 使用 off-policy 方法使用另一个不同的 policy 和 actor, 与环境进行互动并用 collect data 进行原先的 policy 的更新。这样等价于使用同一组 data, 在同一个回合, 我们对于整个的 policy model 更新了多次, 这样会更加有效率。
- 使用 important sampling 时需要注意的问题有哪些。

答: 我们可以在 important sampling 中将 p 替换为任意的 q , 但是本质上需要要求两者的分布不能差的太多, 即使我们补偿了不同数据分布的权重 $\frac{p(x)}{q(x)}$ 。 $E_{x \sim p}[f(x)] = E_{x \sim q}\left[f(x)\frac{p(x)}{q(x)}\right]$ 当我们对于两者的采样次数都比较多时, 最终的结果是一样的, 没有影响的。但是通常我们不会取理想的数量的 sample data, 所以如果两者的分布相差较大, 最后结果的 variance 差距 (平方级) 将会很大。
- 基于 off-policy 的 importance sampling 中的 data 是从 θ' sample 出来的, 从 θ 换成 θ' 有什么优势?

答: 使用 off-policy 的 importance sampling 后, 我们不用 θ 去跟环境做互动, 假设有另外一个 policy θ' , 它就是另外一个 actor。它的工作是他要做 demonstration, θ' 的工作是要去示范给 θ 看。它去跟环境做互动, 告诉 θ 说, 它跟环境做互动会发生什么事。然后, 借此来训练 θ 。我们要训练的是 θ , θ' 只是负责做 demo, 负责跟环境做互动, 所以 sample 出来的东西跟 θ 本身是没有关系的。所以你就可以让 θ' 做互动 sample 一大堆的 data, θ 可以 update 参数很多次。然后一直到 θ train 到一定的程度, update 很多次以后, θ' 再重新去做 sample, 这就是 on-policy 换成 off-policy 的妙用。
- 在本节中 PPO 中的 KL divergence 指的是什么?

答: 本质来说, KL divergence 是一个 function, 其度量的是两个 action (对应的参数分别为 θ 和 θ') 间的行为上的差距, 而不是参数上的差距。这里行为上的差距 (behavior distance) 可以理解为在相同的 state 的情况下, 输出的 action 的差距 (他们的概率分布上的差距), 这里的概率分布即为 KL divergence。

5.5 Something About Interview

- 高冷的面试官: 请问什么是重要性采样呀?

答：使用另外一种数据分布，来逼近所求分布的一种方法，算是一种期望修正的方法，公式是：

$$\int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx = E_{x \sim q}[f(x)\frac{p(x)}{q(x)}] = E_{x \sim p}[f(x)]$$

我们在已知 q 的分布后，可以使用上述公式计算出从 p 分布的期望值。也就可以使用 q 来对于 p 进行采样了，即为重要性采样。

- 高冷的面试官：请问 on-policy 跟 off-policy 的区别是什么？

答：用一句话概括两者的区别，生成样本的 policy (value-funciton) 和网络参数更新时的 policy (value-funciton) 是否相同。具体来说，on-policy：生成样本的 policy (value function) 跟网络更新参数时使用的 policy (value function) 相同。SARAS 算法就是 on-policy 的，基于当前的 policy 直接执行一次 action，然后用这个样本更新当前的 policy，因此生成样本的 policy 和学习时的 policy 相同，算法为 on-policy 算法。该方法会遭遇探索-利用的矛盾，仅利用目前已知的最优选择，可能学不到最优解，收敛到局部最优，而加入探索又降低了学习效率。epsilon-greedy 算法是这种矛盾下的折衷。优点是直接了当，速度快，劣势是不一定找到最优策略。off-policy：生成样本的 policy (value function) 跟网络更新参数时使用的 policy (value function) 不同。例如，Q-learning 在计算下一状态的预期收益时使用了 max 操作，直接选择最优动作，而当前 policy 并不一定能选择到最优动作，因此这里生成样本的 policy 和学习时的 policy 不同，即为 off-policy 算法。

- 高冷的面试官：请简述下 PPO 算法。其与 TRPO 算法有何关系呢？

答：PPO 算法的提出：旨在借鉴 TRPO 算法，使用一阶优化，在采样效率、算法表现，以及实现和调试的复杂度之间取得了新的平衡。这是因为 PPO 会在每一次迭代中尝试计算新的策略，让损失函数最小化，并且保证每一次新计算出的策略能够和原策略相差不大。具体来说，在避免使用 important sampling 时由于在 θ 下的 $p_\theta(a_t|s_t)$ 跟在 θ' 下的 $p_{\theta'}(a_t|s_t)$ 差太多，导致 important sampling 结果偏差较大而采取的算法。

References

- OpenAI Spinning Up

第 6 章 DQN

为了在连续的状态和动作空间中计算值函数 $Q^\pi(s, a)$ ，我们可以用一个函数 $Q_\phi(s, a)$ 来表示近似计算，称为**价值函数近似 (Value Function Approximation)**。

$$Q_\phi(s, a) \approx Q^\pi(s, a)$$

其中

- s, a 分别是状态 s 和动作 a 的向量表示，
- 函数 $Q_\phi(s, a)$ 通常是一个参数为 ϕ 的函数，比如神经网络，输出为一个实数，称为**Q 网络 (Q-network)**。

6.1 State Value Function

Q-learning 是 value-based 的方法。在 value-based 的方法里面，我们学习的不是策略，我们要学习的是一个 **critic(评论家)**。评论家要做的事情是评价现在的行为有多好或是有多不好。假设有一个演员 (actor) π ，评论家就是来评价这个演员的策略 π 好还是不好，即 **Policy Evaluation(策略评估)**。

注：「李宏毅深度强化学习」课程提到的 Q-learning，其实是 DQN(Deep Q-network)。

DQN 是指基于深度学习的 Q-learning 算法，主要结合了**值函数近似 (Value Function Approximation)**与神经网络技术，并采用了目标网络和经历回放的方法进行网络的训练。

在 Q-learning 中，我们使用表格来存储每个状态 s 下采取动作 a 获得的奖励，即状态-动作值函数 $Q(s, a)$ 。然而，这种方法在状态量巨大甚至是连续的任务中，会遇到维度灾难问题，往往是不可行的。因此，DQN 采用了价值函数近似的表示方法。

举例来说，有一种评论家叫做 **state value function(状态价值函数)**。状态价值函数的意思就是说，假设演员叫做 π ，拿 π 跟环境去做互动。假设 π 看到了某一个状态 s ，如果在玩 Atari 游戏的话，状态 s 是某一个画面，看到某一个画面的时候，接下来一直玩到游戏结束，期望的累积奖励有多大。所以 V^π 是一个函数，这个函数输入一个状态，然后它会输出一个标量 (scalar)。这个标量代表说， π 这个演员看到状态 s 的时候，接下来预期到游戏结束的时候，它可以得到多大的值。

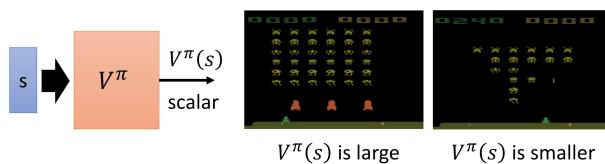


图 6.1

举个例子，假设你是玩 space invader 的话，

- 左边这个状态 s ，这个游戏画面， $V^\pi(s)$ 也许会很大，因为还有很多的怪可以杀，所以你会得到很大的分数。一直到游戏结束的时候，你仍然有很多的分数可以吃。
- 右边这种情况你得到的 $V^\pi(s)$ 可能就很小，因为剩下的怪也不多了，并且红色的防护罩已经消失了，所以可能很快就会死掉。所以接下来得到预期的奖励，就不会太大。

这边需要强调的一个点是说，评论家都是绑一个演员的，评论家没有办法去凭空去评价一个状态的好坏，它所评价的东西是在给定某一个状态的时候，假设接下来互动的演员是 π ，那我会得到多少奖励。因为就算是给同样的状态，你接下来的 π 不一样，你得到的奖励也是不一样的。

举例来说，在左边的情况，假设是一个正常的 π ，它可以杀很多怪，那假设它是一个很弱的 π ，它就站在原地不动，然后马上就被射死了，那你得到的 $V^\pi(s)$ 还是很小。所以评论家的输出值取决于状态和演

员。所以评论家其实都要绑一个演员，它是在衡量某一个演员的好坏，而不是衡量一个状态的好坏。这边要强调一下，评论家的输出是跟演员有关的，状态的价值其实取决于你的演员，当演员变的时候，状态价值函数的输出其实也是会跟着改变的。

6.1.1 State Value Function Estimation

怎么衡量这个状态价值函数 $V^\pi(s)$ 呢？有两种不同的做法：MC-based 的方法和 TD-based 的方法。

Monte-Carlo(MC)-based 的方法就是让演员去跟环境做互动，你要看演员好不好，你就让演员去跟环境做互动，给评论家看。然后，评论家就统计说，演员如果看到状态 s_a ，接下来的累积奖励会有多大。如果它看到状态 s_b ，接下来的累积奖励会有多大。

但是实际上，你不可能把所有的状态通通都扫过。如果你是玩 Atari 游戏的话，状态是图像，你没有办法把所有的状态通通扫过。所以实际上 $V^\pi(s)$ 是一个网络。对一个网络来说，就算输入状态是从来都没有看过的，它也可以想办法估测一个值的值。

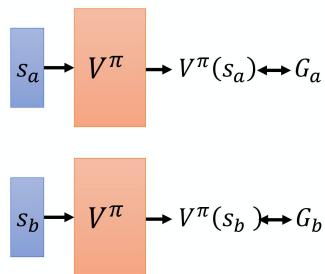


图 6.2

怎么训练这个网络呢？因为如果在状态 s_a ，接下来的累积奖励就是 G_a 。也就是说，对这个价值函数来说，如果输入是状态 s_a ，正确的输出应该是 G_a 。如果输入状态 s_b ，正确的输出应该是值 G_b 。所以在训练的时候，它就是一个回归问题 (regression problem)。网络的输出就是一个值，你希望在输入 s_a 的时候，输出的值跟 G_a 越近越好，输入 s_b 的时候，输出的值跟 G_b 越近越好。接下来把网络训练下去，就结束了。这是 MC-based 的方法。

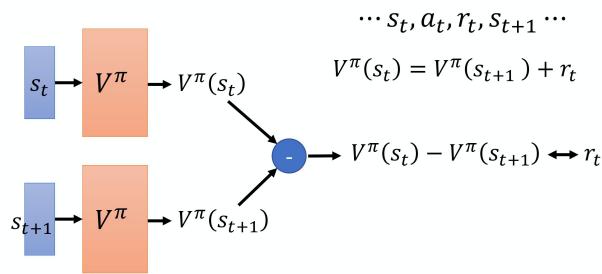


图 6.3

第二个方法是Temporal-difference(时序差分)的方法，即 TD-based 的方法。

在 MC-based 的方法中，每次我们都要算累积奖励，也就是从某一个状态 s_a 一直玩到游戏结束的时候，得到的所有奖励的总和。所以你要使用 MC-based 的方法，你必须至少把这个游戏玩到结束。但有些游戏非常长，你要玩到游戏结束才能够更新网络，花的时间太长了，因此我们会采用 TD-based 的方法。

TD-based 的方法不需要把游戏玩到底，只要在游戏的某一个情况，某一个状态 s_t 的时候，采取动作 a_t 得到奖励 r_t ，跳到状态 s_{t+1} ，就可以使用 TD 的方法。

怎么使用 TD 的方法呢？这边是基于以下这个式子：

$$V^\pi(s_t) = V^\pi(s_{t+1}) + r_t$$

假设我们现在用的是某一个策略 π ，在状态 s_t ，它会采取动作 a_t ，给我们奖励 r_t ，接下来进入 s_{t+1} 。状态 s_{t+1} 的值跟状态 s_t 的值，它们的中间差了一项 r_t 。因为你把 s_{t+1} 得到的值加上得到的奖励 r_t 就会等于 s_t 得到的值。有了这个式子以后，你在训练的时候，你并不是直接去估测 V ，而是希望你得到的结果 V 可以满足这个式子。

也就是说你会是这样训练的，你把 s_t 丢到网络里面，因为 s_t 丢到网络里面会得到 $V^\pi(s_t)$ ，把 s_{t+1} 丢到你的值网络里面会得到 $V^\pi(s_{t+1})$ ，这个式子告诉我们， $V^\pi(s_t)$ 减 $V^\pi(s_{t+1})$ 的值应该是 r_t 。然后希望它们两个相减的 loss 跟 r_t 越接近，训练下去，更新 V 的参数，你就可以把 V function 学习出来。

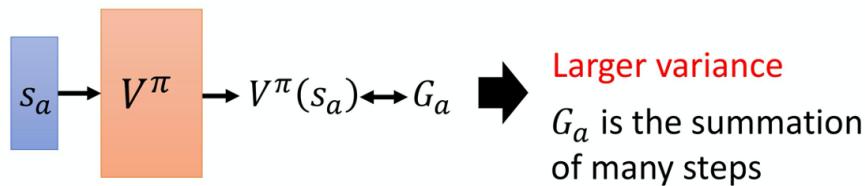


图 6.4

MC 跟 TD 有什么样的差别呢？

MC 最大的问题就是方差很大。因为我们在玩游戏的时候，它本身是有随机性的。所以你可以把 G_a 看成一个随机变量。因为你每次同样走到 s_a 的时候，最后你得到的 G_a 其实是不一样的。你看到同样的状态 s_a ，最后玩到游戏结束的时候，因为游戏本身是有随机性的，玩游戏的模型搞不好也有随机性，所以你每次得到的 G_a 是不一样的，每一次得到 G_a 的差别其实会很大。为什么它会很大呢？因为 G_a 其实是很多个不同的步骤的奖励的和。假设你每一个步骤都会得到一个奖励， G_a 是从状态 s_a 开始，一直玩到游戏结束，每一个步骤的奖励的和。

举例来说，通过下面式子，我们知道 G_a 的方差相较于某一个状态的奖励，它会是比较大的。

$$\text{Var}[kX] = k^2 \text{Var}[X]$$

其中，Var 是指 variance。

如果用 TD 的话，你是要去最小化这样的一个式子：

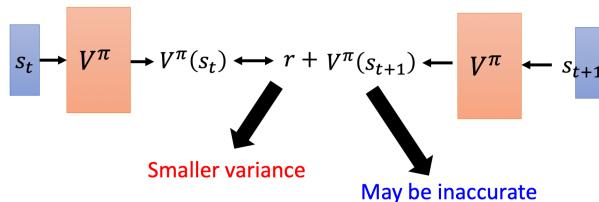


图 6.5

在这中间会有随机性的是 r 。因为计算你在 s_t 采取同一个动作，你得到的奖励也不一定是一样的，所以 r 是一个随机变量。但这个随机变量的方差会比 G_a 还要小，因为 G_a 是很多 r 合起来，这边只是某一个 r 而已。 G_a 的方差会比较大， r 的方差会比较小。但是这边你会遇到的一个问题是，你这个 V 不一定估得准。假设你的这个 V 估得是不准的，那你使用这个式子学习出来的结果，其实也会是不准的。所以 MC 跟 TD 各有优劣。今天其实 TD 的方法是比较常见的，MC 的方法其实是比较少用的。

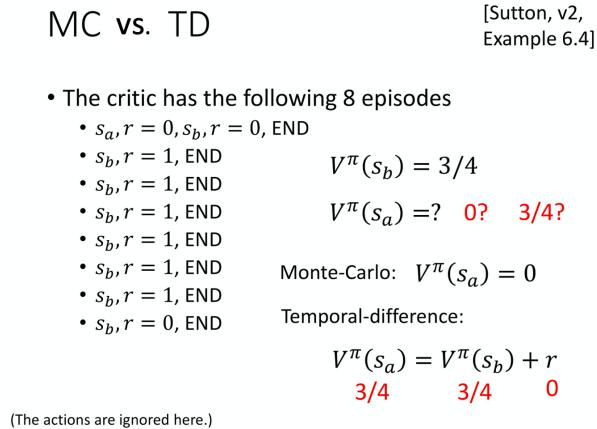


图 6.6

上图是讲 TD 跟 MC 的差异。假设有某一个评论家，它去观察某一个策略 π 跟环境互动的 8 个 episode 的结果。有一个演员 π 跟环境互动了 8 次，得到了 8 次玩游戏的结果。接下来这个评论家去估测状态的值。

我们先计算 s_b 的值。状态 s_b 在 8 场游戏里面都有经历过，其中有 6 场得到奖励 1，有 2 场得到奖励 0。所以如果你是要算期望值的话，就算看到状态 s_b 以后得到的奖励，一直到游戏结束的时候得到的累积奖励期望值是 $3/4$ ，计算过程如下式所示：

$$\frac{6 \times 1 + 2 \times 0}{8} = \frac{6}{8} = \frac{3}{4} \quad (6.1)$$

但 s_a 期望的奖励到底应该是多少呢？这边其实有两个可能的答案：一个是 0，一个是 $3/4$ 。为什么有两个可能的答案呢？这取决于你用 MC 还是 TD。用 MC 跟用 TD 算出来的结果是不一样的。

假如你用 MC 的话，你会发现这个 s_a 就出现一次，看到 s_a 这个状态，接下来累积奖励就是 0，所以 s_a 期望奖励就是 0。

但 TD 在计算的时候，它要更新下面这个式子：

$$V^\pi(s_a) = V^\pi(s_b) + r \quad (6.2)$$

因为我们在状态 s_a 得到奖励 $r=0$ 以后，跳到状态 s_b 。所以状态 s_b 的奖励会等于状态 s_b 的奖励加上在状态 s_a 跳到状态 s_b 的时候可能得到的奖励 r 。而这个得到的奖励 r 的值是 0， s_b 期望奖励是 $3/4$ ，那么 s_a 的奖励应该是 $3/4$ 。

用 MC 跟 TD 估出来的结果很有可能是不一样的。就算评论家观察到一样的训练数据，它最后估出来的结果也不一定是一样的。为什么会这样呢？你可能问说，哪一个结果比较对呢？其实就都对。

因为在第一个 trajectory， s_a 得到奖励 0 以后，再跳到 s_b 也得到奖励 0。这边有两个可能。

- 一个可能是： s_a 是一个标志性的状态，只要看到 s_a 以后， s_b 就会拿不到奖励， s_a 可能影响了 s_b 。如果是用 MC 的算法的话，它会把 s_a 影响 s_b 这件事考虑进去。所以看到 s_a 以后，接下来 s_b 就得不到奖励， s_b 期望的奖励是 0。
- 另一个可能是：看到 s_a 以后， s_b 的奖励是 0 这件事只是一个巧合，并不是 s_a 所造成，而是因为说 s_b 有时候就是会得到奖励 0，这只是单纯运气的问题。其实平常 s_b 会得到奖励期望值是 $3/4$ ，跟 s_a 是完全没有关系的。所以假设 s_a 之后会跳到 s_b ，那其实得到的奖励按照 TD 来算应该是 $3/4$ 。

所以不同的方法考虑了不同的假设，运算结果不同。

6.2 State-action Value Function(Q-function)

还有另外一种评论家叫做 Q-function。它又叫做 state-action value function(状态-动作价值函数)。

- 状态价值函数的输入是一个状态，它是根据状态去计算出，看到这个状态以后的期望的累积奖励 (expected accumulated reward) 是多少。
- 状态-动作价值函数的输入是一个状态、动作对，它的意思是说，在某一个状态采取某一个动作，假设我们都使用演员 π ，得到的累积奖励的期望值有多大。

Q -function 有一个需要注意的问题是，这个演员 π ，在看到状态 s 的时候，它采取的动作不一定是 a 。 Q -function 假设在状态 s 强制采取动作 a 。不管你现在考虑的这个演员 π ，它会不会采取动作 a ，这不重要。在状态 s 强制采取动作 a 。接下来都用演员 π 继续玩下去，就只有在状态 s ，我们才强制一定要采取动作 a ，接下来就进入自动模式，让演员 π 继续玩下去，得到的期望奖励才是 $Q^\pi(s, a)$ 。

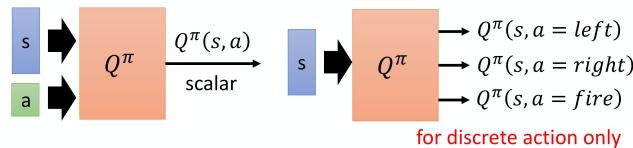


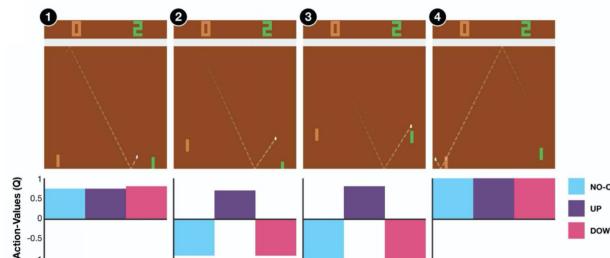
图 6.7

Q -function 有两种写法：

- 输入是状态跟动作，输出就是一个标量；
- 输入是一个状态，输出就是好几个值。

假设动作是离散的，动作就只有 3 个可能：往左往右或是开火。那这个 Q -function 输出的 3 个值就分别代表 a 是向左的时候的 Q 值， a 是向右的时候的 Q 值，还有 a 是开火的时候的 Q 值。

那你要注意的事情是，上图右边的函数只有离散动作才能够使用。如果动作是无法穷举的，你只能够用上图左边这个式子，不能够用右边这个式子。



[Mnih et al., 2015. Human-level control through deep reinforcement learning]

图 6.8

上图是文献上的结果，你去估计 Q -function 的话，看到的结果可能如上图所示。假设我们有 3 个动作：原地不动、向上、向下。

- 假设是在第一个状态，不管是采取哪个动作，最后到游戏结束的时候，得到的期望奖励其实都差不多。因为球在这个地方，就算是你向下，接下来你应该还可以急救。所以不管采取哪个动作，都差不多了太多。
- 假设在第二个状态，这个乒乓球它已经反弹到很接近边缘的地方，这个时候你采取向上，你才能得到正的奖励，才接的到球。如果你是站在原地不动或向下的话，接下来你都会错过这个球。你得到的奖励就会是负的。
- 假设在第三个状态，球很近了，所以就要向上。
- 假设在第四个状态，球被反弹回去，这时候采取哪个动作就都没有差了。

这是状态-动作价值的一个例子。

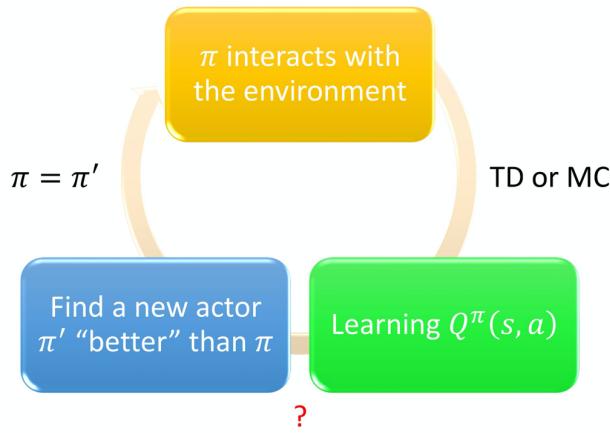


图 6.9

虽然表面上我们学习一个 Q-function，它只能拿来评估某一个演员 π 的好坏，但只要有了这个 Q-function，我们就可以做强化学习。有了这个 Q-function，我们就可以决定要采取哪一个动作，我们就可以进行策略改进 (Policy Improvement)。

它的大原则是这样，假设你有一个初始的演员，也许一开始很烂，随机的也没有关系。初始的演员叫做 π ，这个 π 跟环境互动，会收集数据。接下来你学习一个 π 这个演员的 Q 值，你去衡量一下 π 在某一个状态强制采取某一个动作，接下来用 π 这个策略会得到的期望奖励，用 TD 或 MC 都是可以的。你学习出一个 Q-function 以后，就保证你可以找到一个新的策略 π' ，policy π' 一定会比原来的策略 π 还要好。那等一下会定义说，什么叫做好。所以假设你有一个 Q-function 和某一个策略 π ，你根据策略 π 学习出策略 π 的 Q-function，接下来保证你可以找到一个新的策略 π' ，它一定会比 π 还要好，然后你用 π' 取代 π ，再去找它的 Q-function，得到新的以后，再去找一个更好的策略。这样一直循环下去，policy 就会越来越好。

首先要定义的是什么叫做比较好？我们说 π' 一定会比 π 还要好，什么叫做好呢？这边好是说，对所有可能的状态 s 而言， $V^{\pi'}(s) \geq V^\pi(s)$ 。也就是说我们走到同一个状态 s 的时候，如果拿 π 继续跟环境互动下去，我们得到的奖励一定会小于等于用 π' 跟环境互动下去得到的奖励。所以不管在哪一个状态，你用 π' 去做交互，得到的期望奖励一定会比较大。所以 π' 是比 π 还要好的一个策略。

有了 Q-function 以后，怎么找这个 π' 呢？如果你根据以下的这个式子去决定你的动作，

$$\pi'(s) = \arg \max_a Q^\pi(s, a) \quad (6.3)$$

根据上式去决定你的动作的步骤叫做 π' 的话，那 π' 一定会比 π 还要好。假设你已经学习出 π 的 Q-function，今天在某一个状态 s ，你把所有可能的动作 a 都一一带入这个 Q-function，看看哪一个 a 可以让 Q-function 的值最大，那这个动作就是 π' 会采取的动作。

这边要注意一下，给定这个状态 s ，你的策略 π 并不一定会采取动作 a ，我们是给定某一个状态 s 强制采取动作 a ，用 π 继续互动下去得到的期望奖励，这个才是 Q-function 的定义。所以在状态 s 里面不一定会采取动作 a 。用 π' 在状态 s 采取动作 a 跟 π 采取的动作是不一定会一样的， π' 所采取的动作会让它得到比较大的奖励。

所以这个 π' 是用 Q-function 推出来的，没有另外一个网络决定 π' 怎么交互，有 Q-function 就可以找出 π' 。

但是这边有另外一个问题是，在这边要解一个 $\arg \max$ 的问题，所以 a 如果是连续的就会有问题。如果是离散的， a 只有 3 个选项，一个一个带进去，看谁的 Q 最大，没有问题。但如果 a 是连续的，要解 $\arg \max$ 问题，你就会有问题。

接下来讲一下为什么用 $Q^\pi(s, a)$ 决定出来的 π' 一定会比 π 好。

假设有一个策略叫做 π' ，它是由 Q^π 决定的。我们要证对所有的状态 s 而言， $V^{\pi'}(s) \geq V^\pi(s)$ 。

怎么证呢？我们先把 $V^\pi(s)$ 写出来：

$$V^\pi(s) = Q^\pi(s, \pi(s))$$

假设在状态 s follow π 这个演员，它会采取的动作就是 $\pi(s)$ ，那你算出来的 $Q^\pi(s, \pi(s))$ 会等于 $V^\pi(s)$ 。一般而言， $Q^\pi(s, \pi(s))$ 不一定等于 $V^\pi(s)$ ，因为动作不一定是 $\pi(s)$ 。但如果这个动作是 $\pi(s)$ 的话， $Q^\pi(s, \pi(s))$ 是等于 $V^\pi(s)$ 的。

$Q^\pi(s, \pi(s))$ 还满足如下的关系：

$$Q^\pi(s, \pi(s)) \leq \max_a Q^\pi(s, a) \quad (6.4)$$

因为 a 是所有动作里面可以让 Q 最大的那个动作，所以今天这一项一定会比它大。这一项就是 $Q^\pi(s, a)$ ， a 就是 $\pi'(s)$ 。因为 $\pi'(s)$ 输出的 a 就是可以让 $Q^\pi(s, a)$ 最大的那一个，所以我们得到了下面的式子：

$$\max_a Q^\pi(s, a) = Q^\pi(s, \pi'(s)) \quad (6.5)$$

于是：

$$V^\pi(s) \leq Q^\pi(s, \pi'(s))$$

也就是说某一个状态，如果你按照策略 π 一直做下去，你得到的奖励一定会小于等于，在这个状态 s 你故意不按照 π 所给你指示的方向，而是按照 π' 的方向走一步，但只有第一步是按照 π' 的方向走，只有在状态 s 这个地方，你才按照 π' 的指示走，接下来你就按照 π 的指示走。虽然只有一步之差，但是从上面这个式子可知，虽然只有一步之差，但你得到的奖励一定会比完全 follow π 得到的奖励还要大。

接下来要证下面的式子：

$$Q^\pi(s, \pi'(s)) \leq V^{\pi'}(s)$$

也就是说，只有一步之差，你会得到比较大的奖励。但假设每步都是不一样的，每步都是 follow π' 而不是 π 的话，那你得到的奖励一定会更大。如果你要用数学式把它写出来的话，你可以写成 $Q^\pi(s, \pi'(s))$ ，它的意思就是说，我们在状态 s_t 采取动作 a_t ，得到奖励 r_t ，然后跳到状态 s_{t+1} ，即如下式所示：

$$Q^\pi(s, \pi'(s)) = E[r_t + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)]$$

在文献上有时也会说：在状态 s_t 采取动作 a_t 得到奖励 r_{t+1} ，有人会写成 r_t ，但意思其实都是一样的。

在状态 s 按照 π' 采取某一个动作 a_t ，得到奖励 r_t ，然后跳到状态 s_{t+1} ， $V^\pi(s_{t+1})$ 是状态 s_{t+1} 根据 π 这个演员所估出来的值。因为在同样的状态采取同样的动作，你得到的奖励和会跳到的状态不一定一样，所以这边需要取一个期望值。

因为 $V^\pi(s) \leq Q^\pi(s, \pi'(s))$ ，也就是 $V^\pi(s_{t+1}) \leq Q^\pi(s_{t+1}, \pi'(s_{t+1}))$ ，所以我们得到下式：

$$\begin{aligned} & E[r_t + V^\pi(s_{t+1}) | s_t = s, a_t = \pi'(s_t)] \\ & \leq E[r_t + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \end{aligned} \quad (6.6)$$

因为 $Q^\pi(s_{t+1}, \pi'(s_{t+1})) = r_{t+1} + V^\pi(s_{t+2})$ ，所以我们得到下式：

$$\begin{aligned} & E[r_t + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\ & = E[r_t + r_{t+1} + V^\pi(s_{t+2}) | s_t = s, a_t = \pi'(s_t)] \end{aligned} \quad (6.7)$$

然后你再代入 $V^\pi(s) \leq Q^\pi(s, \pi'(s))$, 一直算到回合结束, 即:

$$\begin{aligned}
 V^\pi(s) &\leq Q^\pi(s, \pi'(s)) \\
 &= E[r_t + Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = \pi'(s_t)] \\
 &= E[r_t + r_{t+1} + V^\pi(s_{t+2}) | s_t = s, a_t = \pi'(s_t)] \\
 &\leq E[r_t + r_{t+1} + Q^\pi(s_{t+2}, \pi'(s_{t+2})) | s_t = s, a_t = \pi'(s_t)] \\
 &= E[r_t + r_{t+1} + r_{t+2} + V^\pi(s_{t+3}) | s_t = s, a_t = \pi'(s_t)] \\
 &\leq \dots \\
 &\leq E[r_t + r_{t+1} + r_{t+2} + \dots | s_t = s, a_t = \pi'(s_t)] \\
 &= V^{\pi'}(s)
 \end{aligned} \tag{6.8}$$

因此:

$$V^\pi(s) \leq V^{\pi'}(s)$$

从这边我们可以知道, 你可以估计某一个策略的 Q-function, 接下来你就可以找到另外一个策略 π' 比原来的策略还要更好。

6.3 Target Network

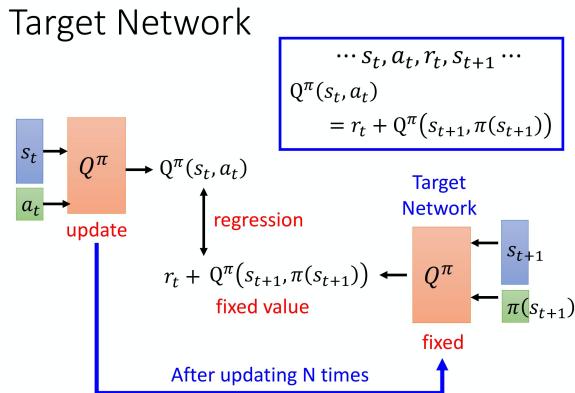


图 6.10

接下来讲一下在 DQN 里你一定会用到的 tip。第一个是 目标网络 (target network), 什么意思呢? 我们在学习 Q-function 的时候, 也会用到 TD 的概念。那怎么用 TD? 你现在收集到一个数据, 是说在状态 s_t , 你采取动作 a_t 以后, 你得到奖励 r_t , 然后跳到状态 s_{t+1} 。然后根据这个 Q-function, 你会知道说

$$Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$$

所以你在学习的时候, 你会说我们有 Q-function, 输入 s_t, a_t 得到的值, 跟输入 $s_{t+1}, \pi(s_{t+1})$ 得到的值中间, 我们希望它差了一个 r_t , 这跟刚才讲的 TD 的概念是一样的。

但是实际上这样的一个输入并不好学习, 因为假设这是一个回归问题, $Q^\pi(s_t, a_t)$ 是网络的输出, $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 是目标, 你会发现目标是会动的。当然你要实现这样的训练, 其实也没有问题, 就是你在做反向传播的时候, Q^π 的参数会被更新, 你会把两个更新的结果加在一起。因为它们是同一个模型 Q^π , 所以两个更新的结果会加在一起。但这样会导致训练变得不太稳定, 因为假设你把 $Q^\pi(s_t, a_t)$ 当作你模型的输出, $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 当作目标的话, 你要去拟合的目标是一直在变的, 这种一直在变的目标的训练是不太好训练的。

所以你会把其中一个 Q 网络，通常是你会把右边这个 Q 网络固定住。也就是说你在训练的时候，你只更新左边的 Q 网络的参数，而右边的 Q 网络的参数会被固定住。因为右边的 Q 网络负责产生目标，所以叫目标网络。因为目标网络是固定的，所以你现在得到的目标 $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 的值也是固定的。因为目标网络是固定的，我们只调左边网络的参数，它就变成是一个回归问题。我们希望模型的输出的值跟目标越接近越好，你会最小化它的均方误差 (mean square error)。

在实现的时候，你会把左边的 Q 网络更新好几次以后，再去用更新过的 Q 网络替换这个目标网络。但它们两个不要一起动，它们两个一起动的话，结果会很容易坏掉。

一开始这两个网络是一样的，然后在训练的时候，你会把右边的 Q 网络固定住。你在做梯度下降的时候，只调左边这个网络的参数，那你可能更新 100 次以后才把这个参数复制到右边的网络去，把它盖过去。把它盖过去以后，你这个目标值就变了。就好像说你本来在做一个回归问题，那你训练后把这个回归问题的 loss 压下去以后，接下来你把这边的参数把它复制过去以后，你的目标就变掉了，接下来就要重新再训练。

6.3.1 Intuition



图 6.11

我们可以通过猫追老鼠的例子来直观地理解为什么要 fix target network。猫是 Q estimation，老鼠是 Q target。一开始的话，猫离老鼠很远，所以我们想让这个猫追上老鼠。

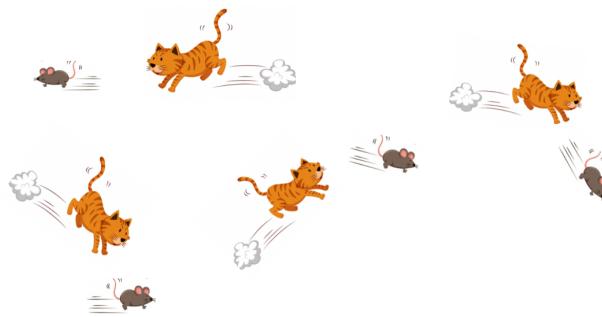


图 6.12

因为 Q target 也是跟模型参数相关的，所以每次优化后，Q target 也会动。这就导致一个问题，猫和老鼠都在动。

然后它们就会在优化空间里面到处乱动，就会产生非常奇怪的优化轨迹，这就使得训练过程十分不稳定。所以我们可以固定 Q target，让老鼠动得不是那么频繁，可能让它每 5 步动一次，猫则是每一步都在动。如果老鼠每 5 次动一步的话，猫就有足够的时间来接近老鼠。然后它们之间的距离会随着优化过程越来越小，最后它们就可以拟合，拟合过后就可以得到一个最好的 Q 网络。



图 6.13

6.4 Exploration

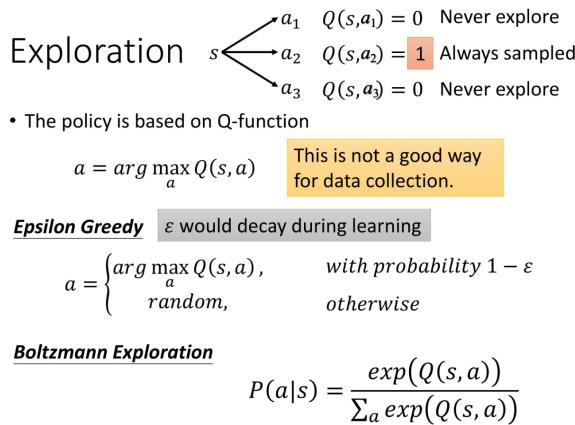


图 6.14

第二个 tip 是探索 (Exploration)。当我们使用 Q-function 的时候，policy 完全取决于 Q-function。给定某一个状态，你就穷举所有的 a ，看哪个 a 可以让 Q 值最大，它就是采取的动作。这个跟策略梯度不一样，在做策略梯度的时候，输出其实是随机的。我们输出一个动作的分布，根据这个动作的分布去做采样，所以在策略梯度里面，你每次采取的动作是不一样的，是有随机性的。

像这种 Q-function，如果你采取的动作总是固定的，会有什么问题呢？你会遇到的问题就是这不是一个好的收集数据的方式。因为假设我们今天真的要估某一个状态，你可以采取动作 a_1, a_2, a_3 。你要估测在某一个状态采取某一个动作会得到的 Q 值，你一定要在那个状态采取过那个动作，才估得出它的值。如果你没有在那个状态采取过那个动作，你其实估不出那个值的。如果是用深的网络，就你的 Q-function 是一个网络，这种情形可能会没有那么严重。但是一般而言，假设 Q-function 是一个表格，没有看过的 state-action pair，它就是估不出值来。网络也是会有一样的问题，只是没有那么严重。所以今天假设你在某一个状态，动作 a_1, a_2, a_3 你都没有采取过，那你估出来的 $Q(s, a_1), Q(s, a_2), Q(s, a_3)$ 的值可能都是一样的，就都是一个初始值，比如说 0，即

$$\begin{aligned} Q(s, a_1) &= 0 \\ Q(s, a_2) &= 0 \\ Q(s, a_3) &= 0 \end{aligned}$$

但是假设你在状态 s ，你采样过某一个动作 a_2 ，它得到的值是正的奖励。那 $Q(s, a_2)$ 就会比其他的动作都要好。在采取动作的时候，就看谁的 Q 值最大就采取谁，所以之后你永远都只会采样到 a_2 ，其他的动作就再也不会被做了，所以就会有问题。就好像说你进去一个餐厅吃饭，其实你都很难选。你今天

点了某一个东西以后，假说点了某一样东西，比如说椒麻鸡，你觉得还可以。接下来你每次去就都会点椒麻鸡，再也不会点别的东西了，那你就知道说别的东西是不是会比椒麻鸡好吃，这个是一样的问题。

如果你没有好的探索的话，你在训练的时候就会遇到这种问题。举个例子，假设你用 DQN 来玩 slither.io。你会有一个蛇，它在环境里面走来走去，吃到星星，它就加分。假设这个游戏一开始，它往上走，然后就吃到那个星星，它就得到分数，它就知道说往上走可以得到奖励。接下来它就再也不会采取往上走以外的动作了，所以接下来就会变成每次游戏一开始，它就往上冲，然后就死掉。所以需要有探索的机制，让机器知道说，虽然根据之前采样的结果， a_2 好像是不错的，但你至少偶尔也试一下 a_1 跟 a_3 ，说不定它们更好。

这个问题其实就是探索-利用窘境 (Exploration-Exploitation dilemma) 问题。

有两个方法解这个问题，一个是 Epsilon Greedy。Epsilon Greedy(ε -greedy) 的意思是说，我们有 $1 - \varepsilon$ 的概率会按照 Q-function 来决定动作，通常 ε 就设一个很小的值， $1 - \varepsilon$ 可能是 90%，也就是 90% 的概率会按照 Q-function 来决定动作，但是你有 10% 的机率是随机的。通常在实现上 ε 会随着时间递减。在最开始的时候。因为还不知道那个动作是比较好的，所以你会花比较大的力气在做探索。接下来随着训练的次数越来越多。已经比较确定说哪一个 Q 是比较好的。你就会减少你的探索，你会把 ε 的值变小，主要根据 Q-function 来决定你的动作，比较少随机决定动作，这是 Epsilon Greedy。

还有一个方法叫做 Boltzmann Exploration，这个方法就比较像是策略梯度。在策略梯度里面，网络的输出是一个期望的动作空间上面的一个的概率分布，再根据概率分布去做采样。那其实你也可以根据 Q 值去定一个概率分布，假设某一个动作的 Q 值越大，代表它越好，我们采取这个动作的机率就越高。但是某一个动作的 Q 值小，不代表我们不能尝试。

Q: 我们有时候也要尝试那些 Q 值比较差的动作，怎么做呢？

A: 因为 Q 值是有正有负的，所以可以把它弄成一个概率，你先取指数，再做归一化。然后把 $\exp(Q(s, a))$ 做归一化的这个概率当作是你在决定动作的时候采样的概率。在实现上，Q 是一个网络，所以你有点难知道，在一开始的时候网络的输出到底会长什么样子。假设你一开始没有任何的训练数据，参数是随机的，那给定某一个状态 s，不同的 a 输出的值可能就是差不多的，所以一开始 $Q(s, a)$ 应该会倾向于均匀的。也就是在一开始的时候，你这个概率分布算出来，它可能是比较均匀的。

6.5 Experience Replay

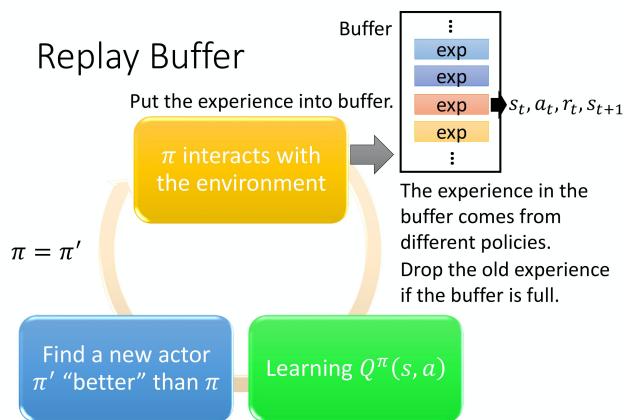


图 6.15

第三个 tip 是 Experience Replay(经验回放)。Experience Replay 会构建一个 Replay Buffer，Replay Buffer 又被称为 Replay Memory。Replay Buffer 是说现在会有某一个策略 π 去跟环境做互动，然后它会去收集数据。我们会把所有的数据放到一个 buffer 里面，buffer 里面就存了很多数据。比如说 buffer 是 5

万，这样它里面可以存 5 万笔资料，每一笔资料就是记得说，我们之前在某一个状态 s_t ，采取某一个动作 a_t ，得到了奖励 r_t 。然后跳到状态 s_{t+1} 。那你用 π 去跟环境互动很多次，把收集到的资料都放到这个 replay buffer 里面。

这边要注意是 replay buffer 里面的经验可能是来自于不同的策略，你每次拿 π 去跟环境互动的时候，你可能只互动 10000 次，然后接下来你就更新你的 π 了。但是这个 buffer 里面可以放 5 万笔资料，所以 5 万笔资料可能是来自于不同的策略。Buffer 只有在它装满的时候，才会把旧的资料丢掉。所以这个 buffer 里面它其实装了很多不同的策略的经验。

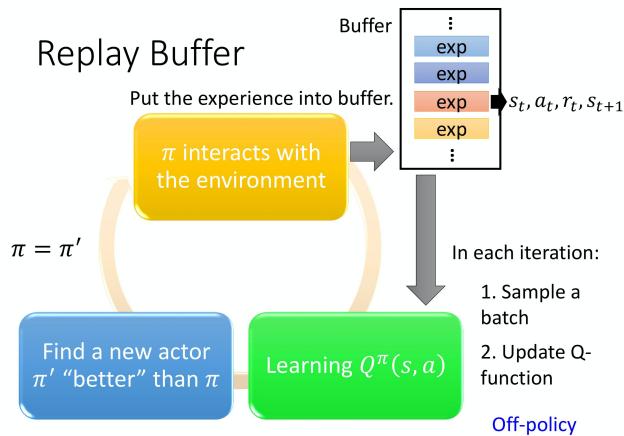


图 6.16

有了这个 buffer 以后，你是怎么训练这个 Q 的模型呢，怎么估 Q-function？你的做法是这样：你会迭代地去训练这个 Q-function，在每次迭代里面，你从这个 buffer 里面随机挑一个 batch 出来，就跟一般的网络训练一样，你从那个训练集里面，去挑一个 batch 出来。你去采样一个 batch 出来，里面有一把的经验，根据这把经验去更新你的 Q-function。就跟 TD learning 要有一个目标网络是一样的。你去采样一堆 batch，采样一个 batch 的数据，采样一堆经验，然后再去更新你的 Q-function。

当我们这么做的时候，它变成了一个 off-policy 的做法。因为本来我们的 Q 是要观察 π 的经验，但实际上存在你的 replay buffer 里面的这些经验不是通通来自于 π ，有些是过去其他的 π 所遗留下来的经验。因为你不会拿某一个 π 就把整个 buffer 装满，然后拿去测 Q-function，这个 π 只是采样一些数据塞到那个 buffer 里面去，然后接下来就让 Q 去训练。所以 Q 在采样的时候，它会采样到过去的一些资料。

这么做有两个好处。

第一个好处，其实在做强化学习的时候，往往最花时间的步骤是在跟环境做互动，训练网络反而是比较快的。因为你用 GPU 训练其实很快，真正花时间的往往是在跟环境做互动。用 replay buffer 可以减少跟环境做互动的次数，因为在做训练的时候，你的经验不需要通通来自于某一个策略。一些过去的策略所得到的经验可以放在 buffer 里面被使用很多次，被反复的再利用，这样让你的采样到经验的利用是比较高效的。

第二个好处，在训练网络的时候，其实我们希望一个 batch 里面的数据越多样 (diverse) 越好。如果你的 batch 里面的数据都是同样性质的，你训练下去是容易坏掉的。如果 batch 里面都是一样的数据，你训练的时候，performance 会比较差。我们希望 batch 的数据越多样越好。那如果 buffer 里面的那些经验通通来自于不同的策略，那你采样到的一个 batch 里面的数据会是比较多样化的。

Q：我们明就是要观察 π 的值，里面混杂了一些不是 π 的经验，这有没有关系？

A：没关系。这并不是因为过去的 π 跟现在的 π 很像，就算过去的 π 没有很大差异，其实也是没有关系的。主要的原因是因为，我们并不是去采样一个 trajectory，我们只采样了一笔经验，所以跟是不是 off-policy 这件事是没有关系的。就算是 off-policy，就算是这些经验不是来自于 π ，我们其实还是可以拿这些经验来估测 $Q^\pi(s, a)$ 。这件事有点难解释，不过你就记得说 Experience Replay 在理论上也是没有问题的。

6.6 DQN

Deep Q-network(DQN) Algorithm

- Initialize Q-function Q , target Q-function $\hat{Q} = Q$
- In each episode
 - For each time step t
 - Given state s_t , take action a_t based on Q (epsilon greedy)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
 - Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
 - Every C steps reset $\hat{Q} = Q$

图 6.17

上图就是一般的 Deep Q-network(DQN) 的算法。

这个算法是这样的。初始化的时候，你初始化 2 个网络：Q 和 \hat{Q} ，其实 \hat{Q} 就等于 Q。一开始这个目标 Q 网络，跟你原来的 Q 网络是一样的。在每一个 episode，你拿你的演员去跟环境做互动，在每一次互动的过程中，你都会得到一个状态 s_t ，那你会采取某一个动作 a_t 。怎么知道采取哪一个动作 a_t 呢？你就根据你现在的 Q-function。但是你要有探索的机制。比如说你用 Boltzmann 探索或是 Epsilon Greedy 的探索。那接下来你得到奖励 r_t ，然后跳到状态 s_{t+1} 。所以现在收集到一笔数据，这笔数据是 (s_t, a_t, r_t, s_{t+1}) 。这笔数据就塞到你的 buffer 里面去。如果 buffer 满的话，你就再把一些旧的资料丢掉。接下来你就从你的 buffer 里面去采样数据，那你采样到的是 (s_i, a_i, r_i, s_{i+1}) 。这笔数据跟你刚放进去的不一定是同一笔，你可能抽到一个旧的。要注意的是，其实你采样出来不是一笔数据，你采样出来的是一个 batch 的数据，你采样一个 batch 出来，采样一把经验出来。接下来就是计算你的目标。假设你采样出这么一笔数据。根据这笔数据去算你的目标。你的目标是什么呢？目标记得要用目标网络 \hat{Q} 来算。目标是：

$$y = r_i + \max_a \hat{Q}(s_{i+1}, a) \quad (6.9)$$

其中 a 就是让 \hat{Q} 的值最大的 a。因为我们在状态 s_{i+1} 会采取的动作 a，其实就是那个可以让 Q 值最大的那一个 a。接下来我们要更新 Q 的值，那就把它当作一个回归问题。希望 $Q(s_i, a_i)$ 跟你的目标越接近越好。然后假设已经更新了某一个数量的次，比如说 C 次，设 $C = 100$ ，那你就把 \hat{Q} 设成 Q，这就是 DQN。

Q: DQN 和 Q-learning 有什么不同？

A: 整体来说，DQN 与 Q-learning 的目标价值以及价值的更新方式都非常相似，主要的不同点在于：

- DQN 将 Q-learning 与深度学习结合，用深度网络来近似动作价值函数，而 Q-learning 则是采用表格存储；
- DQN 采用了经验回放的训练方法，从历史数据中随机采样，而 Q-learning 直接采用下一个状态的数据进行学习。

6.7 Keywords

- DQN(Deep Q-Network): 基于深度学习的 Q-learning 算法, 其结合了 Value Function Approximation (价值函数近似) 与神经网络技术, 并采用了目标网络 (Target Network) 和经验回放 (Experience Replay) 等方法进行网络的训练。
- State-value Function: 本质是一种 critic。其输入为 actor 某一时刻的 state, 对应的输出为一个标量, 即当 actor 在对应的 state 时, 预期的到过程结束时间段中获得的 value 的数值。
- State-value Function Bellman Equation: 基于 state-value function 的 Bellman Equation, 它表示在状态 s_t 下带来的累积奖励 G_t 的期望。
- Q-function: 其也被称为 state-action value function。其 input 是一个 state 跟 action 的 pair, 即在某一个 state 采取某一个 action, 假设我们都使用 actor π , 得到的 accumulated reward 的期望值有多大。
- Target Network: 为了解决在基于 TD 的 Network 的问题时, 优化目标 $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 左右两侧会同时变化使得训练过程不稳定, 从而增大 regression 的难度。target network 选择将上式的右部分即 $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 固定, 通过改变上式左部分的 network 的参数, 进行 regression, 这也是一个 DQN 中比较重要的 tip。
- Exploration: 在我们使用 Q-function 的时候, 我们的 policy 完全取决于 Q-function, 有可能导致出现对应的 action 是固定的某几个数值的情况, 而不像 policy gradient 中的 output 为随机的, 我们再从随机的 distribution 中 sample 选择 action。这样会导致我们继续训练的 input 的值一样, 从而“加重”output 的固定性, 导致整个模型的表达能力的急剧下降, 这也就是探索-利用窘境难题 (Exploration-Exploitation dilemma)。所以我们使用 Epsilon Greedy 和 Boltzmann Exploration 等 Exploration 方法进行优化。
- Experience Replay (经验回放): 其会构建一个 Replay Buffer (Replay Memory), 用来保存许多 data, 每一个 data 的形式如下: 在某一个 state s_t , 采取某一个 action a_t , 得到了 reward r_t , 然后跳到 state s_{t+1} 。我们使用 π 去跟环境互动很多次, 把收集到的数据都放到这个 replay buffer 中。当我们的 buffer “装满” 后, 就会自动删去最早进入 buffer 的 data。在训练时, 对于每一轮迭代都有相对应的 batch (与我们训练普通的 Network 一样通过 sample 得到), 然后用这个 batch 中的 data 去 update 我们的 Q-function。综上, Q-function 再 sample 和训练的时候, 会用到过去的经验数据, 所以这里称这个方法为 Experience Replay, 其也是 DQN 中比较重要的 tip。

6.8 Questions

- 为什么在 DQN 中采用价值函数近似 (Value Function Approximation) 的表示方法?
答: 首先 DQN 为基于深度学习的 Q-learning 算法, 而在 Q-learning 中, 我们使用表格来存储每一个 state 下 action 的 reward, 即我们前面所讲的状态-动作值函数 $Q(s, a)$ 。但是在我们的实际任务中, 状态量通常数量巨大并且在连续的任务中, 会遇到维度灾难的问题, 所以使用真正的 Value Function 通常是不切实际的, 所以使用了价值函数近似 (Value Function Approximation) 的表示方法。
- critic output 通常与哪几个值直接相关?
答: critic output 与 state 和 actor 有关。我们在讨论 output 时通常是对一个 actor 下来衡量一个 state 的好坏, 也就是 state value 本质上来说是依赖于 actor。不同的 actor 在相同的 state 下也会有不同的 output。
- 我们通常怎么衡量 state value function $V^\pi(s)$? 分别的优势和劣势有哪些?
答: - 基于 Monte-Carlo (MC) 的方法: 本质上就是让 actor 与 environment 做互动。critic 根据“统计”的结果, 将 actor 和 state 对应起来, 即当 actor 如果看到某一 state s_a , 将预测接下来的 accumulated reward 有多大如果它看到 state s_b , 接下来 accumulated reward 会有多大。但是因为其普适性不好, 其需要把所有的 state 都匹配到, 如果我们是做一个简单的贪吃蛇游戏等 state

有限的问题，还可以进行。但是如果我们做的是一个图片型的任务，我们几乎不可能将所有的 state (对应每一帧的图像) 的都”记录“下来。总之，其不能对于未出现过的 input state 进行对应的 value 的输出。

- 基于 MC 的 Network 方法：为了解决上面描述的 Monte-Carlo (MC) 方法的不足，我们将其中的 state value function $V^\pi(s)$ 定义为一个 Network，其可以对于从未出现过的 input state，根据 network 的泛化和拟合能力，也可以”估测“出一个 value output。

- 基于 Temporal-difference (时序差分) 的 Network 方法，即 TD based Network：与我们再前 4 章介绍的 MC 与 TD 的区别一样，这里两者区别也相同。在 MC based 的方法中，每次我们都要算 accumulated reward，也就是从某一个 state s_a 一直玩到游戏结束的时候，得到的所有 reward 的总和。所以要应用 MC based 方法时，我们必须至少把这个游戏玩到结束。但有些游戏非常的长，你要玩到游戏结束才能够 update network，花的时间太长了。因此我们会采用 TD based 的方法。TD based 的方法不需要把游戏玩到底，只要在游戏的某一个情况，某一个 state s_t 的时候，采取 action a_t 得到 reward r_t ，跳到 state s_{t+1} ，就可以应用 TD 的方法。公式与之前介绍的 TD 方法类似，即： $V^\pi(s_t) = V^\pi(s_{t+1}) + r_t$ 。

- 基于 MC 和基于 TD 的区别在于：MC 本身具有很大的随机性，我们可以将其 G_a 堪称一个 random 的变量，所以其最终的 variance 很大。而对于 TD，其具有随机性的变量为 r ，因为计算 s_t 我们采取同一个 action，你得到的 reward 也不一定是一样的，所以对于 TD 来说， r 是一个 random 变量。但是相对于 MC 的 G_a 的随机程度来说， r 的随机性非常小，这是因为本身 G_a 就是由很多的 r 组合而成的。但另一个角度来说，在 TD 中，我们的前提是 $r_t = V^\pi(s_{t+1}) - V^\pi(s_t)$ ，但是我们通常无法保证 $V^\pi(s_{t+1}) - V^\pi(s_t)$ 计算的误差为零。所以当 $V^\pi(s_{t+1}) - V^\pi(s_t)$ 计算的不准确的话，那应用上式得到的结果，其实也会是不准的。所以 MC 跟 TD 各有优劣。

- 目前，TD 的方法是比较常见的，MC 的方法其实是比较少用的。

- 基于我们上面说的 network (基于 MC) 的方法，我们怎么训练这个网络呢？或者我们应该将其看做 ML 中什么类型的问题呢？

答：理想状态，我们期望对于一个 input state 输出其无误差的 reward value。也就是说这个 value function 来说，如果 input 是 state s_a ，正确的 output 应该是 G_a 。如果 input state s_b ，正确的 output 应该是 value G_b 。所以在训练的时候，其就是一个典型的 ML 中的回归问题 (regression problem)。所以我们实际中需要输出的仅仅是一个非精确值，即你希望在 input s_a 的时候，output value 跟 G_a 越近越好，input s_b 的时候，output value 跟 G_b 越近越好。其训练方法，和我们在训练 CNN、DNN 时的方法类似，就不再一一赘述。

- 基于上面介绍的基于 TD 的 network 方法，具体地，我们应该怎么训练模型呢？

答：核心的函数为 $V^\pi(s_t) = V^\pi(s_{t+1}) + r_t$ 。我们将 state s_t 作为 input 输入 network 里，因为 s_t 丢到 network 里面会得到 output $V^\pi(s_t)$ ，同样将 s_{t+1} 作为 input 输入 network 里面会得到 $V^\pi(s_{t+1})$ 。同时核心函数： $V^\pi(s_t) = V^\pi(s_{t+1}) + r_t$ 告诉我们， $V^\pi(s_t)$ 减 $V^\pi(s_{t+1})$ 的值应该是 r_t 。然后希望它们两个相减的 loss 跟 r_t 尽可能地接近。这也就是我们这个 network 的优化目标或者说 loss function。

- state-action value function (Q-function) 和 state value function 的有什么区别和联系？

答：

- state value function 的 input 是一个 state，它是根据 state 去计算出，看到这个 state 以后的 expected accumulated reward 是多少。

- state-action value function 的 input 是一个 state 跟 action 的 pair，即在某一个 state 采取某一个 action，假设我们都使用 actor π ，得到的 accumulated reward 的期望值有多大。

- Q-function 的两种表示方法？

答：

- 当 input 是 state 和 action 的 pair 时，output 就是一个 scalar。

- 当 input 仅是一个 state 时， output 就是好几个 value。

- 当我们有了 Q-function 后，我们怎么找到更好的策略 π' 呢？或者说这个 π' 本质来说是什么？

答：首先， π' 是由 $\pi'(s) = \arg \max_a Q^\pi(s, a)$ 计算而得，其表示假设你已经 learn 出 π 的 Q-function，今天在某一个 state s，把所有可能的 action a 都一一带入这个 Q-function，看看说哪一个 a 可以让 Q-function 的 value 最大，那这一个 action，就是 π' 会采取的 action。所以根据上式决定的 actoin 的步骤一定比原来的 π 要好，即 $V^{\pi'}(s) \geq V^\pi(s)$ 。

- 解决探索-利用窘境 (Exploration-Exploitation dilemma) 问题的 Exploration 的方法有哪些？他们具体的方法是怎样的？

答：

1. Epsilon Greedy：我们有 $1 - \varepsilon$ 的机率，通常 ε 很小，完全按照 Q-function 来决定 action。但是有 ε 的机率是随机的。通常在实现上 ε 会随着时间递减。也就是在最开始的时候。因为还不知道那个 action 是比较好的，所以你会花比较大的力气在做 exploration。接下来随着 training 的次数越来越多。已经比较确定说哪一个 Q 是比较好的。你就会减少你的 exploration，你会把 ε 的值变小，主要根据 Q-function 来决定你的 action，比较少做 random，这是 Epsilon Greedy。
2. Boltzmann Exploration：这个方法就比较像是 policy gradient。在 policy gradient 里面 network 的 output 是一个 expected action space 上面的一个 probability distribution。再根据 probability distribution 去做 sample。所以也可以根据 Q value 去定一个 probability distribution，假设某一个 action 的 Q value 越大，代表它越好，我们采取这个 action 的机率就越高。这是 Boltzmann Exploration。

- 我们使用 Experience Replay (经验回放) 有什么好处？

答：

1. 首先，在强化学习的整个过程中，最花时间的 step 是在跟环境做互动，使用 GPU 乃至 TPU 加速来训练 network 相对来说是比较快的。而用 replay buffer 可以减少跟环境做互动的次数，因为在训练的时候，我们的 experience 不需要通通来自于某一个 policy (或者当前时刻的 policy)。一些过去的 policy 所得到的 experience 可以放在 buffer 里面被使用很多次，被反复的再利用，这样让你的 sample 到 experience 的利用是高效的。
2. 另外，在训练网络的时候，其实我们希望一个 batch 里面的 data 越 diverse 越好。如果你的 batch 里面的 data 都是同样性质的，我们的训练出的模型拟合能力不会很乐观。如果 batch 里面都是一样的 data，你 train 的时候，performance 会比较差。我们希望 batch data 越 diverse 越好。那如果 buffer 里面的那些 experience 通通来自于不同的 policy，那你 sample 到的一个 batch 里面的 data 会是比较 diverse。这样可以保证我们模型的性能至少不会很差。

- 在 Experience Replay 中我们是要观察 π 的 value，里面混杂了一些不是 π 的 experience，这会有影响吗？

答：没关系。这并不是因为过去的 π 跟现在的 π 很像，就算过去的 π 没有很像，其实也是没有关系的。主要的原因是我们并不是去 sample 一个 trajectory，我们只 sample 了一个 experience，所以跟是不是 off-policy 这件事是没有关系的。就算是 off-policy，就算是这些 experience 不是来自于 π ，我们其实还是可以拿这些 experience 来估测 $Q^\pi(s, a)$ 。

- DQN (Deep Q-learning) 和 Q-learning 有什么异同点？

答：整体来说，从名称就可以看出，两者的目标价值以及价值的 update 方式基本相同，另外一方面，不同点在于：

- 首先，DQN 将 Q-learning 与深度学习结合，用深度网络来近似动作价值函数，而 Q-learning 则是采用表格存储。
- DQN 采用了我们前面所描述的经验回放 (Experience Replay) 训练方法，从历史数据中随机采样，而 Q-learning 直接采用下一个状态的数据进行学习。

6.9 Something About Interview

- 高冷的面试官：请问 DQN (Deep Q-Network) 是什么？其两个关键性的技巧分别是什么？
答：Deep Q-Network 是基于深度学习的 Q-learning 算法，其结合了 Value Function Approximation (价值函数近似) 与神经网络技术，并采用了目标网络 (Target Network) 和经验回放 (Experience Replay) 的方法进行网络的训练。
- 高冷的面试官：接上题，DQN 中的两个 trick：目标网络和 experience replay 的具体作用是什么呢？
答：在 DQN 中某个动作值函数的更新依赖于其他动作值函数。如果我们一直更新值网络的参数，会导致更新目标不断变化，也就是我们在追逐一个不断变化的目标，这样势必会不太稳定。为了解决在基于 TD 的 Network 的问题时，优化目标 $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 左右两侧会同时变化使得训练过程不稳定，从而增大 regression 的难度。target network 选择将上式的右部分即 $r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ 固定，通过改变上式左部分的 network 的参数，进行 regression。对于经验回放，其会构建一个 Replay Buffer (Replay Memory)，用来保存许多 data，每一个 data 的形式如下：在某一个 state s_t ，采取某一个 action a_t ，得到了 reward r_t ，然后跳到 state s_{t+1} 。我们使用 π 去跟环境互动很多次，把收集到的数据都放到这个 replay buffer 中。当我们的 buffer “装满” 后，就会自动删去最早进入 buffer 的 data。在训练时，对于每一轮迭代都有相对应的 batch (与我们训练普通的 Network 一样通过 sample 得到)，然后用这个 batch 中的数据去 update 我们的 Q-function。也就是说，Q-function 再 sample 和训练的时候，会用到过去的经验数据，也可以消除样本之间的相关性。
- 高冷的面试官：DQN (Deep Q-learning) 和 Q-learning 有什么异同点？
答：整体来说，从名称就可以看出，两者的目标价值以及价值的 update 方式基本相同，另外一方面，不同点在于：
 - 首先，DQN 将 Q-learning 与深度学习结合，用深度网络来近似动作价值函数，而 Q-learning 则是采用表格存储。
 - DQN 采用了我们前面所描述的经验回放 (Experience Replay) 训练方法，从历史数据中随机采样，而 Q-learning 直接采用下一个状态的数据进行学习。
- 高冷的面试官：请问，随机性策略和确定性策略有什么区别吗？
答：随机策略表示为某个状态下动作取值的分布，确定性策略在每个状态只有一个确定的动作可以选。从熵的角度来说，确定性策略的熵为 0，没有任何随机性。随机策略有利于我们进行适度的探索，确定性策略的探索问题更为严峻。
- 高冷的面试官：请问不打破数据相关性，神经网络的训练效果为什么就不好？
答：在神经网络中通常使用随机梯度下降法。随机的意思是我们随机选择一些样本来增量式的估计梯度，比如常用的采用 batch 训练。如果样本是相关的，那就意味着前后两个 batch 的很可能也是相关的，那么估计的梯度也会呈现出某种相关性。如果不幸的情况下，后面的梯度估计可能会抵消掉前面的梯度量。从而使得训练难以收敛。

References

- Intro to Reinforcement Learning (强化学习纲要)
- 神经网络与深度学习
- 强化学习基础 David Silver 笔记

第 7 章 Tips of DQN

7.1 Double DQN

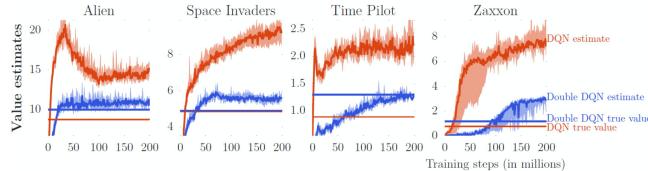


图 7.1

接下来要讲的是训练 DQN 的一些 tips。第一个 tip 是做 Double DQN。为什么要有 Double DQN 呢？因为在实现上，你会发现 Q 值往往是被高估的。上图来自于 Double DQN 的原始 paper，它想要显示的结果就是 Q 值往往是被高估的。

这边有 4 个不同的小游戏，横轴是训练的时间，红色锯齿状一直在变的线就是 Q-function 对不同的状态估计出来的平均 Q 值，有很多不同的状态，每个状态你都 sample 一下，然后算它们的 Q 值，把它们平均起来。

这条红色锯齿状的线在训练的过程中会改变，但它是不断上升的。因为 Q-function 是取决于你的策略的。学习的过程中你的策略越来越强，你得到的 Q 值会越来越大。在同一个状态，你得到 reward 的期望会越来越大，所以一般而言，这个值都是上升的，但这是 Q-network 估测出来的值。

接下来你真地去算它，怎么真地去算？你有策略，然后真的去玩那个游戏，就玩很多次，玩个一百万次。然后就去真地算说，在某一个状态，你会得到的 Q 值到底有多少。你会得到在某一个状态采取某一个动作。你接下来会得到累积奖励 (accumulated reward) 是多少。你会发现估测出来的值远比实际的值大，在每一个游戏都是这样，都大很多。所以今天要提出 Double DQN 的方法，它可以让估测的值跟实际的值是比较接近的。

我们先看它的结果，蓝色的锯齿状的线是 Double DQN 的 Q-network 所估测出来的 Q 值，蓝色的无锯齿状的线是真正的 Q 值，你会发现它们是比较接近的。用网络估测出来的就不用管它，比较没有参考价值。用 Double DQN 得出来真正的累积奖励，在这 3 种情况下都是比原来的 DQN 高的，代表 Double DQN 学习出来的那个策略比较强。所以它实际上得到的 reward 是比较大的。虽然一般的 DQN 的 Q-network 高估了自己会得到的 reward，但实际上它得到的 reward 是比较低的。

- Q value is usually over estimate

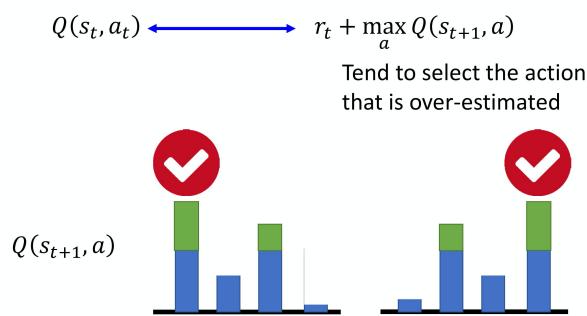


图 7.2

Q: 为什么 Q 值总是被高估了呢？

A: 因为实际上在做的时候，是要让左边这个式子跟右边这个目标越接近越好。你会发现目标的值很

容易一不小心就被设得太高。因为在算这个目标的时候，我们实际上在做的事情是，看哪一个 a 可以得到最大的 Q 值，就把它加上去，就变成我们的目标。所以假设有一个动作得到的值是被高估的。

举例来说，如图 7.2 所示，现在有 4 个动作，本来它们得到的值都是差不多的，它们得到的 reward 都是差不多的。但是在估计的时候，网络是有误差的。

假设是第一个动作被高估了，假设绿色的东西代表是被高估的量，它被高估了，那这个目标就会选这个动作，然后就会选这个高估的 Q 值来加上 r_t ，来当作你的目标。如果第四个动作被高估了，那就会选第四个动作来加上 r_t 来当作你的目标值。所以你总是会选那个 Q 值被高估的，你总是会选那个 reward 被高估的动作当作这个 \max 的结果去加上 r_t 当作你的目标，所以你的目标总是太大。

- Q value is usually over estimate

$$Q(s_t, a_t) \xrightarrow{\text{blue arrow}} r_t + \max_a Q(s_{t+1}, a)$$

- Double DQN: two functions Q and Q' Target Network

$$Q(s_t, a_t) \xrightarrow{\text{blue arrow}} r_t + Q'\left(s_{t+1}, \arg \max_a Q(s_{t+1}, a)\right)$$

If Q over-estimate a , so it is selected. Q' would give it proper value.

How about Q' overestimate? The action will not be selected by Q .

Hado V. Hasselt, "Double Q-learning", NIPS 2010
Hado van Hasselt, Arthur Guez, David Silver, "Deep Reinforcement Learning with Double Q-learning", AAAI 2016

图 7.3

Q: 怎么解决目标值总是太大的问题呢？

A: 在 Double DQN 里面，选动作的 Q -function 跟算值的 Q -function 不是同一个。在原来的 DQN 里面，你穷举所有的 a ，把每一个 a 都带进去，看哪一个 a 可以给你的 Q 值最高，那你就把那个 Q 值加上 r_t 。但是在 Double DQN 里面，你有两个 Q -network：

- 第一个 Q -network Q 决定哪一个动作的 Q 值最大（你把所有的 a 带入 Q 中，看看哪一个 Q 值最大）。
- 你决定你的动作以后，你的 Q 值是用 Q' 算出来的。

假设我们有两个 Q -function，

假设第一个 Q -function 高估了它现在选出来的动作 a ，只要第二个 Q -function Q' 没有高估这个动作 a 的值，那你算出来的还是正常的值。假设 Q' 高估了某一个动作的值，那也没差，因为只要前面这个 Q 不要选那个动作出来就没事了，这个就是 Double DQN 神奇的地方。

Q: 哪来 Q 跟 Q' 呢？哪来两个网络呢？

A: 在实现上，你有两个 Q -network：目标的 Q -network 和你会更新的 Q -network。所以在 Double DQN 里面，你会拿你会更新参数的那个 Q -network 去选动作，然后你拿目标网络（固定住不动的网络）去算值。

Double DQN 相较于原来的 DQN 的更改是最少的，它几乎没有增加任何的运算量，连新的网络都不用，因为原来就有两个网络了。你唯一要做的事情只有，本来你在找 Q 值最大的 a 的时候，你是用 Q' 来算，你是用目标网络来算，现在改成用另外一个是会更新的 Q -network 来算。

假如你今天只选一个 tip 的话，正常人都是实现 Double DQN，因为很容易实现。

7.2 Dueling DQN

第二个 tip 是 [Dueling DQN](#)。其实 Dueling DQN 也蛮好做的，相较于原来的 DQN，它唯一的差别是改了网络的架构。 Q -network 就是输入状态，输出就是每一个动作的 Q 值。Dueling DQN 唯一做的事情是改了网络的架构，其它的算法都不需要动。

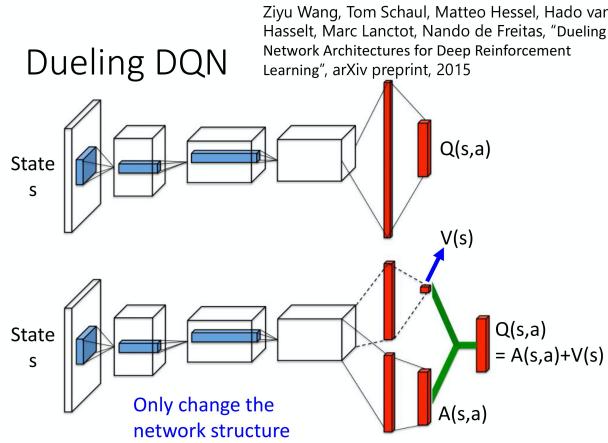


图 7.4

Q: Dueling DQN 是怎么改了网络的架构呢？

A: 本来的 DQN 就是直接输出 Q 值的值。现在这个 dueling 的 DQN，就是下面这个网络的架构。它不直接输出 Q 值的值，它分成两条路径去运算：

- 第一条路径会输出一个 scalar，这个 scalar 叫做 $V(s)$ 。因为它跟输入 s 是有关系，所以叫做 $V(s)$ ， $V(s)$ 是一个 scalar。
- 第二条路径会输出一个 vector，这个 vector 叫做 $A(s, a)$ 。下面这个 vector，它是每一个动作都有一个值。

你再把这两个东西加起来就可以得到你的 Q 值。

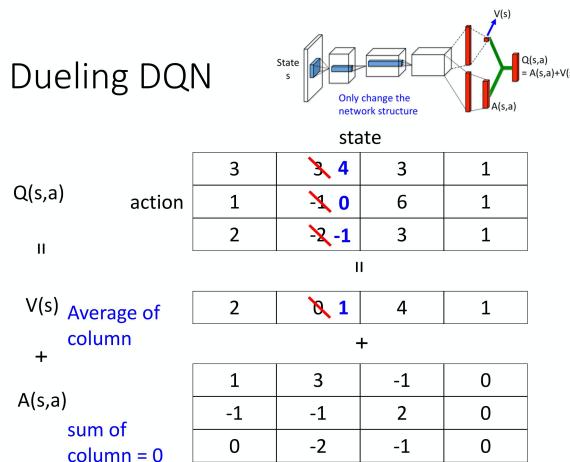


图 7.5

Q: 这么改有什么好处？

A : 那我们假设说，原来的 $Q(s, a)$ 就是一个表格。我们假设状态是离散的，实际上状态不是离散的。为了说明方便，我们假设就是只有 4 个不同的状态，只有 3 个不同的动作，所以 $Q(s, a)$ 你可以看作是一个表格。

我们知道：

$$Q(s, a) = V(s) + A(s, a)$$

其中

- $V(s)$ 是对不同的状态，它都有一个值。

- $A(s, a)$ 它是对不同的状态，不同的动作都有一个值。

你把这个 V 的值加到 A 的每一列就会得到 Q 的值。把 $2+1$, $2+(-1)$, $2+0$, 就得到 3 , 1 , 2 , 以此类推。

如图 7.5 所示，假设说你在训练网络的时候，目标是希望这一个值变成 4 ，这一个值变成 0 。但是你实际上能更改的并不是 Q 的值，你的网络更改的是 V 跟 A 的值。根据网络的参数， V 跟 A 的值输出以后，就直接把它们加起来，所以其实不是更动 Q 的值。

然后在学习网络的时候，假设你希望这边的值，这个 3 增加 1 变成 4 ，这个 -1 增加 1 变成 0 。最后你在训练网络的时候，网络可能会说，我们就不要动这个 A 的值，就动 V 的值，把 V 的值从 0 变成 1 。把 0 变成 1 有什么好处呢？你会发现说，本来你只想动这两个东西的值，那你会发现说，这个第三个值也动了， -2 变成 -1 。所以有可能说你在某一个状态，你明明只 sample 到这 2 个动作，你没 sample 到第三个动作，但是你其实也可以更改第三个动作的 Q 值。这样的好处就是你不需要把所有的 state-action pair 都 sample 过，你可以用比较高效的方式去估计 Q 值出来。因为有时候你更新的时候，不一定是更新下面这个表格。而是只更新了 $V(s)$ ，但更新 $V(s)$ 的时候，只要一改所有的值就会跟着改。这是一个比较有效率的方法，去使用你的数据，这个是 Dueling DQN 可以带给我们的好处。

那可是接下来有人就会问说会不会最后学习出来的结果是说，反正 machine 就学到 V 永远都是 0 ，然后反正 A 就等于 Q ，那你就没有得到任何 Dueling DQN 可以带给你的好处，就变成跟原来的 DQN 一模一样。为了避免这个问题，实际上你要给 A 一些约束，让更新 A 其实比较麻烦，让网络倾向于会想要去用 V 来解问题。

举例来说，你可以看原始的文献，它有不同的约束。一个最直觉的约束是你必须要让这个 A 的每一列的和都是 0 ，所以看我这边举的例子，列的和都是 0 。如果这边列的和都是 0 ，这边这个 V 的值，你就可以想成是上面 Q 的每一列的平均值。这个平均值，加上这些值才会变成是 Q 的值。所以今天假设你发现说你在更新参数的时候，你是要让整个行一起被更新。你就不会想要更新这边，因为你不会想要更新 A 这个矩阵。因为 A 这个矩阵的每一列的和都要是 0 ，所以你没有办法说，让这边的值，通通都 $+1$ ，这件事是做不到的。因为它的约束就是你的和永远都是要 0 。所以不可以都 $+1$ ，这时候就会强迫网络去更新 V 的值，然后让你可以用比较有效率的方法，去使用你的数据。

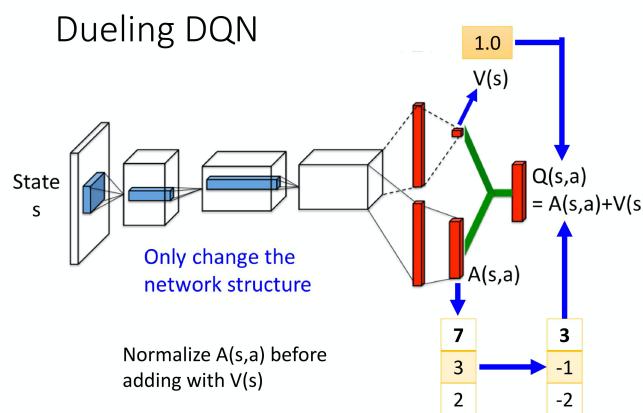


图 7.6

实现时，你要给这个 A 一个约束。举个例子，如图 7.6 所示，假设你有 3 个动作，然后在这边输出的 vector 是 $[7, 3, 2]^T$ ，你在把这个 A 跟这个 V 加起来之前，先加一个归一化 (normalization)，就好像做那个层归一化 (layer normalization) 一样。加一个归一化，这个归一化做的事情就是把 $7+3+2$ 加起来等于 12 , $12/3 = 4$ 。然后把这边通通减掉 4 ，变成 $3, -1, 2$ 。再把 $3, -1, 2$ 加上 1.0 ，得到最后的 Q 值。这个归一化的步骤就是网络的其中一部分，在训练的时候，你从这边也是一路 back propagate 回来的，只是归一化是没有参数的，它只是一个归一化的操作。把它可以放到网络里面，跟网络的其他部分 jointly trained,

这样 A 就会有比较大的约束。这样网络就会给它一些好处，倾向于去更新 V 的值，这个是 Dueling DQN。

7.3 Prioritized Experience Replay

<https://arxiv.org/abs/1511.05952?context=cs>

Prioritized Experience Replay

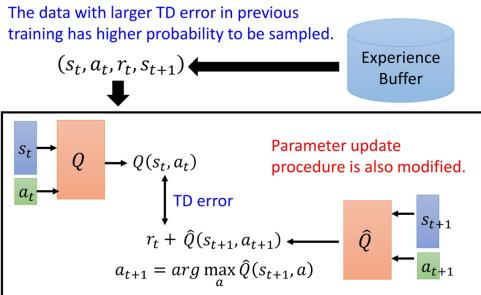


图 7.7

有一个技巧叫做 [Prioritized Experience Replay](#)。Prioritized Experience Replay 是什么意思呢？

如图 7.7 所示，我们原来在 sample 数据去训练你的 Q-network 的时候，你是均匀地从 experience buffer 里面去 sample 数据。那这样不见得是最好的，因为也许有一些数据比较重要。假设有一些数据，你之前有 sample 过。你发现这些数据的 TD error 特别大 (TD error 就是网络的输出跟目标之间的差距)，那这些数据代表说你在训练网络的时候，你是比较训练不好的。那既然比较训练不好，那你就应该给它比较大的概率被 sample 到，即给它 [priority](#)。这样在训练的时候才会多考虑那些训练不好的训练数据。实际上在做 prioritized experience replay 的时候，你不仅会更改 sampling 的 process，你还会因为更改了 sampling 的过程，更改更新参数的方法。所以 prioritized experience replay 不仅改变了 sample 数据的分布，还改变了训练过程。

7.4 Balance between MC and TD

Multi-step Balance between MC and TD

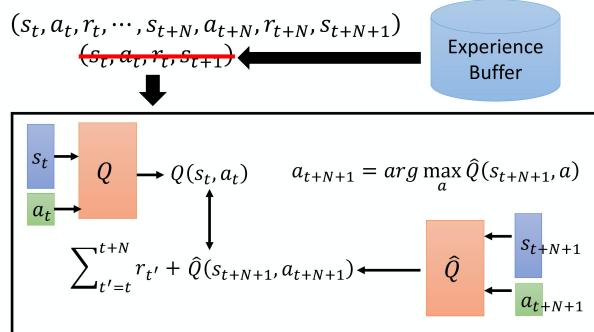


图 7.8

另外一个可以做的方法是 balance MC 跟 TD。MC 跟 TD 的方法各自有各自的优劣，怎么在 MC 跟 TD 里面取得一个平衡呢？我们的做法是这样，在 TD 里面，在某一个状态 s_t 采取某一个动作 a_t 得到

reward r_t , 接下来跳到那一个状态 s_{t+1} 。但是我们可以不要只存一个步骤的数据，我们存 N 个步骤的数据。

我们记录在 s_t 采取 a_t , 得到 r_t , 会跳到什么样 s_t 。一直纪录到在第 N 个步骤以后，在 s_{t+N} 采取 a_{t+N} 得到 reward r_{t+N} , 跳到 s_{t+N+1} 的这个经验，通通把它存下来。实际上你今天在做更新的时候，在做 Q-network learning 的时候，你的 learning 的方法会是这样，你 learning 的时候，要让 $Q(s_t, a_t)$ 跟你的目标值越接近越好。 \hat{Q} 所计算的不是 s_{t+1} , 而是 s_{t+N+1} 的。你会把 N 个步骤以后的状态丢进来，去计算 N 个步骤以后，你会得到的 reward。要算目标值的话，要再加上多步 (multi-step) 的 reward $\sum_{t'=t}^{t+N} r_{t'}$ ，多步的 reward 是从时间 t 一直到 t+N 的 N 个 reward 的和。然后希望你的 $Q(s_t, a_t)$ 和目标值越接近越好。

你会发现说这个方法就是 MC 跟 TD 的结合。因此它就有 MC 的好处跟坏处，也有 TD 的好处跟坏处。如果看它的这个好处的话，因为我们现在 sample 了比较多的步骤，之前是只 sample 了一个步骤，所以某一个步骤得到的数据是 real 的，接下来都是 Q 值估测出来的。现在 sample 比较多步骤，sample N 个步骤才估测值，所以估测的部分所造成的影响就会比小。当然它的坏处就跟 MC 的坏处一样，因为你的 r 比较多项，你把 N 项的 r 加起来，方差就会比较大。但是你可以去调这个 N 的值，去在方差跟不精确的 Q 之间取得一个平衡。N 就是一个 hyperparameter，你要调这个 N 到底是多少，你是要多 sample 三步，还是多 sample 五步。

7.5 Noisy Net

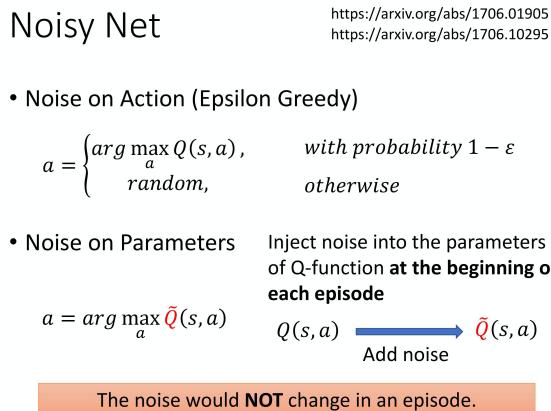


图 7.9

我们还可以改进探索。Epsilon Greedy 这样的探索是在动作的空间上面加噪声，但是有一个更好的方法叫做 [Noisy Net](#)，它是在参数的空间上面加噪声。

Noisy Net 的意思是说，每一次在一个 episode 开始的时候，在你要跟环境互动的时候，你就把你的 Q-function 拿出来，Q-function 里面其实就是一个网络，就变成你把那个网络拿出来，在网络的每一个参数上面加上一个高斯噪声 (Gaussian noise)，那你就把原来的 Q-function 变成 \tilde{Q} 。因为 \hat{Q} 已经用过， \hat{Q} 是那个目标网络，我们用 \tilde{Q} 来代表一个 [Noisy Q-function](#)。我们把每一个参数都加上一个高斯噪声，就得到一个新的网络叫做 \tilde{Q} 。

这边要注意在每个 episode 开始的时候，开始跟环境互动之前，我们就 sample 网络。接下来你就会用这个固定住的 noisy 网络去玩这个游戏，直到游戏结束，你才重新再去 sample 新的噪声。OpenAI 跟 DeepMind 又在同时间提出了一模一样的方法，通通都发表在 ICLR 2018，两篇 paper 的方法就是一样的。不一样的地方是，他们用不同的方法，去加噪声。OpenAI 加的方法好像比较简单，他就直接加一个高斯噪声就结束了，就把每一个参数，每一个 weight 都加一个高斯噪声就结束了。DeepMind 做比较复

杂，他们的噪声是由一组参数控制的，也就是说网络可以自己决定说它那个噪声要加多大，但是概念就是一样的。总之就是把你的 Q-function 的里面的那个网络加上一些噪声，把它变得有点不一样，跟原来的 Q-function 不一样，然后拿去跟环境做互动。两篇 paper 里面都有强调说，你这个参数虽然会加噪声，但在同一个 episode 里面你的参数就是固定的，你是在换 episode，玩第二场新的游戏的时候，你才会重新 sample 噪声，在同一场游戏里面就是同一个 noisy Q-network 在玩那一场游戏，这件事非常重要。为什么这件事非常重要呢？因为这是导致了 Noisy Net 跟原来的 Epsilon Greedy 或其它在动作做 sample 方法的本质上的差异。

Noisy Net

- Noise on Action
 - Given the same state, the agent may takes different actions.
 - No real policy works in this way 隨機亂試
- Noise on Parameters
 - Given the same (similar) state, the agent takes the same action.
 - → State-dependent Exploration
 - Explore in a *consistent* way 有系統地試

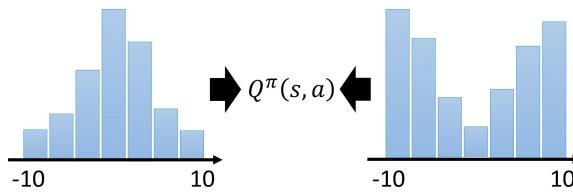
图 7.10

有什么样本质上的差异呢？在原来 sample 的方法，比如说 Epsilon Greedy 里面，就算是给同样的状态，你的 agent 采取的动作也不一定是一样的。因为你是用 sample 决定的，给定同一个状态，要根据 Q-function 的网络，你会得到一个动作，你 sample 到 random，你会采取另外一个动作。所以给定同样的状态，如果你今天是用 Epsilon Greedy 的方法，它得到的动作是不一样的。但实际上你的策略并不是这样运作的啊。在一个真实世界的策略，给同样的状态，他应该会有同样的回应。而不是给同样的状态，它其实有时候吃 Q-function，然后有时候又是随机的，所以这是一个不正常的动作，是在真实的情况下不会出现的动作。但是如果你是在 Q-function 上面去加噪声的话，就不会有这个情形。因为如果你今天在 Q-function 上加噪声，在 Q-function 的网络的参数上加噪声，那在整个互动的过程中，在同一个 episode 里面，它的网络的参数总是固定的，所以看到同样的状态，或是相似的状态，就会采取同样的动作，那这个是比较正常的。在 paper 里面有说，这个叫做 *state-dependent exploration*，也就是说你虽然会做探索这件事，但是你的探索是跟状态有关系的，看到同样的状态，你就会采取同样的探索的方式，而 noisy 的动作只是随机乱试。但如果你是在参数下加噪声，那在同一个 episode 里面，里面你的参数是固定的。那你就是有系统地在尝试，每次会试说，在某一个状态，我都向左试试看。然后再下一次在玩这个同样游戏的时候，看到同样的状态，你就说我再向右试试看，你是有系统地在探索这个环境。

7.6 Distributional Q-function

还有一个技巧叫做 *Distributional Q-function*。我们不讲它的细节，只告诉你大致的概念。*Distributional Q-function* 还蛮有道理的，但是它没有红起来。你就发现说没有太多人在实现的时候用这个技术，可能一个原因就是它不好实现。如图 7.11 所示，Q-function 是累积奖励的期望值，所以我们算出来的这个 Q 值其实是一个期望值。因为环境是有随机性的，在某一个状态采取某一个动作的时候，我们把所有的 reward 玩到游戏结束的时候所有的 reward 进行一个统计，你其实得到的是一个分布。也许在 reward 得到 0 的机率很高，在 -10 的概率比较低，在 +10 的概率比较低，但是它是一个分布。我们对这一个分布算它的平均值才是这个 Q 值，我们算出来是累积奖励的期望。所以累积奖励是一个分布，对它取期望，对它取平均

- State-action value function $Q^\pi(s, a)$
 - When using actor π , the cumulated reward expects to be obtained after seeing observation s and taking a



Different distributions can have the same values.

图 7.11

值，你得到了 Q 值。但不同的分布，它们其实可以有同样的平均值。也许真正的分布是右边的分布，它算出来的平均值跟左边的分布算出来的平均值其实是一样的，但它们背后所代表的分布其实是不一样的。假设我们只用 Q 值的期望来代表整个 reward 的话，其实可能会丢失一些信息，你没有办法 model reward 的分布。

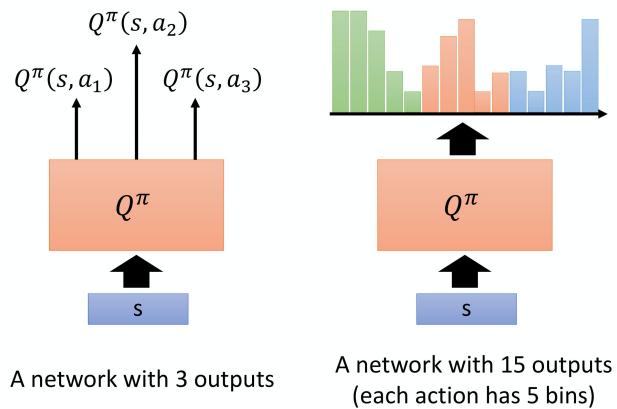


图 7.12

如图 7.12 所示，Distributional Q-function 它想要做的事情是对分布 (distribution) 建模，怎么做呢？在原来的 Q-function 里面，假设你只能够采取 a_1, a_2, a_3 3 个动作，那你就是输入一个状态，输出 3 个值。3 个值分别代表 3 个动作的 Q 值，但是这个 Q 值是一个分布的期望值。所以 Distributional Q-function 的想法就是何不直接输出那个分布。但是要直接输出一个分布也不知道怎么做。

实际上的做法是说，假设分布的值就分布在某一个 range 里面，比如说 -10 到 10，那把 -10 到 10 中间拆成一个一个的 bin，拆成一个一个的长条图。举例来说，在这个例子里面，每一个动作的 reward 的空间就拆成 5 个 bin。假设 reward 可以拆成 5 个 bin 的话，今天你的 Q-function 的输出是要预测说，你在某一个状态，采取某一个动作，你得到的 reward，落在某一个 bin 里面的概率。

所以其实这边的概率的和，这些绿色的 bar 的和应该是 1，它的高度代表说，在某一个状态采取某一个动作的时候，它落在某一个 bin 的机率。这边绿色的代表动作 1，红色的代表动作 2，蓝色的代表动作 3。所以今天你就可以真的用 Q-function 去估计 a_1 的分布， a_2 的分布， a_3 的分布。那实际上在做测试的时候，我们还是要选某一个动作去执行，那选哪一个动作呢？实际上在做的时候，还是选这个平均值最大的那个动作去执行。

但假设我们可以对 distribution 建模的话，除了选平均值最大的以外，也许在未来你可以有更多其他的运用。举例来说，你可以考虑它的分布长什么样子。如果分布方差很大，代表说采取这个动作虽然平均值可能平均而言很不错，但也许风险很高，你可以训练一个网络它是可以规避风险的。就在 2 个动作平均

值都差不多的情况下，也许可以选一个风险比较小的动作来执行，这是 Distributional Q-function 的好处。关于怎么训练这样的 Q-network 的细节，我们就不讲，你只要记得说 Q-network 有办法输出一个分布就对了。我们可以不只是估测得到的期望 reward 平均值的值。我们其实是可以估测一个分布的。

7.7 Rainbow

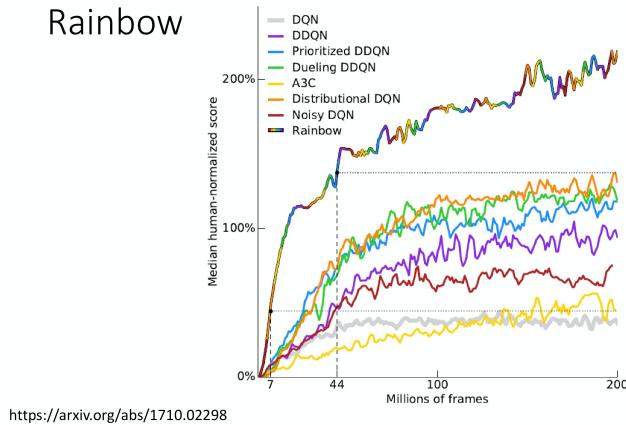


图 7.13

最后一个技巧叫做 rainbow，把刚才所有的方法都综合起来就变成 rainbow。因为刚才每一个方法，就是有一种自己的颜色，把所有的颜色通通都合起来，就变成 rainbow，它把原来的 DQN 也算是一种方法，故有 7 色。

那我们来看看这些不同的方法。横轴是训练过程，纵轴是玩了 10 几个 Atari 小游戏的平均的分数的和，但它取的是中位数的分数，为什么是取中位数不是直接取平均呢？因为它说每一个小游戏的分数，其实差很多。如果你取平均的话，到时候某几个游戏就控制了你的结果，所以它取中位数的值。

如图 7.13 所示，如果你是一般的 DQN，就灰色这一条线，就没有很强。那如果是你换 Noisy DQN，就强很多。如果这边每一个单一颜色的线是代表说只用某一个方法，那紫色这一条线是 DDQN(Double DQN)，DDQN 还蛮有效的。然后 Prioritized DDQN、Dueling DDQN 和 Distributional DQN 都蛮强的，它们都差不多很强的。A3C 其实是 Actor-Critic 的方法。单纯的 A3C 看起来是比 DQN 强的。这边怎么没有多步的方法，多步的方法就是平衡 TD 跟 MC，我猜是因为 A3C 本身内部就有做多步的方法，所以他可能觉得说有实现 A3C 就算是有实现多步的方法。所以可以把这个 A3C 的结果想成是多步方法的结果。其实这些方法他们本身之间是没有冲突的，所以全部都用上去就变成七彩的一个方法，就叫做 rainbow，然后它很高。

如图 7.14 所示，在 rainbow 这个方法里面，如果我们每次拿掉其中一个技术，到底差多少。因为现在是把所有的方法都加在一起，发现说进步很多，但会不会有些方法其实是没用的。所以看看说，每一个方法哪些方法特别有用，哪些方法特别没用。这边的虚线就是拿掉某一种方法以后的结果，你会发现说，黄色的虚线，拿掉多步掉很多。Rainbow 是彩色这一条，拿掉多步会掉下来。拿掉 Prioritized Experience Replay 后也马上就掉下来。拿掉分布，它也掉下来。

这边有一个有趣的地方是说，在开始的时候，分布的训练的方法跟其他方法速度差不多。但是如果你拿掉分布的时候，你的训练不会变慢，但是性能 (performance) 最后会收敛在比较差的地方。拿掉 Noisy Net 后性能也是差一点。拿掉 Dueling 也是差一点。拿掉 Double 没什么差，所以看来全部合在一起的时候，Double 是比较没有影响的。其实在 paper 里面有给一个 make sense 的解释，其实当你有用 Distributional DQN 的时候，本质上就不会高估你的 reward。我们是为了避免高估 reward 才加了 Double DQN。那在 paper 里面有讲说，如果有做 Distributional DQN，就比较不会有高估的结果。事实上他有真的算了一下

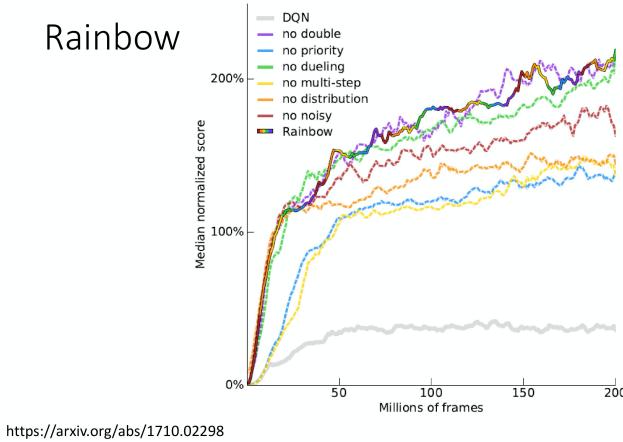


图 7.14

发现说，其实多数的状况是低估 reward 的，所以变成 Double DQN 没有用。

为什么做 Distributional DQN，不会高估 reward，反而会低估 reward 呢？因为这个 Distributional DQN 的输出的是一个分布的范围，输出的范围不可能是无限宽的，你一定是设一个范围，比如说最大输出范围就是从 -10 到 10。假设今天得到的 reward 超过 10 怎么办？是 100 怎么办，就当作没看到这件事。所以 reward 很极端的值，很大的值其实是会被丢掉的，所以用 Distributional DQN 的时候，你不会有高估的现象，反而会低估。

7.8 Keywords

- Double DQN：在 Double DQN 中存在有两个 Q-network，首先，第一个 Q-network，决定的是哪一个 action 的 Q value 最大，从而决定了你的 action。另一方面，Q value 是用 Q' 算出来的，这样就可以避免 over estimate 的问题。具体来说，假设我们有两个 Q-function，假设第一个 Q-function 它高估了它现在选出来的 action a ，那没关系，只要第二个 Q-function Q' 没有高估这个 action a 的值，那你算出来的，就还是正常的值。
- Dueling DQN：将原来的 DQN 的计算过程分为两个 path。对于第一个 path，会计算一个于 input state 有关的一个标量 $V(s)$ ；对于第二个 path，会计算出一个 vector $A(s, a)$ ，其对应每一个 action。最后的网络是将两个 path 的结果相加，得到我们最终需要的 Q value。用一个公式表示也就是 $Q(s, a) = V(s) + A(s, a)$ 。
- Prioritized Experience Replay（优先经验回放）：这个方法是为了解决我们在 chapter6 中提出的 Experience Replay（经验回放）方法不足进一步优化提出的。我们在使用 Experience Replay 时是 uniformly 取出的 experience buffer 中的 sample data，这里并没有考虑数据间的权重大小。例如，我们应该将那些 train 的效果不好的 data 对应的权重加大，即其应该有更大的概率被 sample 到。综上，prioritized experience replay 不仅改变了 sample data 的 distribution，还改变了 training process。
- Noisy Net：其在每一个 episode 开始的时候，即要和环境互动的时候，将原来的 Q-function 的每一个参数上面加上一个 Gaussian noise。那你就把原来的 Q-function 变成 \tilde{Q} ，即 Noisy Q-function。同样的我们把每一个 network 的权重等参数都加上一个 Gaussian noise，就得到一个新的 network \tilde{Q} 。我们会使用这个新的 network 从与环境互动开始到互动结束。
- Distributional Q-function：对于 DQN 进行 model distribution。将最终的网络的 output 的每一类别的 action 再进行 distribution。
- Rainbow：也就是将我们这两节内容所有的七个 tips 综合起来的方法，7 个方法分别包括：DQN、DDQN、Prioritized DDQN、Dueling DDQN、A3C、Distributional DQN、Noisy DQN，进而考察

每一个方法的贡献度或者是否对于与环境的交互式正反馈的。

7.9 Questions

- 为什么传统的 DQN 的效果并不好？参考公式 $Q(s_t, a_t) = r_t + \max_a Q(s_{t+1}, a)$

答：因为实际上在做的时候，是要让左边这个式子跟右边这个 target 越接近越好。比较容易可以发现 target 的值很容易一不小心就被设得太高。因为在算这个 target 的时候，我们实际上在做的事情是看哪一个 a 可以得到最大的 Q value，就把它加上去，就变成我们的 target。

举例来说，现在有 4 个 actions，本来其实它们得到的值都是差不多的，它们得到的 reward 都是差不多的。但是在 estimate 的时候，那毕竟是个 network。所以 estimate 的时候是有误差的。所以假设今天是第一个 action 它被高估了，假设绿色的东西代表是被高估的量，它被高估了，那这个 target 就会选这个 action。然后就会选这个高估的 Q value 来加上 r_t ，来当作你的 target。如果第 4 个 action 被高估了，那就会选第 4 个 action 来加上 r_t 来当作你的 target value。所以你总是会选那个 Q value 被高估的，你总是会选那个 reward 被高估的 action 当作这个 max 的结果去加上 r_t 当作你的 target。所以你的 target 总是太大。

- 接着上个思考题，我们应该怎么解决 target 总是太大的问题呢？

答：我们可以使用 Double DQN 解决这个问题。首先，在 Double DQN 里面，选 action 的 Q-function 跟算 value 的 Q-function 不同。在原来的 DQN 里面，你穷举所有的 a，把每一个 a 都带进去，看哪一个 a 可以给你的 Q value 最高，那你就把那个 Q value 加上 r_t 。但是在 Double DQN 里面，你有两个 Q-network，第一个 Q-network，决定哪一个 action 的 Q value 最大，你用第一个 Q-network 去带入所有的 a，去看看哪一个 Q value 最大。然后你决定你的 action 以后，你的 Q value 是用 Q' 算出来的，这样子有什么好处呢？为什么这样就可以避免 over estimate 的问题呢？因为今天假设我们有两个 Q-function，假设第一个 Q-function 它高估了它现在选出来的 action a，那没关系，只要第二个 Q-function Q' 没有高估这个 action a 的值，那你算出来的，就还是正常的值。假设反过来是 Q' 高估了某一个 action 的值，那也没差，因为反正只要前面这个 Q 不要选那个 action 出来就没事了。

- 哪来 Q 跟 Q' 呢？哪来两个 network 呢？

答：在实现上，你有两个 Q-network，一个是 target 的 Q-network，一个是真正你会 update 的 Q-network。所以在 Double DQN 里面，你的实现方法会是拿你会 update 参数的那个 Q-network 去选 action，然后你拿 target 的 network，那个固定住不动的 network 去算 value。而 Double DQN 相较于原来的 DQN 的更改是最少的，它几乎没有增加任何的运算量，连新的 network 都不用，因为你原来就有两个 network 了。你唯一要做的事情只有，本来你在找最大的 a 的时候，你在决定这个 a 要放哪一个的时候，你是用 Q' 来算，你是用 target network 来算，现在改成用另外一个会 update 的 Q-network 来算。

- 如何理解 Dueling DQN 的模型变化带来的好处？

答：对于我们的 $Q(s, a)$ 其对应的 state 由于为 table 的形式，所以是离散的，而实际中的 state 不是离散的。对于 $Q(s, a)$ 的计算公式， $Q(s, a) = V(s) + A(s, a)$ 。其中的 $V(s)$ 是对于不同的 state 都有值，对于 $A(s, a)$ 对于不同的 state 都有不同的 action 对应的值。所以本质上来说，我们最终的矩阵 $Q(s, a)$ 的结果是将每一个 $V(s)$ 加到矩阵 $A(s, a)$ 中得到的。从模型的角度考虑，我们的 network 直接改变的 $Q(s, a)$ 而是更改的 V 和 A 。但是有时我们 update 时不一定会将 $V(s)$ 和 $Q(s, a)$ 都更新。我们将其分成两个 path 后，我们就不需要将所有的 state-action pair 都 sample 一遍，我们可以使用更高效的 estimate Q value 方法将最终的 $Q(s, a)$ 计算出来。

- 使用 MC 和 TD 平衡方法的优劣分别有哪些？

答：

- 优势：因为我们现在 sample 了比较多的 step，之前是只 sample 了一个 step，所以某一个 step 得

到的 data 是真实值，接下来都是 Q value 估测出来的。现在 sample 比较多 step，sample N 个 step 才估测 value，所以估测的部分所造成的影响就会比小。

- 劣势：因为我们的 reward 比较多，当我们把 N 步的 reward 加起来，对应的 variance 就会比较大。但是我们可以选择通过调整 N 值，去在 variance 跟不精确的 Q 之间取得一个平衡。这里介绍的参数 N 就是一个 hyper parameter，你要调这个 N 到底是多少，你是要多 sample 三步，还是多 sample 五步。

7.10 Something About Interview

- 高冷的面试官：DQN 都有哪些变种？引入状态奖励的是哪种？

答：DQN 三个经典的变种：Double DQN、Dueling DQN、Prioritized Replay Buffer。

- Double-DQN：将动作选择和价值估计分开，避免价值过高估计。

- Dueling-DQN：将 Q 值分解为状态价值和优势函数，得到更多有用信息。

- Prioritized Replay Buffer：将经验池中的经验按照优先级进行采样。

- 简述 double DQN 原理？

答：DQN 由于总是选择当前值函数最大的动作值函数来更新当前的动作值函数，因此存在着过估计问题（估计的值函数大于真实的值函数）。为了解耦这两个过程，double DQN 使用了两个值网络，一个网络用来执行动作选择，然后用另一个值函数对一个的动作值更新当前网络。

- 高冷的面试官：请问 Dueling DQN 模型有什么优势呢？

答：对于我们的 $Q(s, a)$ 其对应的 state 由于为 table 的形式，所以是离散的，而实际中的 state 不是离散的。对于 $Q(s, a)$ 的计算公式， $Q(s, a) = V(s) + A(s, a)$ 。其中的 $V(s)$ 是对于不同的 state 都有值，对于 $A(s, a)$ 对于不同的 state 都有不同的 action 对应的值。所以本质上来说，我们最终的矩阵 $Q(s, a)$ 的结果是将每一个 $V(s)$ 加到矩阵 $A(s, a)$ 中得到的。从模型的角度考虑，我们的 network 直接改变的 $Q(s, a)$ 而是更改的 V 、 A 。但是有时我们 update 时不一定会将 $V(s)$ 和 $Q(s, a)$ 都更新。我们将其分成两个 path 后，我们就需要将所有的 state-action pair 都 sample 一遍，我们可以使用更高效的 estimate Q value 方法将最终的 $Q(s, a)$ 计算出来。

7.11 Solve Cartpole with DQN

推荐使用 Double-DQN 去解决，即建立两个初始参数相同的全连接网络 target_net 和 policy_net。

7.11.1 CartPole-v0

CartPole-v0 是 OpenAI gym 中的一个经典环境，通过向左 (action=0) 或向右 (action=1) 推车能够实现平衡，所以动作空间由两个动作组成。每进行一个 step 就会有一个 +1 的 reward，如果无法保持平衡那么 done 等于 true，本次 episode 失败。

理想状态下，每个 episode 至少能进行 200 个 step，也就是说每个 episode 的 reward 总和至少为 200，step 数目至少为 200。

环境建立如下：

```
env = gym.make('CartPole-v0')
env.seed(1) # 设置env随机种子
n_states = env.observation_space.shape[0] # 获取总的状态数
n_actions = env.action_space.n # 获取总的动作数
```

7.11.2 强化学习基本接口

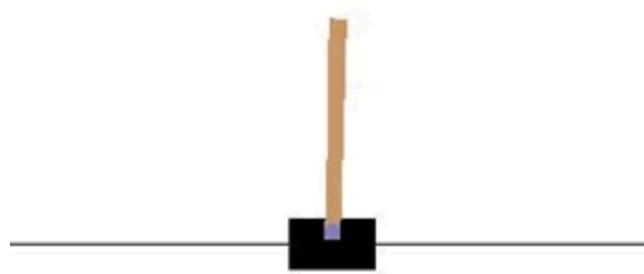


图 7.15

```

rewards = [] # 记录总的rewards
moving_average_rewards = [] # 记录总的经滑动平均处理后的rewards
ep_steps = []
for i_episode in range(1, cfg.max_episodes+1): # cfg.max_episodes为最大训练的episode数
    state = env.reset() # reset环境状态
    ep_reward = 0
    for i_step in range(1, cfg.max_steps+1): # cfg.max_steps为每个episode的补偿
        action = agent.select_action(state) # 根据当前环境state选择action
        next_state, reward, done, _ = env.step(action) # 更新环境参数
        ep_reward += reward
        agent.memory.push(state, action, reward, next_state, done) # 将state等这些transition存入memory
        state = next_state # 跳转到下一个状态
        agent.update() # 每步更新网络
        if done:
            break
    # 更新target network, 复制DQN中的所有weights and biases
    if i_episode % cfg.target_update == 0: # cfg.target_update为target_net的更新频率
        agent.target_net.load_state_dict(agent.policy_net.state_dict())
    print('Episode:', i_episode, ' Reward: %i' %
          int(ep_reward), 'n_steps:', i_step, 'done:', done, ' Explore: %.2f' % agent.epsilon)
    ep_steps.append(i_step)
    rewards.append(ep_reward)
    # 计算滑动窗口的reward
    if i_episode == 1:
        moving_average_rewards.append(ep_reward)
    else:
        moving_average_rewards.append(
            0.9*moving_average_rewards[-1]+0.1*ep_reward)

```

7.11.3 CartPole-v0

训练并绘制 reward 以及滑动平均后的 reward 随 episode 的变化曲线图并记录超参数写成报告，如图 7.16 所示。

同时也可以绘制测试 (eval) 模型时的曲线，如图 7.17 所示。

也可以tensorboard查看结果，如图 7.18 所示。

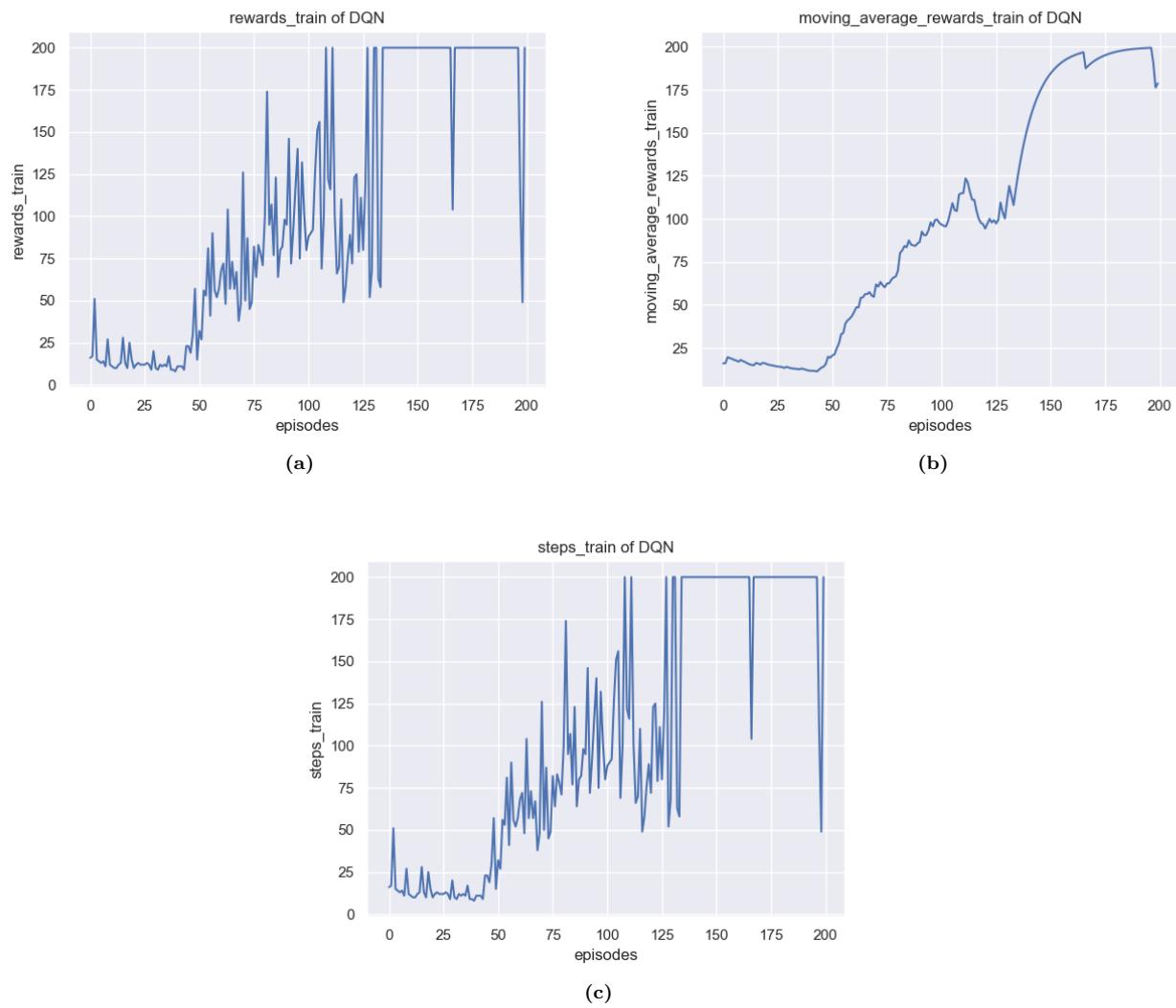


图 7.16

7.11.4 代码清单

main.py: 保存强化学习基本接口，以及相应的超参数，可使用 argparse
model.py: 保存神经网络，比如全链接网络
dqn.py: 保存算法模型，主要包含 select_action 和 update 两个函数
memory.py: 保存 Replay Buffer
plot.py: 保存相关绘制函数，可选
参考代码

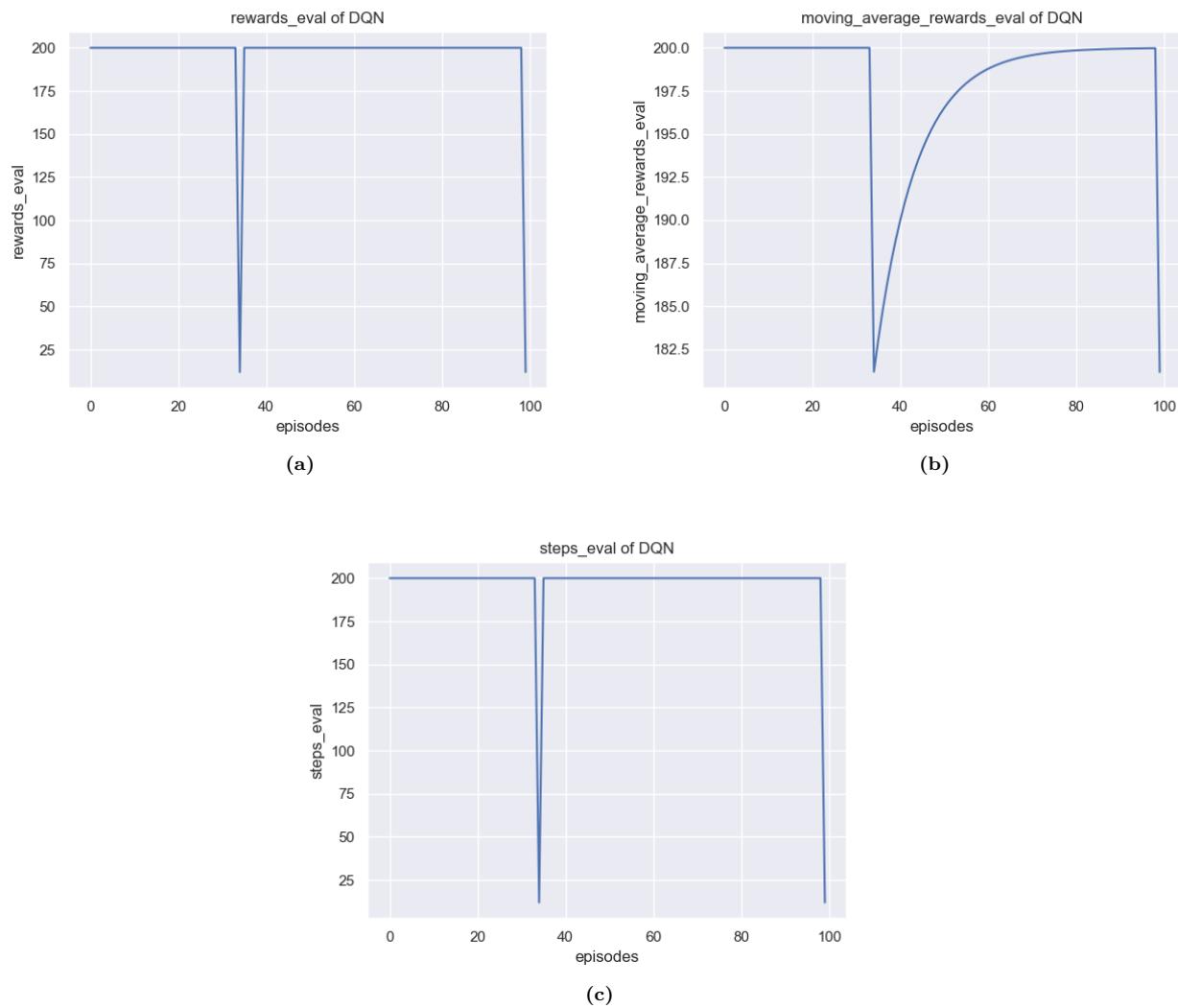


图 7.17

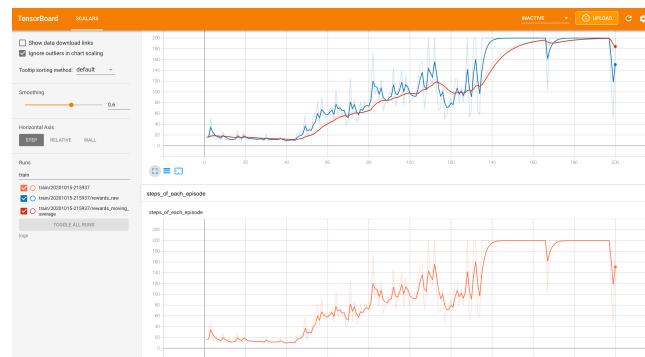


图 7.18

第 8 章 Q-learning for Continuous Actions

8.1 Solution 1 & Solution 2

Continuous Actions

- Action a is a *continuous vector*

$$a = \arg \max_a Q(s, a)$$

Solution 1

Sample a set of actions: $\{a_1, a_2, \dots, a_N\}$

See which action can obtain the largest Q value

Solution 2

Using gradient ascent to solve the optimization problem.

图 8.1

继续讲一下 Q-learning，其实跟 policy gradient based 方法比起来，Q-learning 是比较稳的。policy gradient 是没有太多游戏是玩得起来的，policy gradient 比较不稳，尤其在没有 PPO 之前，你很难用 policy gradient 做什么事情。Q-learning 相对而言是比较稳的。最早 DeepMind 的 paper 拿 deep reinforcement learning 来玩 Atari 的游戏，用的就是 Q-learning。Q-learning 比较容易 train 的一个理由是：在 Q-learning 里面，你只要能够 estimate 出 Q-function，就保证你一定可以找到一个比较好的 policy。也就是你只要能够 estimate 出 Q-function，就保证你可以 improve 你的 policy。而 estimate Q-function 这件事情，是比较容易的，因为它就是一个 regression problem。在这个 regression problem 里面，你可以轻易地知道 model learn 得是不是越来越好，只要看那个 regression 的 loss 有没有下降，你就知道说你的 model learn 得好不好，所以 estimate Q-function 相较于 learn 一个 policy 是比较容易的。你只要 estimate Q-function，就可以保证说现在一定会得到比较好的 policy。所以一般而言 Q-learning 比较容易操作。

Q: Q-learning 有什么问题呢？

A: 最大的问题是它不太容易处理 continuous action。很多时候 action 是 continuous 的。什么时候你的 action 会是 continuous 的呢？我们玩 Atari 的游戏，你的 agent 只需要决定比如说上下左右，这种 action 是 discrete 的。那很多时候你的 action 是 continuous 的。举例来说假设你的 agent 要做的事情是开自驾车，它要决定说它方向盘要左转几度，右转几度，这是 continuous 的。假设 agent 是一个机器人，它身上有 50 个关节，它的每一个 action 就对应到它身上的这 50 个关节的角度。而那些角度也是 continuous 的。所以很多时候 action 并不是一个 discrete 的东西，它是一个 vector。在这个 vector 里面，它的每一个 dimension 都有一个对应的 value，都是 real number，它是 continuous 的。假设 action 是 continuous 的，做 Q-learning 就会有困难。因为在做 Q-learning 里面一个很重要的一步是你要能够解这个 optimization problem。你 estimate 出 Q-function $Q(s, a)$ 以后，必须要找到一个 a ，它可以让 $Q(s, a)$ 最大。假设 a 是 discrete 的，那 a 的可能性都是有限的。举例来说，Atari 的小游戏里面， a 就是上下左右跟开火，它是有限的，你可以把每一个可能的 action 都带到 Q 里面算它的 Q value。但假如 a 是 continuous 的，你无法穷举所有可能的 continuous action，试试看哪一个 continuous action 可以让 Q 的 value 最大。

所以怎么办呢？在概念上，我们就是要能够解这个问题。怎么解这个问题呢？就有各种不同的 solution。

第一个 solution 是假设你不知道怎么解这个问题，因为 a 是没有办法穷举的。怎么办？用 sample 的。Sample 出 N 个可能的 a ，一个一个带到 Q-function 里面，看谁最快。这个方法其实也不会太不 efficient，因为你真的在运算的时候，你会用 GPU，一次会把 N 个 continuous action 都丢到 Q-function 里面，一次得到 N 个 Q value，然后看谁最大。当然这不是一个非常精确的做法，因为你没有办法做太多的 sample，所以你 estimate 出来的 Q value，你最后决定的 action 可能不是非常的精确，这是第一个 solution。

第二个 solution 是什么呢？既然要解的是一个 optimization problem，其实是要 maximize objective function，要 maximize 一个东西，就可以用 gradient ascent。你就把 a 当作是 parameter，然后你要找一组 a 去 maximize 你的 Q-function，你就用 gradient ascent 去 update a 的 value，最后看看你能不能找到一个 a 去 maximize 你的 Q-function，也就是你的 objective function。当然这样子你会遇到 global maximum 的问题，就不见得能够真的找到 optimal 的结果，而且这个运算量显然很大，因为你要迭代地 update a 。我们 train 一个 network 就很花时间了。如果你用 gradient ascent 的方法来处理 continuous 的 problem，等于是你每次要决定 take 哪一个 action 的时候，你都还要做一次 train network 的 process，显然运算量是很大的。这是第二个 solution。

8.2 Solution 3: Design a network

Continuous Actions

Solution 3 Design a network to make the optimization easy.

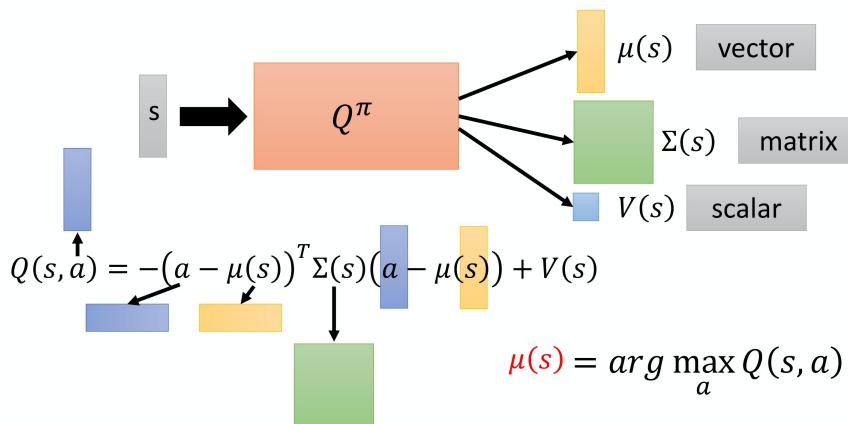


图 8.2

第三个 solution 是特别 design 一个 network 的架构，特别 design 你的 Q-function，使得解 $\arg \max$ 的 problem 变得非常容易。也就是这边的 Q-function 不是一个 general 的 Q-function，特别设计一下它的样子，让你要找让这个 Q-function 最大的 a 的时候非常容易。

上图是一个例子，这边有我们的 Q-function，这个 Q-function 的做法是这样。

- Input state s ，通常它就是一个 image，可以用一个向量或一个 matrix 来表示。
- Input 这个 s ，Q-function 会 output 3 个东西。它会 output $\mu(s)$ ，这是一个 vector。它会 output $\Sigma(s)$ ，这是一个 matrix。它会 output $V(s)$ ，这是一个 scalar。
- output 这 3 个东西以后，我们知道 Q-function 其实是吃一个 s 跟 a ，然后决定一个 value。Q-function 意思是说在某一个 state，take 某一个 action 的时候，你 expected 的 reward 有多大。到目前为止

这个 Q-function 只吃 s , 它还没有吃 a 进来, a 在哪里呢? 当这个 Q-function 吐出 μ 、 Σ 跟 V 的时候, 我们才把 a 引入, 用 a 跟 $\mu(s)$ $\Sigma(s)$ V 互相作用一下, 你才算出最终的 Q value。

- a 怎么和这 3 个东西互相作用呢? 实际上 $Q(s, a)$, 你的 Q-function 的运作方式是先 input s , 让你得到 μ, Σ 跟 V 。然后再 input a , 然后接下来把 a 跟 μ 相减。注意一下 a 现在是 continuous 的 action, 所以它也是一个 vector。假设你现在是要操作机器人的话, 这个 vector 的每一个 dimension, 可能就对应到机器人的某一个关节, 它的数值就是关节的角度, 所以 a 是一个 vector。把 a 的这个 vector 减掉 μ 的这个 vector, 取 transpose, 所以它是一个横的 vector。 Σ 是一个 matrix。然后 a 减掉 $\mu(s)$, a 和 $\mu(s)$ 都是 vector, 减掉以后还是一个竖的 vector。所以 $-(a - \mu(s))^T \Sigma(s)(a - \mu(s)) + V(s)$ 是一个 scalar, 这个数值就是 Q value $Q(s, a)$ 。
- 假设 $Q(s, a)$ 定义成这个样子, 我们要怎么找到一个 a 去 maximize 这个 Q value 呢? 这个 solution 非常简单, 什么样的 a , 可以让这一个 Q-function 最终的值最大呢? 因为 $(a - \mu(s))^T \Sigma(s)(a - \mu(s))$ 一定是正的, 它前面乘上一个负号, 所以第一项就假设我们不看这个负号的话, 第一项的值越小, 最终的 Q value 就越大。因为我们是把 $V(s)$ 减掉第一项, 所以第一项的值越小, 最后的 Q value 就越大。怎么让第一项的值最小呢? 你直接把 a 代入 μ 的值, 让它变成 0, 就会让第一项的值最小。
- Σ 一定是正定的。因为这个东西就像是 Gaussian distribution, 所以 μ 就是 Gaussian 的 mean, Σ 就是 Gaussian 的 variance。但 variance 是一个 positive definite 的 matrix, 怎么样让这个 Σ 一定是 positive definite 的 matrix 呢? 其实在 Q^π 里面, 它不是直接 output Σ , 如果直接 output 一个 Σ , 它不一定是 positive definite 的 matrix。它其实是 output 一个 matrix, 然后再把那个 matrix 跟另外一个 matrix 做 transpose 相乘, 然后可以确保 Σ 是 positive definite 的。这边要强调的点就是说, 实际上它不是直接 output 一个 matrix。你再去那个 paper 里面 check 一下它的 trick, 它可以保证说 Σ 是 positive definite 的。
- 你把 a 代入 $\mu(s)$ 以后, 你可以让 Q 的值最大。所以假设要你 arg max 这个东西, 虽然一般而言, 若 Q 是一个 general function, 你很难算, 但是我们这边 design 了 Q 这个 function, a 只要设 $\mu(s)$, 我们就得到最大值。你在解这个 arg max 的 problem 的时候就变得非常容易。所以 Q-learning 也可以用在 continuous 的 case, 只是有一些局限, 就是 function 不能够随便乱设, 它必须有一些限制。

8.3 Solution 4: Don't use Q-learning

第 4 招就是不要用 Q-learning。用 Q-learning 处理 continuous action 还是比较麻烦。

我们讲了 policy-based 的方法 PPO 和 value-based 的方法 Q-learning, 这两者其实是可以结合在一起的, 也就是 Actor-Critic 的方法。

8.4 Questions

- Q-learning 相比于 policy gradient based 方法为什么训练起来效果更好, 更平稳?
- 答: 在 Q-learning 中, 只要能够 estimate 出 Q-function, 就可以保证找到一个比较好的 policy, 同样的只要能够 estimate 出 Q-function, 就保证可以 improve 对应的 policy。而因为 estimate Q-function 作为一个回归问题, 是比较容易的。在这个回归问题中, 我们可以时刻观察我们的模型训练的效果是不是越来越好, 一般情况下我们只需要关注 regression 的 loss 有没有下降, 你就知道你的 model learn 的好不好。所以 estimate Q-function 相较于 learn 一个 policy 是比较容易的。你只要 estimate Q-function, 就可以保证说现在一定会得到比较好的 policy, 同样其也比较容易操作。
- Q-learning 在处理 continuous action 时存在什么样的问题呢?

答: 在日常的问题中, 我们的问题都是 continuous action 的, 例如我们的 agent 要做的事情是开自动驾驶, 它要决定说它方向盘要左转几度, 右转几度, 这就是 continuous 的; 假设我们的 agent 是一个机器人, 假设它身上有 50 个关节, 它的每一个 action 就对应到它身上的这 50 个关节的角度, 而那些角度也是 continuous 的。

Continuous Actions

Solution 4 Don't use Q-learning

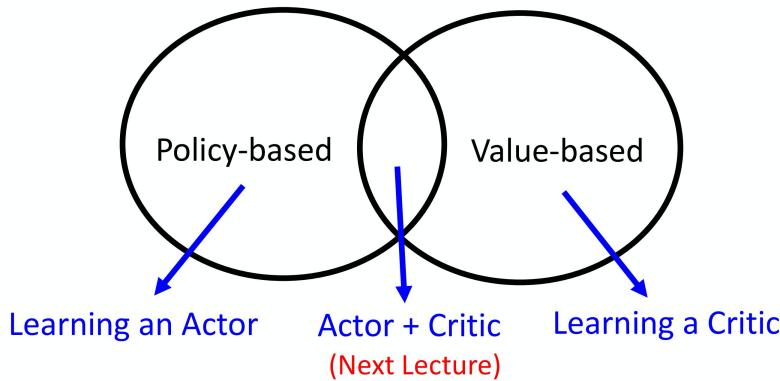


图 8.3

然而在解决 Q-learning 问题时，很重要的一步是要求能够解对应的优化问题。当我们 estimate 出 Q-function $Q(s, a)$ 以后，必须要找到一个 action，它可以让 $Q(s, a)$ 最大。假设 action 是 discrete 的，那 a 的可能性都是有限的。但如果 action 是 continuous 的情况下，我们就不能像离散的 action 一样，穷举所有可能的 continuous action 了。

为了解决这个问题，有以下几种 solutions：

- 第一个解决方法：我们可以使用所谓的 sample 方法，即随机 sample 出 N 个可能的 action，然后一个一个带到我们的 Q-function 中，计算对应的 N 个 Q value 比较哪一个的值最大。但是这个方法因为是 sample 所以不会非常的精确。
- 第二个解决方法：我们将这个 continuous action 问题，看为一个优化问题，从而自然而然地想到了可以用 gradient ascend 去最大化我们的目标函数。具体地，我们将 action 看为我们的变量，使用 gradient ascend 方法去 update action 对应的 Q-value。但是这个方法通常的时间花销比较大，因为是需要迭代运算的。
- 第三个解决方法：设计一个特别的 network 架构，设计一个特别的 Q-function，使得解我们 argmax Q-value 的问题变得非常容易。也就是这边的 Q-function 不是一个 general 的 Q-function，特别设计一下它的样子，让你要找让这个 Q-function 最大的 a 的时候非常容易。但是这个方法的 function 不能随意乱设，其必须有一些额外的限制。具体的设计方法，可以我们的 chapter8 的详细教程。
- 第四个解决方法：不用 Q-learning，毕竟用其处理 continuous 的 action 比较麻烦。

第 9 章 Actor-Critic

9.1 Actor-Critic

在 REINFORCE 算法中，每次需要根据一个策略采集一条完整的轨迹，并计算这条轨迹上的回报。这种采样方式的方差比较大，学习效率也比较低。我们可以借鉴时序差分学习的思想，使用动态规划方法来提高采样的效率，即从状态 s 开始的总回报可以通过当前动作的即时奖励 $r(s, a, s')$ 和下一个状态 s' 的值函数来近似估计。

演员-评论家算法 (Actor-Critic Algorithm) 是一种结合策略梯度和时序差分学习的强化学习方法，其中：

- 演员 (Actor) 是指策略函数 $\pi_\theta(a|s)$ ，即学习一个策略来得到尽量高的回报。
 - 评论家 (Critic) 是指值函数 $V^\pi(s)$ ，对当前策略的值函数进行估计，即评估演员的好坏。
 - 借助于值函数，演员-评论家算法可以进行单步更新参数，不需要等到回合结束才进行更新。
- 在 Actor-Critic 算法里面，最知名的方法就是 A3C(Asynchronous Advantage Actor-Critic)。
- 如果去掉 Asynchronous，只有 Advantage Actor-Critic，就叫做 A2C。
 - 如果加了 Asynchronous，变成 Asynchronous Advantage Actor-Critic，就变成 A3C。

9.1.1 Review: Policy Gradient

Review – Policy Gradient

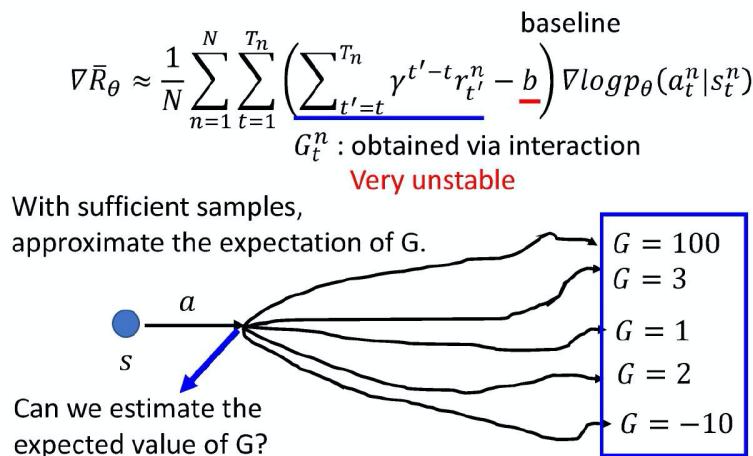


图 9.1

那我们复习一下 policy gradient，在 policy gradient，我们在更新 policy 的参数 θ 的时候，我们是用了下面这个式子来算出 gradient。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} \left(\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b \right) \nabla \log p_\theta(a_t^n | s_t^n)$$

这个式子是在说，我们先让 agent 去跟环境互动一下，那我们可以计算出在某一个状态 s ，采取了某一个动作 a 的概率 $p_\theta(a_t|s_t)$ 。接下来，我们去计算在某一个状态 s 采取了某一个动作 a 之后，到游戏结束为止，累积奖励有多大。我们把这些奖励从时间 t 到时间 T 的奖励通通加起来，并且会在前面乘一个折扣因子，可能设 0.9 或 0.99。我们会减掉一个 baseline b ，减掉这个值 b 的目的，是希望括号这里面这一项

是有正有负的。如果括号里面这一项是正的，我们就要增加在这个状态采取这个动作的机率；如果括号里面是负的，我们就要减少在这个状态采取这个动作的机率。

我们把用 G 来表示累积奖励。但 G 这个值，其实是非常不稳定的。因为互动的过程本身是有随机性的，所以在某一个状态 s 采取某一个动作 a ，然后计算累积奖励，每次算出来的结果都是不一样的，所以 G 其实是一个随机变量。给同样的状态 s ，给同样的动作 a ， G 可能有一个固定的分布。但我们是采取采样的方式，我们在某一个状态 s 采取某一个动作 a ，然后玩到底，我们看看得到多少的奖励，我们就把这个东西当作 G 。

把 G 想成是一个随机变量的话，我们实际上是对这个 G 做一些采样，然后拿这些采样的结果，去更新我们的参数。但实际上在某一个状态 s 采取某一个动作 a ，接下来会发生什么事，它本身是有随机性的。虽然说有个固定的分布，但它本身是有随机性的，而这个随机变量的方差可能会非常大。你在同一个状态采取同一个动作，你最后得到的结果可能会是天差地远的。

假设我们可以采样足够的次数，在每次更新参数之前，我们都可以采样足够的次数，那其实没有什么问题。但问题就是我们每次做 policy gradient，每次更新参数之前都要做一些采样，这个采样的次数其实是不可能太多的，我们只能做非常少量的采样。如果你正好采样到差的结果，比如说你采样到 $G = 100$ ，采样到 $G = -10$ ，那显然你的结果会是很差的。

9.1.2 Review: Q-learning

Review – Q-learning

- State value function $V^\pi(s)$
 - When using actor π , the *cumulated reward* expects to be obtained after visiting state s
- State-action value function $Q^\pi(s, a)$
 - When using actor π , the *cumulated reward* expects to be obtained after taking a at state s

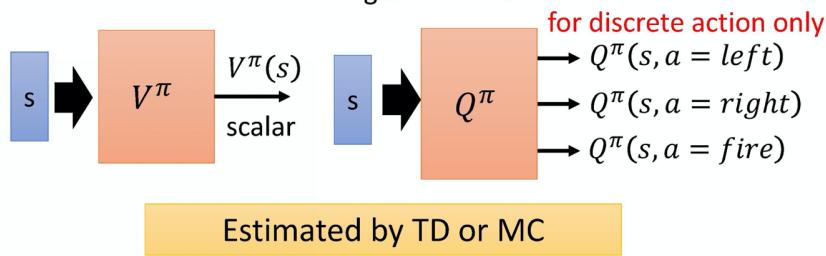


图 9.2

Q: 能不能让整个训练过程变得比较稳定一点，能不能够直接估测 G 这个随机变量的期望值？

A: 我们在状态 s 采取动作 a 的时候，直接用一个网络去估测在状态 s 采取动作 a 的时候， G 的期望值。如果这件事情是可行的，那之后训练的时候，就用期望值来代替采样的值，这样会让训练变得比较稳定。

Q: 怎么拿期望值代替采样的值呢？

A: 这边就需要引入基于价值的 (value-based) 的方法。基于价值的方法就是 Q-learning。Q-learning 有两种函数，有两种 critics。

- 第一种 critic 是 $V^\pi(s)$ ，它的意思是说，假设 actor 是 π ，拿 π 去跟环境做互动，当我们看到状态 s 的时候，接下来累积奖励的期望值有多少。

- 还有一个 critic 是 $Q^\pi(s, a)$ 。 $Q^\pi(s, a)$ 把 s 跟 a 当作输入，它的意思是说，在状态 s 采取动作 a，接下来都用 actor π 来跟环境进行互动，累积奖励的期望值是多少。
- V^π 输入 s，输出一个标量。
- Q^π 输入 s，然后它会给每一个 a 都分配一个 Q value。
- 你可以用 TD 或 MC 来估计。用 TD 比较稳，用 MC 比较精确。

9.1.3 Actor-Critic

Actor-Critic

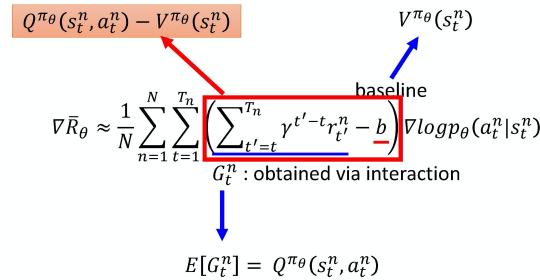


图 9.3

随机变量 G 的期望值正好就是 Q ，即

$$E[G_t^n] = Q^{\pi_\theta}(s_t^n, a_t^n)$$

因为这个就是 Q 的定义。Q-function 的定义就是在某一个状态 s，采取某一个动作 a，假设 policy 就是 π 的情况下会得到的累积奖励的期望值有多大，而这个东西就是 G 的期望值。累积奖励的期望值就是 G 的期望值。

所以假设用 $E[G_t^n]$ 来代表 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ 这一项的话，把 Q-function 套在这里就结束了，我们就可以把 Actor 跟 Critic 这两个方法结合起来。

有不同的方法来表示 baseline，但一个常见的做法是用价值函数 $V^{\pi_\theta}(s_t^n)$ 来表示 baseline。价值函数是说，假设 policy 是 π ，在某一个状态 s 一直互动到游戏结束，期望奖励 (expected reward) 有多大。 $V^{\pi_\theta}(s_t^n)$ 没有涉及到动作， $Q^{\pi_\theta}(s_t^n, a_t^n)$ 涉及到动作。

其实 $V^{\pi_\theta}(s_t^n)$ 会是 $Q^{\pi_\theta}(s_t^n, a_t^n)$ 的期望值，所以 $Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)$ 会有正有负，所以 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b$ 这一项就会是有正有负的。

所以我们就把 policy gradient 里面 $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b$ 这一项换成了 $Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)$ 。

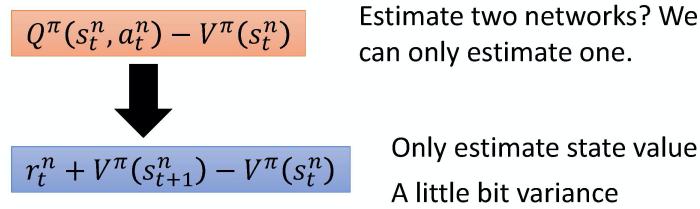
9.1.4 Advantage Actor-Critic

如果你这么实现的话，有一个缺点是：你要估计 2 个网络：Q-network 和 V-network，你估测不准的风险就变成两倍。所以我们何不只估测一个网络？

事实上在这个 Actor-Critic 方法里面。你可以只估测 V 这个网络，你可以用 V 的值来表示 Q 的值， $Q^\pi(s_t^n, a_t^n)$ 可以写成 $r_t^n + V^\pi(s_{t+1}^n)$ 的期望值，即

$$Q^\pi(s_t^n, a_t^n) = E[r_t^n + V^\pi(s_{t+1}^n)]$$

Advantage Actor-Critic



$$Q^\pi(s_t^n, a_t^n) = E[r_t^n + V^\pi(s_{t+1}^n)]$$

$$Q^\pi(s_t^n, a_t^n) = r_t^n + V^\pi(s_{t+1}^n)$$

图 9.4

你在状态 s 采取动作 a , 会得到奖励 r , 然后跳到状态 s_{t+1} 。但是你会得到什么样的奖励 r , 跳到什么样的状态 s_{t+1} , 它本身是有随机性的。所以要把右边这个式子, 取期望值它才会等于 Q-function。但我们现在把期望值这件事情去掉, 即

$$Q^\pi(s_t^n, a_t^n) = r_t^n + V^\pi(s_{t+1}^n)$$

我们就可以把 Q-function 用 $r + V$ 取代掉, 然后得到下式:

$$r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)$$

把这个期望值去掉的好处就是你不需要估计 Q 了, 你只需要估计 V 就够了, 你只要估计一个网络就够了。但这样你就引入了一个随机的东西 r , 它是有随机性的, 它是一个随机变量。但是这个随机变量, 相较于累积奖励 G 可能还好, 因为它是某一个步骤会得到的奖励, 而 G 是所有未来会得到的奖励的总和。 G 的方差比较大, r 虽然也有一些方差, 但它的方差会比 G 要小。所以把原来方差比较大的 G 换成方差比较小的 r 也是合理的。

Q: 为什么可以直接把期望值拿掉?

A: 原始的 A3C paper 试了各种方法, 最后做出来就是这个最好。当然你可能说, 搞不好估计 Q 和 V , 也可以估计很好, 那我告诉你就是做实验的时候, 最后结果就是这个最好, 所以后来大家都用这个。

因为 $r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)$ 叫做 Advantage function。所以这整个方法就叫 Advantage Actor-Critic。

整个流程是这样子的。我们有一个 π , 有个初始的 actor 去跟环境做互动, 先收集资料。在 policy gradient 方法里面收集资料以后, 你就要拿去更新 policy。但是在 actor-critic 方法里面, 你不是直接拿那些资料去更新 policy。你先拿这些资料去估计价值函数, 你可以用 TD 或 MC 来估计价值函数。接下来, 你再基于价值函数, 套用下面这个式子去更新 π 。

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)) \nabla \log p_\theta(a_t^n | s_t^n)$$

然后你有了新的 π 以后, 再去跟环境互动, 再收集新的资料, 去估计价值函数。然后再用新的价值函数去更新 policy, 去更新 actor。

整个 actor-critic 的算法就是这么运作的。

实现 Actor-Critic 的时候, 有两个一定会用的 tip。

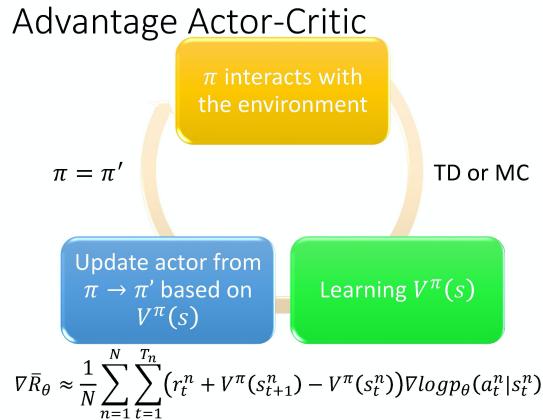
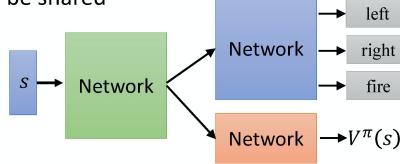


图 9.5

Advantage Actor-Critic

- Tips

- The parameters of actor $\pi(s)$ and critic $V^\pi(s)$ can be shared



- Use output entropy as regularization for $\pi(s)$
- Larger entropy is preferred → exploration

图 9.6

- 第一个 tip 是说，我们需要估计两个网络：V function 和 policy 的网络（也就是 actor）。
 - Critic 网络 $V^\pi(s)$ 输入一个状态，输出一个标量。
 - Actor 网络 $\pi(s)$ 输入一个状态，
 - * 如果动作是离散的，输出就是一个动作的分布。
 - * 如果动作是连续的，输出就是一个连续的向量。
- 上图是举的是离散的例子，但连续的情况也是一样的。输入一个状态，然后它决定你现在要采取哪一个动作。这两个网络，actor 和 critic 的输入都是 s ，所以它们前面几个层 (layer)，其实是可以共享的。
 - 尤其是假设你今天是玩 Atari 游戏，输入都是图像。输入的图像都非常复杂，图像很大，通常你前面都会用一些 CNN 来处理，把那些图像抽象成高级 (high level) 的信息。把像素级别的信息抽象成高级信息这件事情，其实对 actor 跟 critic 来说是可以共用的。所以通常你会让 actor 跟 critic 的共享前面几个层，你会让 actor 跟 critic 的前面几个层共用同一组参数，那这一组参数可能是 CNN 的参数。
 - 先把输入的像素变成比较高级的信息，然后再给 actor 去决定说它要采取什么样的行为，给这个 critic，给价值函数去计算期望奖励。
- 第二个 tip 是我们一样需要探索 (exploration) 的机制。在做 Actor-Critic 的时候，有一个常见的探索的方法是你会对你的 π 的输出的分布下一个约束。这个约束是希望这个分布的熵 (entropy) 不要

太小，希望这个分布的熵可以大一点，也就是希望不同的动作它的被采用的概率平均一点。这样在测试的时候，它才会多尝试各种不同的动作，才会把这个环境探索的比较好，才会得到比较好的结果。这个就是 Advantage Actor-Critic。

9.2 A3C



图 9.7

强化学习有一个问题就是它很慢，那怎么增加训练的速度呢？举个例子，火影忍者就是有一次鸣人说，他想要在一周之内打败晓，所以要加快修行的速度，他老师就教他一个方法：用影分身进行同样修行。两个一起修行的话，经验值累积的速度就会变成 2 倍，所以鸣人就开了 1000 个影分身来进行修行。这个其实就是 Asynchronous(异步的) Advantage Actor-Critic，也就是 A3C 这个方法的精神。

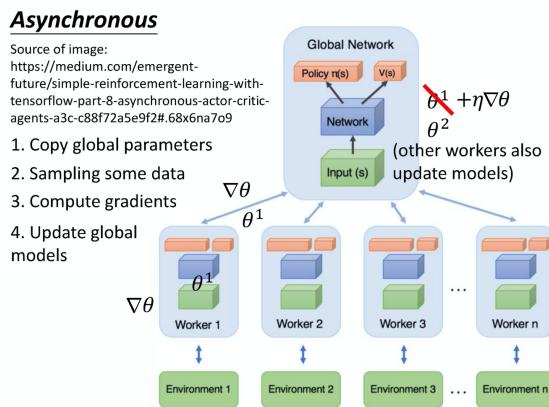


图 9.8

A3C 这个方法就是同时开很多个 worker，那每一个 worker 其实就是一个影分身。那最后这些影分身会把所有的经验，通通集合在一起。你如果没有很多个 CPU，可能也是不好实现的，你可以实现 A2C 就好。

Q: A3C 是怎么运作的？

A:

- A3C 一开始有一个 global network。那我们刚才有讲过，其实 policy network 跟 value network 是绑 (tie) 在一起的，它们的前几个层会被绑一起。我们有一个 global network，它们有包含 policy 的部分和 value 的部分。

- 假设 global network 的参数是 θ_1 , 你会开很多个 worker。每一个 worker 就用一张 CPU 去跑。比如你就开 8 个 worker, 那你至少 8 张 CPU。每一个 worker 工作前都会 global network 的参数复制过来。
- 接下来你就去跟环境做互动, 每一个 actor 去跟环境做互动的时候, 要收集到比较多样性的数据。举例来说, 如果是走迷宫的话, 可能每一个 actor 起始的位置都会不一样, 这样它们才能够收集到比较多样性的数据。
- 每一个 actor 跟环境做互动, 互动完之后, 你就会计算出梯度。计算出梯度以后, 你要拿梯度去更新你的参数。你就计算一下你的梯度, 然后用你的梯度去更新 global network 的参数。就是这个 worker 算出梯度以后, 就把梯度传回给中央的控制中心, 然后中央的控制中心就会拿这个梯度去更新原来的参数。
- 注意, 所有的 actor 都是平行跑的, 每一个 actor 就是各做各的, 不管彼此。所以每个人都是去要了一个参数以后, 做完就把参数传回去。所以当第一个 worker 做完想要把参数传回去的时候, 本来它要的参数是 θ_1 , 等它要把梯度传回去的时候。可能别人已经把原来的参数覆盖掉, 变成 θ_2 了。但是没关系, 它一样会把这个梯度就覆盖过去就是了。Asynchronous actor-critic 就是这么做的, 这个就是 A3C。

9.3 Pathwise Derivative Policy Gradient

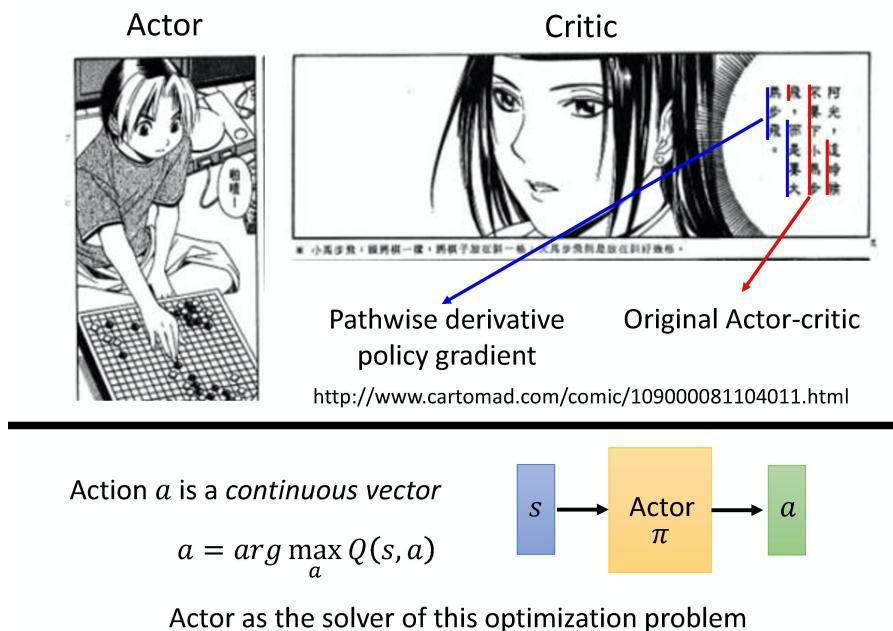


图 9.9

讲完 A3C 之后, 我们要讲另外一个方法叫做 [Pathwise Derivative Policy Gradient](#)。这个方法可以看成是 Q-learning 解连续动作的一种特别的方法, 也可以看成是一种特别的 Actor-Critic 的方法。

用棋灵王来比喻的话, 阿光是一个 actor, 佐为是一个 critic。阿光落某一子以后,

- 如果佐为是一般的 Actor-Critic, 他会告诉阿光说这时候不应该下小马步飞, 他会告诉你, 你现在采取的这一步算出来的 value 到底是好还是不好, 但这样就结束了, 他只告诉你说好还是不好。因为一般的这个 Actor-Critic 里面那个 critic 就是输入状态或输入状态跟动作的对 (pair), 然后给你一个 value 就结束了。所以对 actor 来说, 它只知道它做的这个行为到底是好还是不好。
- 但如果是在 pathwise derivative policy gradient 里面, 这个 critic 会直接告诉 actor 说采取什么样的动作才是好的。所以今天佐为不只是告诉阿光说, 这个时候不要下小马步飞, 同时还告诉阿光说这个

时候应该要下大马步飞，所以这个就是 Pathwise Derivative Policy Gradient 中的 critic。critic 会直接告诉 actor 做什么样的动作才可以得到比较大的 value。

从 Q-learning 的观点来看，Q-learning 的一个问题是你在用 Q-learning 的时候，考虑 continuous vector 会比较麻烦，比较没有通用的解决方法 (general solution)，怎么解这个优化问题呢？

我们用一个 actor 来解这个优化的问题。本来在 Q-learning 里面，如果是一个连续的动作，我们要解这个优化问题。但是现在这个优化问题由 actor 来解，假设 actor 就是一个 solver，这个 solver 的工作就是给定状态 s ，然后它就去解，告诉我们说，哪一个动作可以给我们最大的 Q value，这是从另外一个观点来看 pathwise derivative policy gradient 这件事情。

在 GAN 中也有类似的说法。我们学习一个 discriminator 来评估东西好不好，要 discriminator 生成东西的话，非常困难，那怎么办？因为要解一个 arg max 的问题非常的困难，所以用 generator 来生成。

所以今天的概念其实是一样的，Q 就是那个 discriminator。要根据这个 discriminator 决定动作非常困难，怎么办？另外学习一个网络来解这个优化问题，这个东西就是 actor。

所以两个不同的观点是同一件事。从两个不同的观点来看，一个观点是说，我们可以对原来的 Q-learning 加以改进，我们学习一个 actor 来决定动作以解决 arg max 不好解的问题。

另外一个观点是，原来的 actor-critic 的问题是 critic 并没有给 actor 足够的信息，它只告诉它好或不好，没有告诉它说什么样叫好，那现在有新的方法可以直接告诉 actor 说，什么样叫做好。

Pathwise Derivative Policy Gradient

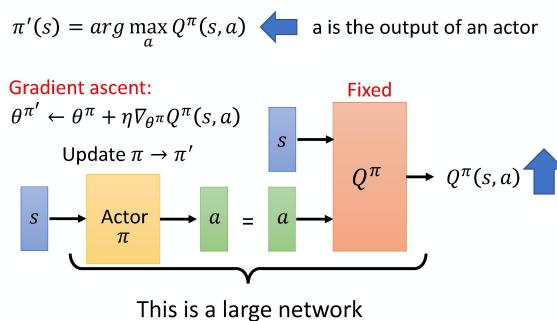


图 9.10

那我们讲一下它的算法。假设我们学习了一个 Q-function，Q-function 就是输入 s 跟 a ，输出就是 $Q^\pi(s, a)$ 。那接下来，我们要学习一个 actor，这个 actor 的工作就是解这个 arg max 的问题。这个 actor 的工作就是输入一个状态 s ，希望可以输出一个动作 a 。这个动作 a 被丢到 Q-function 以后，它可以让 $Q^\pi(s, a)$ 的值越大越好。

那实际上在训练的时候，你其实就是把 Q 跟 actor 接起来变成一个比较大的网络。Q 是一个网络，输入 s 跟 a ，输出一个 value。Actor 在训练的时候，它要做的事情就是输入 s ，输出 a 。把 a 丢到 Q 里面，希望输出的值越大越好。在训练的时候会把 Q 跟 actor 接起来，当作是一个大的网络。然后你会固定住 Q 的参数，只去调 actor 的参数，就用 gradient ascent 的方法去最大化 Q 的输出。这就是一个 GAN，这就是 conditional GAN。Q 就是 discriminator，但在强化学习就是 critic，actor 在 GAN 里面就是 generator，其实它们就是同一件事情。

我们来看一下 pathwise derivative policy gradient 的算法。一开始你会有一个 actor π ，它去跟环境互动，然后，你可能会要它去估计 Q value。估计完 Q value 以后，你就把 Q value 固定，只去学习一个 actor。假设这个 Q 估得是很准的，它知道在某一个状态采取什么样的动作，会真的得到很大的 value。接下来就学习这个 actor，actor 在给定 s 的时候，它采取了 a ，可以让最后 Q-function 算出来的 value 越大越好。你用这个 criteria 去更新你的 actor π 。然后有新的 π 再去跟环境做互动，再估计 Q，再得到新的 π

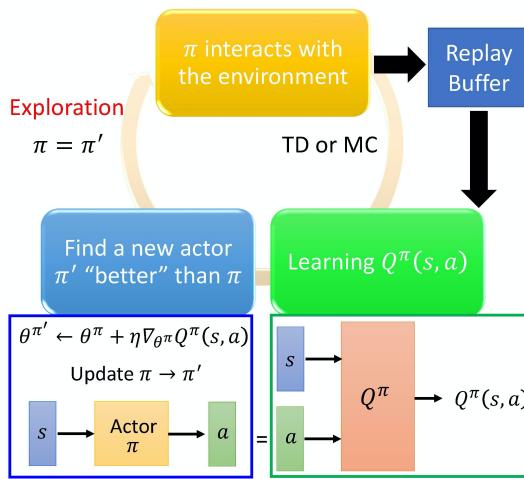


图 9.11

去最大化 Q 的输出。本来在 Q-learning 里面，你用得上的技巧，在这边也几乎都用得上，比如说 replay buffer、exploration 等等。

Q-learning Algorithm

- Initialize Q-function Q , target Q-function $\hat{Q} = Q$
- In each episode
 - For each time step t
 - Given state s_t , take action a_t based on Q (exploration)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
 - Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
 - Every C steps reset $\hat{Q} = Q$

图 9.12

上图是原来 Q-learning 的算法。你有一个 Q-function Q ，你有另外一个目标的 Q-function 叫做 \hat{Q} 。然后在每一次训练，在每一个回合的每一个时间点里面，你会看到一个状态 s_t ，你会采取某一个动作 a_t 。至于采取哪一个动作是由 Q-function 所决定的，因为解一个 arg max 的问题。如果是离散的话没有问题，你就看说哪一个 a 可以让 Q 的 value 最大，就采取哪一个动作。那你需要加一些探索，这样表现才会好。你会得到奖励 r_t ，跳到新的状态 s_{t+1} 。你会把 s_t, a_t, r_t, s_{t+1} 塞到你的 buffer 里面去。你会从你的 buffer 里面采样一个批量的数据，在这个批量数据里面，可能某一笔是 s_i, a_i, r_i, s_{i+1} 。接下来你会算一个目标，这个目标叫做 y ， $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$ 。然后怎么学习你的 Q 呢？你希望 $Q(s_i, a_i)$ 跟 y 越接近越好，这是一个回归的问题，最后每 C 个步骤，你要把用 Q 替代 \hat{Q} 。

接下来我们把 Q-learning 改成 Pathwise Derivative Policy Gradient，这边需要做四个改变。

- 第一个改变是，你要把 Q 换成 π ，本来是用 Q 来决定在状态 s_t 产生那一个动作， a_t 现在是直接用 π 。我们不用再解 arg max 的问题了，我们直接学习了一个 actor。这个 actor 输入 s_t 就会告诉我们应该采取哪一个 a_t 。所以本来输入 s_t ，采取哪一个 a_t ，是 Q 决定的。在 Pathwise Derivative Policy Gradient 里面，我们会直接用 π 来决定，这是第一个改变。
- 第二个改变是，本来这个地方是要计算在 s_{i+1} ，根据你的 policy 采取某一个动作 a 会得到多少的 Q

Q-learning Algorithm ➡ *Pathwise Derivative Policy Gradient*

- Initialize Q-function Q , target Q-function $\hat{Q} = Q$, actor π , target actor $\hat{\pi} = \pi$
- In each episode
 - For each time step t
 - 1 • Given state s_t , take action a_t based on π (exploration)
 - Obtain reward r_t , and reach new state s_{t+1}
 - Store (s_t, a_t, r_t, s_{t+1}) into buffer
 - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
 - 2 • Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a) \hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
 - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
 - 3 • Update the parameters of π to maximize $Q(s_i, \pi(s_i))$
 - Every C steps reset $\hat{Q} = Q$
 - 4 • Every C steps reset $\hat{\pi} = \pi$

图 9.13

value。那你会采取让 \hat{Q} 最大的那个动作 a 。那现在因为我们其实不好解这个 $\arg \max$ 的问题，所以 $\arg \max$ 问题，其实现在就是由 policy π 来解了，所以我们就直接把 s_{i+1} 代到 policy π 里面，你就会知道说给定 s_{i+1} ，哪一个动作会给我们最大的 Q value，那你在这边就会采取那一个动作。在 Q-function 里面，有两个 Q network，一个是真正的 Q network，另外一个是目标 Q network。那实际上你在实现这个算法的时候，你也会有两个 actor，你会有一个真正要学习的 actor π ，你会有一个目标 actor $\hat{\pi}$ 。这个原理就跟为什么要有目标 Q network 一样，我们在算目标 value 的时候，我们并不希望它一直的变动，所以我们会有一个目标的 actor 和一个目标的 Q-function，它们平常的参数就是固定住的，这样可以让你的这个目标的 value 不会一直地变化。所以来到底是要用哪一个动作 a ，你会看说哪一个动作 a 可以让 \hat{Q} 最大。但现在因为哪一个动作 a 可以让 \hat{Q} 最大这件事情已经用 policy 取代掉了，所以我们要知道哪一个动作 a 可以让 \hat{Q} 最大，就直接把那个状态带到 $\hat{\pi}$ 里面，看它得到哪一个 a ，那个 a 就是会让 $\hat{Q}(s, a)$ 的值最大的那个 a 。其实跟原来的这个 Q-learning 也是没什么不同，只是原来你要解 $\arg \max$ 的地方，通通都用 policy 取代掉了，那这个是第二个不同。

- 第三个不同就是之前只要学习 Q ，现在你多学习一个 π ，那学习 π 的时候的方向是什么呢？学习 π 的目的，就是为了最大化 Q-function，希望你得到的这个 actor，它可以让你的 Q-function 输出越来越好，这个跟学习 GAN 里面的 generator 的概念。其实是一样的。
- 第四个步骤，就跟原来的 Q-function 一样。你要把目标的 Q network 取代掉，你现在也要把目标 policy 取代掉。

9.4 Connection with GAN

其实 GAN 跟 Actor-Critic 的方法是非常类似的。这边就不细讲，你可以去找到一篇 paper 叫做 [Connecting Generative Adversarial Network and Actor-Critic Methods](#)。

Q: 知道 GAN 跟 Actor-Critic 非常像有什么帮助呢？

A: 一个很大的帮助就是 GAN 跟 Actor-Critic 都是以难训练而闻名的。所以在文献上就会收集各式各样的方法，告诉你说怎么样可以把 GAN 训练起来。怎么样可以把 Actor-Critic 训练起来。但是因为做 GAN 跟 Actor-Critic 的人是两群人，所以这篇 paper 里面就列出说在 GAN 上面有哪些技术是有人做过的，在 Actor-Critic 上面，有哪些技术是有人做过的。也许在 GAN 上面有试过的技术，你可以试着应用在 Actor-Critic 上，在 Actor-Critic 上面做过的技术，你可以试着应用在 GAN 上面，看看是否 work。

Connection with GAN

Method	GANs	AC
Freezing learning	yes	yes
Label smoothing	yes	no
Historical averaging	yes	no
Minibatch discrimination	yes	no
Batch normalization	yes	yes
Target networks	n/a	yes
Replay buffers	no	yes
Entropy regularization	no	yes
Compatibility	no	yes

David Pfau, Oriol Vinyals, "Connecting Generative Adversarial Networks and Actor-Critic Methods", arXiv preprint, 2016

图 9.14

9.5 Keywords

- A2C: Advantage Actor-Critic 的缩写，一种 Actor-Critic 方法。
- A3C: Asynchronous (异步的) Advantage Actor-Critic 的缩写，一种改进的 Actor-Critic 方法，通过异步的操作，进行 RL 模型训练的加速。
- Pathwise Derivative Policy Gradient: 其为使用 Q-learning 解 continuous action 的方法，也是一种 Actor-Critic 方法。其会对于 actor 提供 value 最大的 action，而不仅仅是提供某一个 action 的好坏程度。

9.6 Questions

- 整个 Advantage actor-critic (A2C) 算法的工作流程是怎样的？

答：在传统的方法中，我们有一个 policy π 以及一个初始的 actor 与 environment 做互动，收集数据以及反馈。通过这些每一步得到的数据与反馈，我们就要进一步更新我们的 policy π ，通常我们所使用的方式是 policy gradient。但是对于 actor-critic 方法，我们不是直接使用每一步得到的数据和反馈进行 policy π 的更新，而是使用这些数据进行 estimate value function，这里我们通常使用的算法包括前几个 chapters 重点介绍的 TD 和 MC 等算法以及他们的优化算法。接下来我们再基于 value function 来更新我们的 policy，公式如下：

$$\nabla \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{T_n} (r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)) \nabla \log p_\theta(a_t^n | s_t^n)$$

其中，上式中的 $r_t^n + V^\pi(s_{t+1}^n) - V^\pi(s_t^n)$ 我们称为 Advantage function，我们通过上式得到新的 policy 后，再去与 environment 进行交互，然后再重复我们的 estimate value function 的操作，再用 value function 来更新我们的 policy。以上的整个方法我们称为 Advantage Actor-Critic。

- 在实现 Actor-Critic 的时候，有哪些我们用到的 tips？

答：与我们上一章讲述的东西有关：

1. estimate 两个 network：一个是 estimate V function，另外一个是 policy 的 network，也就是你的 actor。V-network 的 input 是一个 state，output 是一个 scalar。然后 actor 这个 network 的 input

是一个 state, output 是一个 action 的 distribution。这两个 network, actor 和 critic 的 input 都是 s , 所以它们前面几个 layer, 其实是可以 share 的。尤其是假设你今天是玩 Atari 游戏, input 都是 image。那 input 那个 image 都非常复杂, image 很大, 通常前面都会用一些 CNN 来处理, 把那些 image 抽象成 high level 的 information, 所以对 actor 跟 critic 来说是可以共用的。我们可以让 actor 跟 critic 的前面几个 layer 共用同一组参数。那这一组参数可能是 CNN。先把 input 的 pixel 变成比较 high level 的信息, 然后再给 actor 去决定说它要采取什么样的行为, 给这个 critic, 给 value function 去计算 expected reward。

2. exploration 机制: 其目的是对 policy π 的 output 的分布进行一个限制, 从而使得 distribution 的 entropy 不要太小, 即希望不同的 action 被采用的机率平均一点。这样在 testing 的时候, 它才会多尝试各种不同的 action, 才会把这个环境探索的比较好, 才会得到比较好的结果。

- A3C (Asynchronous Advantage Actor-Critic) 在训练时回有很多的 worker 进行异步的工作, 最后再讲他们所获得的“结果”再集合到一起。那么其具体的如何运作的呢?

答: A3C 一开始会有一个 global network。它们有包含 policy 的部分和 value 的部分, 假设它的参数就是 θ_1 。对于每一个 worker 都用一张 CPU 训练 (举例子说明), 第一个 worker 就把 global network 的参数 copy 过来, 每一个 worker 工作前都会 global network 的参数 copy 过来。然后这个 worker 就要去跟 environment 进行交互, 每一个 actor 去跟 environment 做互动后, 就会计算出 gradient 并且更新 global network 的参数。这里要注意的是, 所有的 actor 都是平行跑的、之间没有交叉。所以每个 worker 都是在 global network “要”了一个参数以后, 做完就把参数传回去。所以当第一个 worker 做完想要把参数传回去的时候, 本来它要的参数是 θ_1 , 等它要把 gradient 传回去的时候。可能别人已经把原来的参数覆盖掉, 变成 θ_2 了。但是没关系, 它一样会把这个 gradient 就覆盖过去就是了。

- 对比经典的 Q-learning 算法, 我们的 Pathwise Derivative Policy Gradient 有哪些改进之处?

答:

1. 首先, 把 $Q(s, a)$ 换成了 π , 之前是用 $Q(s, a)$ 来决定在 state s_t 产生那一个 action, a_t 现在是直接用 π 。原先我们需要解 argmax 的问题, 现在我们直接训练了一个 actor。这个 actor input s_t 就会告诉我们应该采取哪一个 a_t 。综上, 本来 input s_t , 采取哪一个 a_t , 是 $Q(s, a)$ 决定的。在 Pathwise Derivative Policy Gradient 里面, 我们会直接用 π 来决定。

2. 另外, 原本是要计算在 s_{i+1} 时对应的 policy 采取的 action a 会得到多少的 Q value, 那你会采取让 \hat{Q} 最大的那个 action a。现在因为我们不需要再解 argmax 的问题。所以现在我们就直接把 s_{i+1} 代入到 policy π 里面, 直接就会得到在 s_{i+1} 下, 哪一个 action 会给我们最大的 Q value, 那你在这边就会 take 那一个 action。在 Q-function 里面, 有两个 Q network, 一个是真正的 Q network, 另外一个是 target Q network。那实际上你在 implement 这个 algorithm 的时候, 你也会有两个 actor, 你会有一个真正要 learn 的 actor π , 你会有一个 target actor $\hat{\pi}$ 。但现在因为哪一个 action a 可以让 \hat{Q} 最大这件事情已经被用那个 policy 取代掉了, 所以我们要知道哪一个 action a 可以让 \hat{Q} 最大, 就直接把那个 state 带到 $\hat{\pi}$ 里面, 看它得到哪一个 a, 就用那一个 a, 其也就是会让 $\hat{Q}(s, a)$ 的值最大的那个 a。

3. 还有, 之前只要 learn Q, 现在你多 learn 一个 π , 其目的在于 maximize Q-function, 希望你得到的这个 actor, 它可以让你的 Q-function output 越大越好, 这个跟 learn GAN 里面的 generator 的概念类似。

4. 最后, 与原来的 Q-function 一样。我们要把 target 的 Q-network 取代掉, 你现在也要把 target policy 取代掉。

9.7 Something About Interview

- 高冷的面试官: 请简述一下 A3C 算法吧, 另外 A3C 是 on-policy 还是 off-policy 呀?

答：A3C 就是异步优势演员-评论家方法（Asynchronous Advantage Actor-Critic）：评论家学习值函数，同时有多个 actor 并行训练并且不时与全局参数同步。A3C 旨在用于并行训练，是 on-policy 的方法。

- 高冷的面试官：请问 Actor - Critic 有何优点呢？

答：

- 相比以值函数为中心的算法，Actor - Critic 应用了策略梯度的做法，这能让它在连续动作或者高维动作空间中选取合适的动作，而 Q-learning 做这件事会很困难甚至瘫痪。

- 相比单纯策略梯度，Actor - Critic 应用了 Q-learning 或其他策略评估的做法，使得 Actor Critic 能进行单步更新而不是回合更新，比单纯的 Policy Gradient 的效率要高。

- 高冷的面试官：请问 A3C 算法具体是如何异步更新的？

答：下面是算法大纲：

- 定义全局参数 θ 和 w 以及特定线程参数 π 和 w 。

- 初始化时间步 $t = 1$ 。

- 当 $T \leq T_{max}$ ：

- * 重置梯度： $d = 0$ 并且 $dw = 0$ 。

- * 将特定于线程的参数与全局参数同步： $\pi = \pi_g$ 以及 $w = w_g$ 。

- * 令 $t_{start} = t$ 并且随机采样一个初始状态 s_t 。

- * 当 ($s_t \neq$ 终止状态) 并 $t - t_{start} \leq t_{max}$ ：

- 根据当前线程的策略选择当前执行的动作 a_t ($a_t|s_t$)，执行动作后接收回报 r_t 然后转移到下一个状态 s_{t+1} 。

- 更新 t 以及 T : $t=t+1$ 并且 $T=T+1$ 。

- * 初始化保存累积回报估计值的变量

- * 对于 $i = t_1, \dots, t_{start}$:

- $r \leftarrow r + r_i$; 这里 r 是 G_i 的蒙特卡洛估计。

- 累积关于参数 π 的梯度： $d \leftarrow d + \log(a_i|s_i)(r - Vw(s_i))$;

- 累积关于参数 w 的梯度： $dw \leftarrow dw + 2(r - Vw(s_i)) w (r - Vw(s_i))$.

- * 分别使用 d 以及 dw 异步更新 π 以及 w 。

- 高冷的面试官：Actor-Critic 两者的区别是什么？

答：Actor 是策略模块，输出动作；critic 是判别器，用来计算值函数。

- 高冷的面试官：actor-critic 框架中的 critic 起了什么作用？

答：critic 表示了对于当前决策好坏的衡量。结合策略模块，当 critic 判别某个动作的选择时有益的，策略就更新参数以增大该动作出现的概率，反之降低动作出现的概率。

- 高冷的面试官：简述 A3C 的优势函数？

答： $A(s, a) = Q(s, a) - V(s)$ 是为了解决 value-based 方法具有高变异性。它代表着与该状态下采取的平均行动相比所取得的进步。

- 如果 $A(s, a) > 0$: 梯度被推向了该方向

- 如果 $A(s, a) < 0$: (我们的 action 比该 state 下的平均值还差) 梯度被推向了反方

但是这样就需要两套 value function，所以可以使用 TD error 做估计： $A(s, a) = r + \gamma V(s') - V(s)$ 。

References

- 神经网络与深度学习

第 10 章 Sparse Reward

实际上用 reinforcement learning learn agent 的时候，多数的时候 agent 都是没办法得到 reward 的。在没有办法得到 reward 的情况下，训练 agent 是非常困难的。举例来说，假设你要训练一个机器手臂，然后桌上有一个螺丝钉跟螺丝起子，那你要训练它用螺丝起子把螺丝钉栓进去，这个很难，为什么？因为一开始你的 agent 是什么都不知道的，它唯一能够做不同的 action 的原因是 exploration。举例来说，你在做 Q-learning 的时候，会有一些随机性，让它去采取一些过去没有采取过的 action，那你要随机到说，它把螺丝起子捡起来，再把螺丝栓进去，然后就会得到 reward 1，这件事情是永远不可能发生的。所以，不管你的 actor 做了什么事情，它得到 reward 永远都是 0，对它来说不管采取什么样的 action 都是一样糟或者是一样的好。所以，它最后什么都不会学到。

如果环境中的 reward 非常 sparse，reinforcement learning 的问题就会变得非常的困难，但是人类可以在非常 sparse 的 reward 上面去学习。我们的人生通常多数的时候，我们就只是活在那里，都没有得到什么 reward 或是 penalty。但是，人还是可以采取各种各式各样的行为。所以，一个真正厉害的 AI 应该能够在 sparse reward 的情况下也学到要怎么跟这个环境互动。

我们可以通过三个方向来解决 sparse reward 的问题。

10.0.1 Reward Shaping

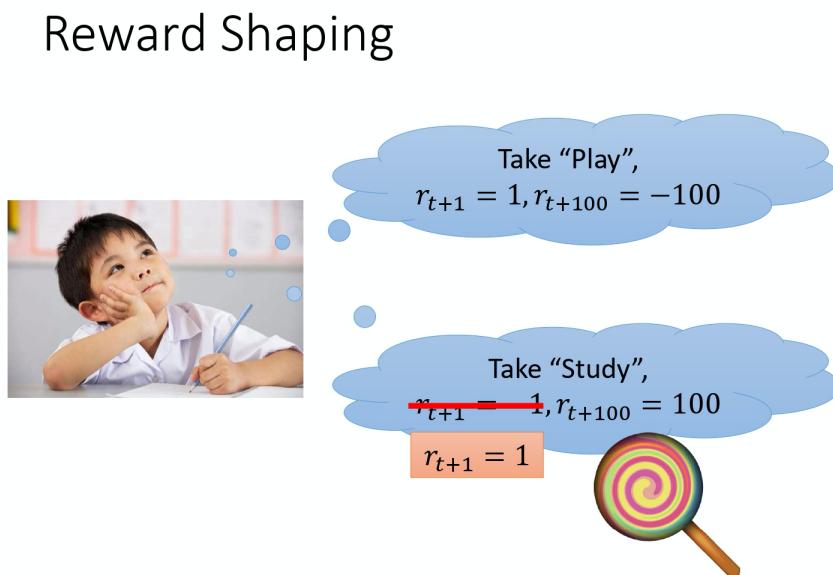


图 10.1

第一个方向是 [reward shaping](#)。Reward shaping 的意思是说环境有一个固定的 reward，它是真正的 reward，但是为了让 agent 学出来的结果是我们要的样子，我们刻意地设计了一些 reward 来引导我们的 agent。

举例来说，如果是把小孩当成一个 agent 的话。那一个小孩，他可以 take 两个 actions，一个 action 是他可以出去玩，那他出去玩的话，在下一秒钟他会得到 reward 1。但是他在月考的时候，成绩可能会很差。所以在 100 个小时之后呢，他会得到 reward -100。然后，他也可以决定要念书，然后在下一个时间，因为他没有出去玩，所以他觉得很不爽，所以他得到 reward -1。但是在 100 个小时后，他可以得到 reward 100。但对一个小孩来说，他可能就会想要 take play 而不是 take study。我们计算的是 accumulated reward，但也许对小孩来说，他的 discount factor 会很大，所以他就不在意未来的 reward。而且因为

他是一个小孩，他还没有很多 experience，所以他的 Q-function estimate 是非常不精准的。所以要他去 estimate 很远以后会得到的 accumulated reward，他其实是预测不出来的。所以这时候大人就要引导他，怎么引导呢？就骗他说，如果你坐下来念书我就给你吃一个棒棒糖。所以，对他来说，下一个时间点会得到的 reward 就变成是 positive 的。所以他觉得说，也许 take 这个 study 是比 play 好的。虽然这并不是真正的 reward，而是其他人骗他的 reward，告诉他说你采取这个 action 是好的。Reward shaping 的概念是一样的，简单来说，就是你自己想办法 design 一些 reward，它不是环境真正的 reward。在玩 Atari 游戏里面，真的 reward 是游戏主机给你的 reward，但你自己去设计一些 reward 好引导你的 machine，做你想要它做的事情。

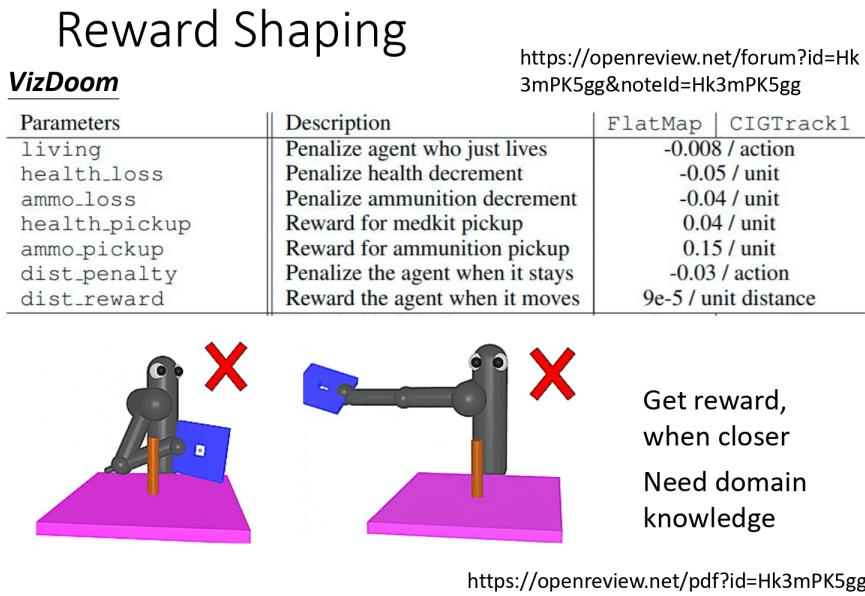


图 10.2

举例来说，这个例子是 Facebook 玩 VizDoom 的 agent。VizDoom 是一个第一人射击游戏，在这个射击游戏中，杀了敌人就得到 positive reward，被杀就得到 negative reward。他们设计了一些新的 reward，用新的 reward 来引导 agent 让他们做得更好，这不是游戏中真正的 reward。比如说掉血就扣 0.05 的分数，弹药减少就扣分，捡到补给包就加分，呆在原地就扣分，移动就加分。活着会扣一个很小的分数，因为不这样做的话，machine 会只想活着，一直躲避敌人，这样会让 machine 好战一点。表格中的参数都是调出来的。

Reward shaping 是有问题的，因为我们需要 domain knowledge，举例来说，机器人想要学会的事情是把蓝色的板子从这个柱子穿过去。机器人很难学会，我们可以做 reward shaping。一个貌似合理的说法是，蓝色的板子离柱子越近，reward 越大。但是 machine 靠近的方式会有问题，它会用蓝色的板子打柱子。而我们要把蓝色板子放在柱子上面去，才能把蓝色板子穿过柱子。这种 reward shaping 的方式是没有帮助的，那至于什么 reward shaping 有帮助，什么 reward shaping 没帮助，会变成一个 domain knowledge，你要去调的。

10.0.2 Curiosity

接下来就是介绍各种你可以自己加进去，in general 看起来是有用的 reward。举例来说，一个技术是给 machine 加上 curiosity，所以叫 *curiosity driven reward*。如上图所示，我们有一个 reward function，它给你某一个 state，给你某一个 action，它就会评断说在这个 state 采取这个 action 得到多少的 reward。那我们当然希望 total reward 越大越好。

<https://arxiv.org/abs/1705.05363>

Curiosity

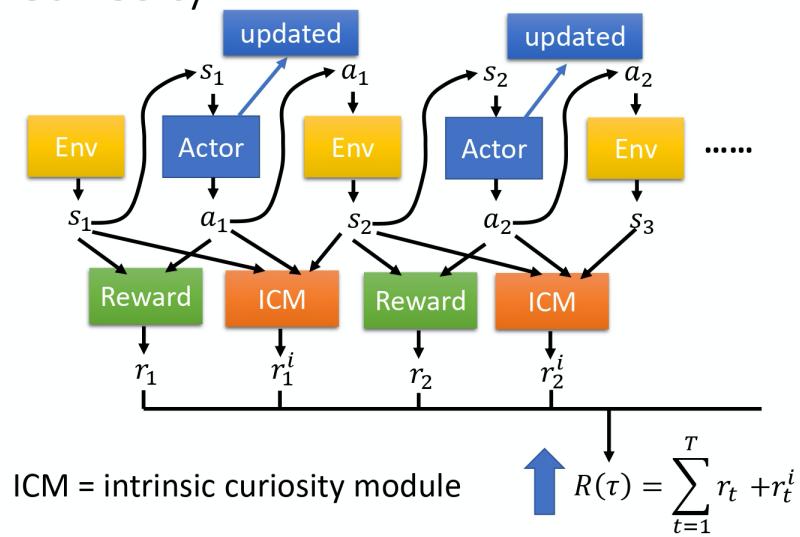


图 10.3

在 curiosity driven 的这种技术里面，你会加上一个新的 reward function。这个新的 reward function 叫做 **ICM(intrinsic curiosity module)**，它就是要给机器加上好奇心。ICM 会吃 3 个东西，它会吃 state s_1 、action a_1 和 state s_2 。根据 s_1 、 a_1 、 s_2 ，它会 output 另外一个 reward r_1^i 。对 machine 来说，total reward 并不是只有 r 而已，还有 r^i 。它不是只有把所有的 r 都加起来，它还把所有 r^i 加起来当作 total reward。所以，它在跟环境互动的时候，它不是只希望 r 越大越好，它还同时希望 r^i 越大越好，它希望从 ICM 的 module 里面得到的 reward 越大越好。ICM 就代表了一种 curiosity。

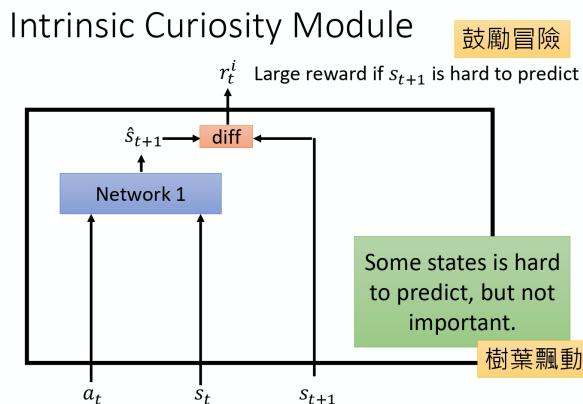


图 10.4

怎么设计这个 ICM？这个是最原始的设计。这个设计是这样，curiosity module 就是 input 3 个东西，input 现在的 state，input 在这个 state 采取的 action，然后 input 下一个 state s_{t+1} 。接下来会 output 一个 reward r_t^i 。那这个 r_t^i 是怎么算出来的呢？在 ICM 里面，你有一个 network，这个 network 会 take a_t 跟 s_t ，然后去 output \hat{s}_{t+1} ，也就是这个 network 根据 a_t 和 s_t 去 predict \hat{s}_{t+1} 。接下来再看说，这个 network 的预测 \hat{s}_{t+1} 跟真实的情况 s_{t+1} 像不像，越不像那得到的 reward 就越大。所以这个 reward r_t^i 的意思是说，如果未来的 state 越难被预测的话，那得到的 reward 就越大。这就是鼓励 machine 去冒

险，现在采取这个 action，未来会发生什么事越没有办法预测的话，这个 action 的 reward 就大。所以如果有这样子的 ICM，machine 就会倾向于采取一些风险比较大的 action，它想要去探索未知的世界，它想要去看看说，假设某一个 state 是它没有办法预测，它会特别去想要采取那个 state，这可以增加 machine exploration 的能力。

这个 network 1 其实是另外 train 出来的。Training 的时候，这个 network 1，你会给它 a_t 、 s_t 、 s_{t+1} ，然后让这个 network 1 去学说 given a_t, s_t ，怎么 predict \hat{s}_{t+1} 。Apply 到 agent 互动的时候，其实要把 ICM module fix 住。其实，这一整个想法里面是有一个问题的。这个问题是某一些 state 它很难被预测并不代表它就是好的，它就应该要去被尝试的。举例来说，俄罗斯轮盘的结果也是没有办法预测的，并不代表说，人应该每天去玩俄罗斯轮盘这样子。所以只是鼓励 machine 去冒险是不够的，因为如果光是只有这个 network 的架构，machine 只知道说什么东西它无法预测。如果在某一个 state 采取某一个 action，它无法预测接下来结果，它就会采取那个 action，但并不代表这样的结果一定是好的。举例来说，可能在某个游戏里面，背景会有风吹草动，会有树叶飘动。那也许树叶飘动这件事情，是很难被预测的，对 machine 来说它在某一个 state 什么都不做，看着树叶飘动，然后，发现这个树叶飘动是没有办法预测的，接下来它就会一直站在那边，看树叶飘动。所以说，光是有好奇心是不够的，还要让它知道说，什么事情是真正重要的。

Intrinsic Curiosity Module

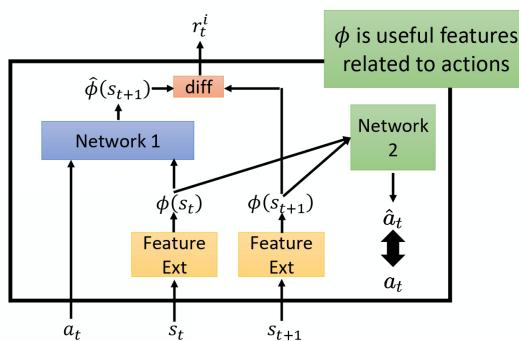


图 10.5

怎么让 machine 知道说什么事情是真正重要的？你要加上另外一个 module，我们要 learn 一个 [feature extractor](#)，黄色的格子代表 feature extractor，它是 input 一个 state，然后 output 一个 feature vector 来代表这个 state，那我们期待这个 feature extractor 可以把那种没有意义的画面，state 里面没有意义的东西把它过滤掉，比如说风吹草动、白云的飘动、树叶的飘动这种没有意义的东西直接把它过滤掉，

假设这个 feature extractor 真的可以把无关紧要的东西过滤掉以后，network 1 实际上做的事情是，给它一个 actor，给它一个 state s_t 的 feature representation，让它预测 state s_{t+1} 的 feature representation。接下来我们再看说，这个预测的结果跟真正的 state s_{t+1} 的 feature representation 像不像，越不像，reward 就越大。怎么 learn 这个 feature extractor 呢？让这个 feature extractor 可以把无关紧要的事情滤掉呢？这边的 learn 法就是 learn 另外一个 network 2。这个 network 2 是吃 $\phi(s_t)$ 、 $\phi(s_{t+1})$ 这两个 vector 当做 input，然后接下来它要 predict action a 是什么，然后它希望呢这个 action a 跟真正的 action a 越接近越好。这个 network 2 会 output 一个 action，它 output 说，从 state s_t 跳到 state s_{t+1} ，要采取哪一个 action 才能够做到，那希望这个 action 跟真正的 action 越接近越好。加上这个 network 2 的好处就是因为要用 $\phi(s_t)$ 、 $\phi(s_{t+1})$ 预测 action。所以，今天我们抽出来的 feature 跟预测 action 这件事情是有关的。所以风吹草动等与 machine 要采取的 action 无关的东西就会被滤掉，就不会被放在抽出来的 vector representation 里面。

10.1 Curriculum Learning

Curriculum Learning

- Starting from simple training examples, and then becoming harder and harder.

VizDoom

	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7
Speed	0.2	0.2	0.4	0.4	0.6	0.8	0.8	1.0
Health	40	40	40	60	60	60	80	100

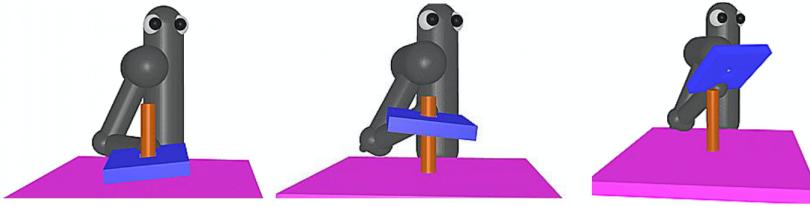


图 10.6

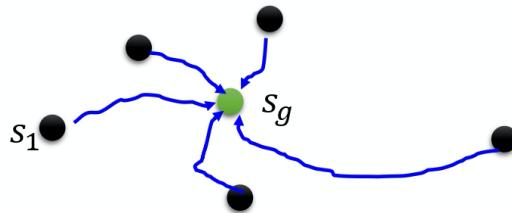
第二个方向是 [curriculum learning](#)。Curriculum learning 不是 reinforcement learning 所独有的概念，其实在 machine learning，尤其是 deep learning 里面，你都会用到 curriculum learning 的概念。举例来说，curriculum learning 的意思是说，你为机器的学习做规划，你给他喂 training data 的时候，是有顺序的，通常都是由简单到难。就好比说，假设你今天要教一个小朋友作微积分，他做错就打他一巴掌，这样他永远都不会做对，太难了。你要先教他九九乘法，然后才教他微积分。所以 curriculum learning 的意思就是在教机器的时候，从简单的题目教到难的题目。就算不是 reinforcement learning，一般在 train deep network 的时候，你有时候也会这么做。举例来说，在 train RNN 的时候，已经有很多的文献都 report 说，你给机器先看短的 sequence，再慢慢给它长的 sequence，通常可以学得比较好。那用在 reinforcement learning 里面，你就是要帮机器规划一下它的课程，从最简单的到最难的。

举例来说，在 Facebook 玩 VizDoom 的 agent 里面，Facebook 玩 VizDoom 的 agent 蛮强的。他们在参加这个 VizDoom 的比赛，机器的 VizDoom 比赛是得第一名的，他们是有为机器规划课程的。先从课程 0 一直上到课程 7。在这个课程里面，怪物的速度跟血量是不一样的。所以，在越进阶的课程里面，怪物的速度越快，然后他的血量越多。在 paper 里面也有讲说，如果直接上课程 7，machine 是学不起来的。你就是要从课程 0 一路玩上去，这样 machine 才学得起来。

再举个例子，把蓝色的板子穿过柱子，怎么让机器一直从简单学到难呢？

- 如第一张图所示，也许一开始机器初始的时候，它的板子就已经在柱子上了。这个时候，机器要做的事情只有把蓝色的板子压下去，就结束了。这比较简单，它应该很快就学的会。它只有往上跟往下这两个选择嘛，往下就得到 reward，就结束了，他也不知道学的是什么。
- 如第二张图所示，这边就是把板子挪高一点，挪高一点，所以它有时候会很笨的往上拉，然后把板子拿出来了。如果它压板子学得会的话，拿板子也比较有机会学得会。假设它现在学的到说，只要板子接近柱子，它就可以把这个板子压下去的话。接下来，你再让它学更 general 的 case。
- 如第三张图所示，一开始，让板子离柱子远一点。然后，板子放到柱子上面的时候，它就会知道把板子压下去，这个就是 curriculum learning 的概念。当然 curriculum learning 有点 ad hoc(特别)，就是需要人去为机器设计它的课程。

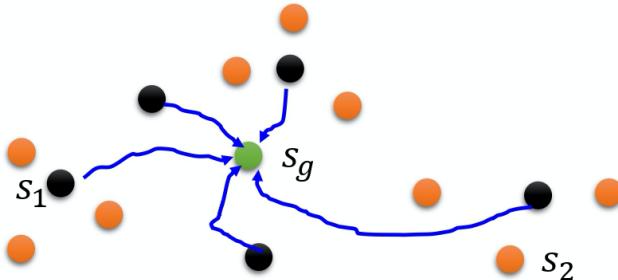
10.1.1 Reverse Curriculum Generation



- Given a goal state s_g .
- Sample some states s_1 “close” to s_g
- Start from states s_1 , each trajectory has reward $R(s_1)$

图 10.7

有一个比较 general 的方法叫做 Reverse Curriculum Generation。你可以用一个比较通用的方法来帮机器设计课程，这个比较通用的方法是怎么样呢？假设你现在一开始有一个 state s_g ，这是你的 gold state，也就是最后最理想的结果。如果拿刚才那个板子和柱子的实验作为例子的话，就把板子放到柱子里面，这样子叫做 gold state。你就已经完成了，或者你让机器去抓东西，你训练一个机器手臂抓东西，抓到东西以后叫做 gold state。接下来你根据你的 gold state 去找其他的 state，这些其他的 state 跟 gold state 是比较接近的。举例来说，如果是让机器抓东西的例子里面，你的机器手臂可能还没有抓到东西。假设这些跟 gold state 很近的 state 叫做 s_1 。你的机械手臂还没有抓到东西，但它离 gold state 很近，那这个叫做 s_1 。至于什么叫做近，这是 case dependent，你要根据你的 task 来 design 说怎么从 s_g sample 出 s_1 。如果是机械手臂的例子，可能就比较好想。其他例子可能就比较难想。接下来呢，你再从这些 s_1 开始做互动，看它能不能够达到 gold state s_g ，那每一个 state，你跟环境做互动的时候，你都会得到一个 reward R 。



- Delete s_1 whose reward is too large (already learned) or too small (too difficult at this moment)
- Sample s_2 from s_1 , start from s_2

图 10.8

接下来，我们把 reward 特别极端的 case 去掉。Reward 特别极端的 case 的意思就是说那些 case 太简单或是太难了。如果 reward 很大，代表说这个 case 太简单了，就不用学了，因为机器已经会了，它可以得到很大的 reward。如果 reward 太小，代表这个 case 太难了，依照机器现在的能力这个课程太难了，它学不会，所以就不要学这个，所以只找一些 reward 适中的 case。

什么叫做适中，这个就是你要调的参数，找一些 reward 适中的 case。接下来，再根据这些 reward 适中的 case 去 sample 出更多的 state。假设你一开始，你机械手臂在这边，可以抓的到以后。接下来，就再离远一点，看看能不能够抓得到，又抓的到以后，再离远一点，看看能不能抓得到。这是一个有用的方法，它叫做 [Reverse Curriculum learning](#)。刚才讲的是 curriculum learning，就是你要为机器规划它学习的顺序。而 reverse curriculum learning 是从 gold state 去反推，就是说你原来的目标是长这个样子，我们从目标去反推，所以这个叫做 reverse。

10.2 Hierarchical RL



图 10.9

第三个方向是[分层强化学习 \(hierarchical reinforcement learning, HRL\)](#)。分层强化学习是说，我们有好几个 agent。然后，有一些 agent 负责比较 high level 的东西，它负责订目标，然后它订完目标以后，再分配给其他的 agent，去把它执行完成。

这样的想法其实也是很合理的。因为人在一生之中，并不是时时刻刻都在做决定。举例来说，假设你想要写一篇 paper，你会说就我先想个梗这样子，然后想完梗以后，你还要跑个实验。跑完实验以后，你还要写。写完以后呢，你还要这个去发表。每一个动作下面又还会再细分，比如说怎么跑实验呢？你要先 collect data，collect 完 data 以后，你要再 label，你要弄一个 network，然后又 train 不起来，要 train 很多次。然后重新 design network 架构好几次，最后才把 network train 起来。

所以，我们要完成一个很大的 task 的时候，我们并不是从非常底层的那些 action 开始想起，我们其实是有 plan。我们先想说，如果要完成这个最大的任务，那接下来要拆解成哪些小任务。每一个小任务要再怎么拆解成小小的任务。举例来说，叫你直接写一本书可能很困难，但叫你先把一本书拆成好几个章节，每个章节拆成好几段，每一段又拆成好几个句子，每一个句子又拆成好几个词汇，这样你可能就比较写得出来，这个就是分层的 reinforcement learning 的概念。

这边是举一个例子，就是假设校长、教授和研究生通通都是 agent。那今天假设我们只要进入百大就可以得到 reward。假设进入百大的话，校长就要提出愿景告诉其他的 agent 说，现在你要达到什么样的目标。那校长的愿景可能就是说教授每年都要发三篇期刊。然后接下来这些 agent 都是有分层的，所以上面的 agent，他的动作就是提出愿景这样。那他把他的愿景传给下一层的 agent，下一层的 agent 就把这个愿景吃下去。如果他下面还有其他人的话，它就会提出新的愿景。比如说，校长要教授发期刊，但其实教授自己也是不做实验的。所以，教授也只能叫下面的研究生做实验。所以教授就提出愿景，就做出实验的规划，然后研究生才是真的去执行这个实验的人。然后，真的把实验做出来，最后大家就可以得到 reward。那现在是这样子的，在 learn 的时候，其实每一个 agent 都会 learn。那他们的整体的目标就是要达到最后的 reward。那前面的这些 agent，他提出来的 actions 就是愿景这样。你如果是玩游戏的话，他提出来的就是，我现在想要产生这样的游戏画面。但是，假设他提出来的愿景是下面的 agent 达不到的，那就会被讨厌。举例来说，教授对研究生都一直逼迫研究生做一些很困难的实验，研究生都做不出来的话，研究生就会跑掉，所以他就会得到一个 penalty。所以如果今天下层的 agent 没办法达到上层 agent 所提出来的 goal 的话，上层的 agent 就会被讨厌，它就会得到一个 negative reward。所以他要避免提出那些愿景是底下的 agent 所做不到的。那每一个 agent 都是把上层的 agent 所提出来的愿景当作输入，然后决定他自己要产生什么输出。

但是你知道说，就算你看到上面的愿景说，叫你做这一件事情。你最后也不一定能做成这一件事情。假设本来教授目标是要写期刊，但是不知道怎么回事，他就要变成一个 YouTuber。这个 paper 里面的 solution，我觉得非常有趣。给大家做一个参考，这其实本来的目标是要写期刊，但却变成 YouTuber，那怎么办呢？把原来的愿景改成变成 YouTuber 就行了，在 paper 里面就是这么做的，为什么这么做呢？因为虽然本来的愿景是要写期刊，但是后来变成 YouTuber，难道这些动作都浪费了吗？不是，这些动作是没有被浪费的。我们就假设说，本来的愿景其实就是要成为 YouTuber，那你就知道成为 YouTuber 要怎做了。这个是分层 RL，是可以做得起来的 tip。

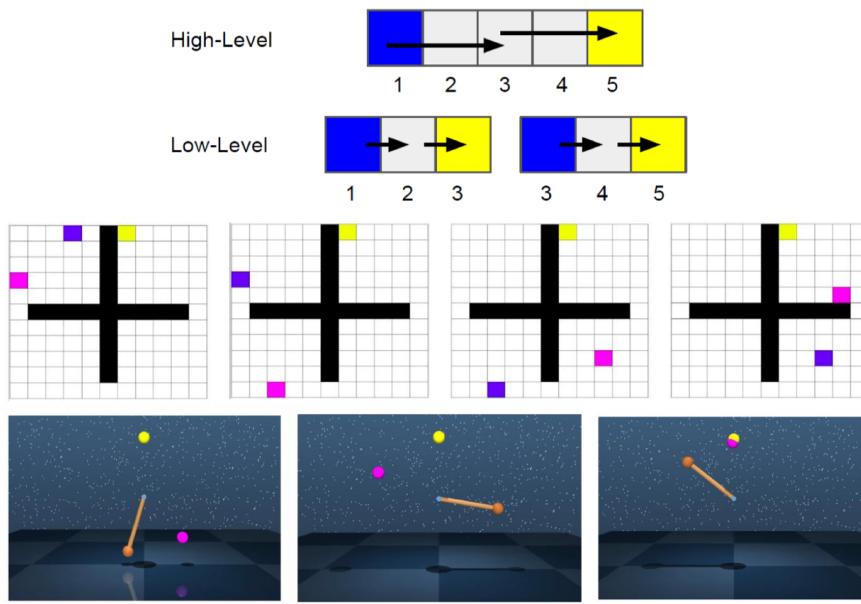


图 10.10

上图是真实的例子。实际上呢，这里面就做了一些比较简单的游戏，这个是走迷宫，蓝色是 agent，蓝色的 agent 要走到黄色的目标。这边也是，这个单摆要碰到黄色的球。那愿景是什么呢？

在这个 task 里面，它只有两个 agent，下层的一个 agent 负责决定说要怎么走，上层的 agent 就负

责提出愿景。虽然，实际上你可以用很多层，但 paper 就用了两层。

走迷宫的例子是说粉红色的这个点代表的就是愿景。上层这个 agent，它告诉蓝色的这个 agent 说，你现在的第一个目标是先走到这个地方，蓝色的 agent 走到以后，再说你的新的目标是走到这里。蓝色的 agent 再走到以后，新的目标在这里。接下来又跑到这边，最后希望蓝色的 agent 就可以走到黄色的位置。

单摆的例子也一样，就是粉红色的这个点代表的是上层的 agent 所提出来的愿景，所以这个 agent 先摆到这边，接下来，新的愿景又跑到这边，所以它又摆到这里。然后，新的愿景又跑到上面。然后又摆到上面，最后就走到黄色的位置了。这个就是 hierarchical 的 reinforcement learning。

最后总结下分层强化学习。分层强化学习是指将一个复杂的强化学习问题分解成多个小的、简单的子问题，每个子问题都可以单独用马尔可夫决策过程来建模。这样，我们可以将智能体的策略分为高层次策略和低层次策略，高层次策略根据当前状态决定如何执行低层次策略。这样，智能体就可以解决一些非常复杂的任务。

10.3 Keywords

- reward shaping: 在我们的 agent 与 environment 进行交互时，我们人为的设计一些 reward，从而“指挥”agent，告诉其采取哪一个 action 是最优的，而这个 reward 并不是 environment 对应的 reward，这样可以提高我们 estimate Q-function 时的准确性。
- ICM (intrinsic curiosity module): 其代表着 curiosity driven 这个技术中的增加新的 reward function 以后的 reward function。
- curriculum learning: 一种广义的用在 RL 的训练 agent 的方法，其在 input 训练数据的时候，采取由易到难的顺序进行 input，也就是认为设计它的学习过程，这个方法在 ML 和 DL 中都会普遍使用。
- reverse curriculum learning: 相较于上面的 curriculum learning，其为更 general 的方法。其从最终最理想的 state (我们称之为 gold state) 开始，依次去寻找距离 gold state 最近的 state 作为想让 agent 达到的阶段性的“理想”的 state，当然我们应该在此过程中有意的去掉一些极端的 case (太简单、太难的 case)。综上，reverse curriculum learning 是从 gold state 去反推，就是说你原来的目标是长这个样子，我们从我们的目标去反推，所以这个叫做 reverse curriculum learning。
- hierarchical (分层) reinforcement learning: 将一个大型的 task，横向或者纵向的拆解成多个 agent 去执行。其中，有一些 agent 负责比较 high level 的东西，负责订目标，然后它订完目标以后，再分配给其他的 agent 把它执行完成。(看教程的 hierarchical reinforcement learning 部分的示例就会比较明了)

10.4 Questions

- 解决 sparse reward 的方法有哪些？

答：Reward Shaping、curiosity driven reward、(reverse) curriculum learning 、Hierarchical Reinforcement learning 等等。

- reward shaping 方法存在什么主要问题？

答：主要的一个问题是人为设计的 reward 需要 domain knowledge，需要我们自己设计出符合 environment 与 agent 更好的交互的 reward，这需要不少的经验知识，需要我们根据实际的效果进行调整。

- ICM 是什么？我们应该如何设计这个 ICM？

答：ICM 全称为 intrinsic curiosity module。其代表着 curiosity driven 这个技术中的增加新的 reward function 以后的 reward function。具体来说，ICM 在更新计算时会考虑三个新的东西，分别是 state s_1 、action a_1 和 state s_2 。根据 s_1 、 a_1 、 a_2 ，它会 output 另外一个新的 reward r_1^i 。所以在 ICM 中

我们 total reward 并不是只有 r 而已，还有 r^i 。它不是只有把所有的 r 都加起来，它还把所有 r^i 加起来当作 total reward。所以，它在跟环境互动的时候，它不是只希望 r 越大越好，它还同时希望 r^i 越大越好，它希望从 ICM 的 module 里面得到的 reward 越大越好。ICM 就代表了一种 curiosity。对于如何设计 ICM，ICM 的 input 就像前面所说的一样包括三部分 input 现在的 state s_1 ，input 在这个 state 采取的 action a_1 ，然后接 input 下一个 state s_{t+1} ，对应的 output 就是 reward r_1^i ，input 到 output 的映射是通过 network 构建的，其使用 s_1 和 a_1 去预测 \hat{s}_{t+1} ，然后继续评判预测的 \hat{s}_{t+1} 和真实的 s_{t+1} 像不像，越不相同得到的 reward 就越大。通俗来说这个 reward 就是，如果未来的状态越难被预测的话，那么得到的 reward 就越大。这也就是 curiosity 的机制，倾向于让 agent 做一些风险比较大的 action，从而增加其 machine exploration 的能力。

同时为了进一步增强 network 的表达能力，我们通常讲 ICM 的 input 优化为 feature extractor，这个 feature extractor 模型的 input 就是 state，output 是一个特征向量，其可以表示这个 state 最主要、重要的特征，把没有意义的东西过滤掉。

References

- 神经网络与深度学习

第 11 章 Imitation Learning

Introduction

- Imitation Learning
 - Also known as learning from demonstration, apprenticeship learning
- An expert demonstrates how to solve the task
 - Machine can also interact with the environment, but cannot explicitly obtain reward.
 - It is hard to define reward in some tasks.
 - Hand-crafted rewards can lead to uncontrolled behavior
- Two approaches:
 - Behavior Cloning
 - Inverse Reinforcement Learning (inverse optimal control)

图 11.1

Imitation learning 讨论的问题是：假设我们连 reward 都没有，那要怎么办呢？Imitation learning 又叫做 learning from demonstration(示范学习), apprenticeship learning(学徒学习), learning by watching(观察学习)。在 Imitation learning 里面，你有一些 expert 的 demonstration，那 machine 也可以跟环境互动，但它没有办法从环境里面得到任何的 reward，它只能看着 expert 的 demonstration 来学习什么是好，什么是不好。其实，多数的情况，我们都无法从环境里面得到非常明确的 reward。举例来说，如果是棋类游戏或者是电玩，你有非常明确的 reward。但是其实多数的任务，都是没有 reward 的。以 chat-bot 为例，机器跟人聊天，聊得怎么样算是好，聊得怎么样算是不好，你无法给出明确的 reward。所以很多 task 是根本就没有办法给出 reward 的。

虽然没有办法给出 reward，但是收集 expert 的 demonstration 是可以做到的。举例来说，

- 在自动驾驶汽车里面，虽然你没有办法给出自动驾驶汽车的 reward，但你可以收集很多人类开车的纪录。
- 在 chat-bot 里面，你可能没有办法定义什么叫做好的对话，什么叫做不好的对话。但是收集很多人的对话当作范例，这一件事情也是可行的。

所以 imitation learning 的使用性非常高。假设你不知道该怎么定义 reward，你就可以收集到 expert 的 demonstration。如果你可以收集到一些范例的话，你可以收集到一些很厉害的 agent(比如人) 跟环境实际上的互动的话，那你就可以考虑 imitation learning 这个技术。在 imitation learning 里面，我们介绍两个方法。第一个叫做 Behavior Cloning，第二个叫做 Inverse Reinforcement Learning 或者叫做 Inverse Optimal Control。

11.1 Behavior Cloning

其实 Behavior Cloning 跟 supervised learning 是一模一样的。如图 11.2 所示，以自动驾驶汽车为例，你可以收集到人开自动驾驶汽车的所有资料，比如说可以通过行车记录器进行收集。看到这样子的 observation 的时候，人会决定向前。机器就采取跟人一样的行为，也向前，就结束了。这个就叫做 Behavior Cloning，Expert 做什么，机器就做一模一样的事。

怎么让机器学会跟 expert 一模一样的行为呢？就把它当作一个 supervised learning 的问题，你去收集很多行车记录器，然后再收集人在那个情境下会采取什么样的行为。你知道说人在 state s_1 会采取 action a_1 ，人在 state s_2 会采取 action a_2 。人在 state s_3 会采取 action a_3 。接下来，你就 learn 一个 network。这个 network 就是你的 actor，它 input s_i 的时候，你就希望它的 output 是 a_i ，就这样结束了。它就是一个的 supervised learning 的 problem。

Behavior Cloning

Yes, this is
supervised learning.

- Self-driving cars as example

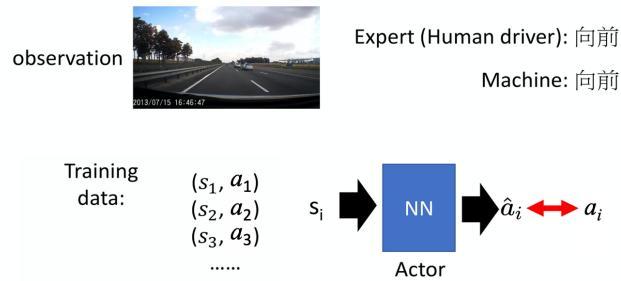


图 11.2

- Problem

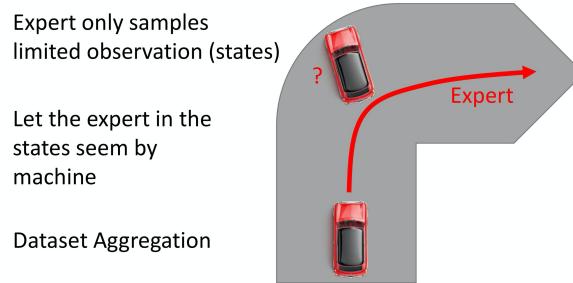


图 11.3

Behavior Cloning 虽然非常简单，但它的问题是如果你只收集 expert 的资料，你可能看过的 observation 会是非常有限的。

举例来说，如图 11.3 所示，

- 假设你要 learn 一部自动驾驶汽车，自动驾驶汽车就是要过这个弯道。如果是 expert 的话，它就是把车顺着这个红线就开过去了。但假设你的 agent 很笨，它今天开着开着，就开到撞墙了，它永远不知道撞墙这种状况要怎么处理，为什么？因为 training data 里面从来没有撞过墙，所以它根本就不知道撞墙这一种 case 要怎么处理。
- 打电玩也是一样，让人去玩 Mario，那 expert 可能非常强，它从来不会跳不上水管，所以机器根本不知道跳不上水管时要怎么处理。

所以光是做 Behavior Cloning 是不够的，只观察 expert 的行为是不够的，需要一个招数，这个招数叫作 Dataset Aggregation。

我们会希望收集更多样性的 data，而不是只收集 expert 所看到的 observation。我们会希望能够收集 expert 在各种极端的情况下，它会采取什么样的行为。如图 11.4 所示，以自动驾驶汽车为例的话，假设一开始，你的 actor 叫作 π_1 ，你让 π_1 去开这个车。但车上坐了一个 expert。这个 expert 会不断地告诉 machine 说，如果在这个情境里面，我会怎么样开。所以 π_1 自己开自己的，但是 expert 会不断地表示它的想法。比如说，如上图所示，一开始的时候，expert 可能说往前走。在拐弯的时候，expert 可能就会说往右转。但 π_1 是不管 expert 的指令的，所以它会继续去撞墙。虽然 expert 说往右转，但是不管它怎么下指令都是没有用的， π_1 会自己做自己的事情，因为我们要做的记录的是说，今天 expert 在 π_1 看到这种 observation 的情况下，它会做什么样的反应。这个方法显然是有一些问题的，因为你每开一次自动驾

- Dataset Aggregation
 - Get actor π_1 by behavior cloning
 - Using π_1 to interact with the environment
 - Ask the expert to label the observation of π_1
 - Using new data to train π_2

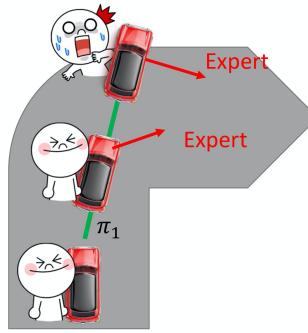
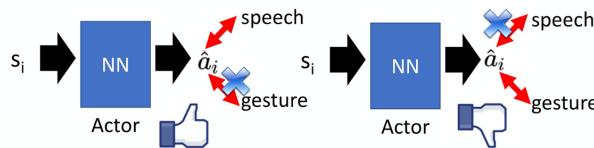


图 11.4

驶汽车就会牺牲一个人。那你用这个方法，你牺牲一个 expert 以后，你就会知道，人类在这样子的 state 下，在快要撞墙的时候，会采取什么样的行为。再把这个 data 拿去 train 新的 π_2 。这个 process 就反复继续下去，这个方法就叫做 Dataset Aggregation。

Behavior Cloning

- Major problem: if machine has limited capacity, it may choose the wrong behavior to copy.



- Some behavior must copy, but some can be ignored.
 - Supervised learning takes all errors equally

图 11.5

Behavior Cloning 还有一个问题：机器会完全 copy expert 的行为，不管 expert 的行为是否有道理，就算没有道理，没有什么用的，就算这是 expert 本身的习惯，机器也会硬把它记下来。如果机器确实可以记住所有 expert 的行为，那也许还好，为什么呢？因为如果 expert 这么做，有些行为是多余的。但是没有问题，假设机器的行为可以完全仿造 expert 行为，那也就算了，那它是跟 expert 一样得好，只是做一些多余的事。但问题是它是一个 machine，它是一个 network，network 的 capacity 是有限的。就算给 network training data，它在 training data 上得到的正确率往往也不是 100%，它有些事情是学不起来的。这个时候，什么该学，什么不该学就变得很重要。

举例来说，在学习中文的时候，你的老师，它有语音，它也有行为，它也有知识，但其实只有语音部分是重要的，知识的部分是不重要的。也许 machine 只能够学一件事，也许它就只学到了语音，那没有问题。如果它只学到了手势，这样子就有问题了。所以让机器学习什么东西是需要 copy，什么东西是不需要 copy，这件事情是重要的。而单纯的 Behavior Cloning 就没有把这件事情学进来，因为机器只是复制 expert 所有的行为而已，它不知道哪些行为是重要，是对接下来有影响的，哪些行为是不重要的，是对接下来是没有影响的。

Behavior Cloning 还有什么样的问题呢？在做 Behavior Cloning 的时候，training data 跟 testing data 是 mismatch 的。我们可以用 Dataset Aggregation 的方法来缓解这个问题。这个问题是：在 training 跟 testing 的时候，data distribution 其实是不一样的。因为在 reinforcement learning 里面，action 会影响到



- In supervised learning, we expect training and testing data have the same distribution.
- In behavior cloning:
 - Training: $(s, a) \sim \hat{\pi}$ (expert)
 - Action a taken by actor influences the distribution of s
 - Testing: $(s', a') \sim \pi^*$ (actor cloning expert)
 - If $\hat{\pi} = \pi^*$, (s, a) and (s', a') from the same distribution
 - If $\hat{\pi}$ and π^* have difference, the distribution of s and s' can be very different.

图 11.6

接下来所看到的 state。我们是先有 state s_1 , 然后采取 action a_1 , action a_1 其实会决定接下来你看到什么样的 state s_2 。所以在 reinforcement learning 里面有一个很重要的特征, 就是你采取了 action 会影响你接下来所看到的 state。如果做了 Behavior Cloning 的话, 我们只能观察到 expert 的一堆 state 跟 action 的 pair。然后我们希望可以 learn 一个 π^* , 我们希望 π^* 跟 $\hat{\pi}$ 越接近越好。如果 π^* 可以跟 $\hat{\pi}$ 一模一样的话, training 的时候看到的 state 跟 testing 的时候所看到的 state 会是一样的。因为虽然 action 会影响我们看到的 state, 但假设两个 policy 一模一样, 在同一个 state 都会采取同样的 action, 那你接下来所看到的 state 都会是一样的。但问题是就是你很难让你的 learn 出来的 policy 跟 expert 的 policy 一模一样。Expert 可是一个人, network 要跟人一模一样, 感觉很难吧。

如果你的 π^* 跟 $\hat{\pi}$ 有一点误差。这个误差在一般 supervised learning problem 里面, 每一个 example 都是 independent 的, 也许还好。但对 reinforcement learning 的 problem 来说, 可能在某个地方就是失之毫厘, 差之千里。可能在某个地方, 也许 machine 没有办法完全复制 expert 的行为, 它复制的差了一点点, 也许最后得到的结果就会差很多这样。所以 Behavior Cloning 并不能够完全解决 imitation learning 这件事情。所以就有另外一个比较好的做法叫做 [Inverse Reinforcement Learning](#)。

11.2 Inverse RL

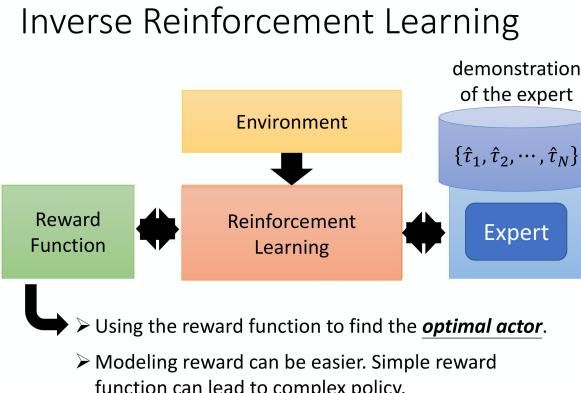


图 11.7

为什么叫 Inverse Reinforcement Learning，因为原来的 Reinforcement Learning 里面，有一个环境和一个 reward function。根据环境和 reward function，通过 Reinforcement Learning 这个技术，你会找到一个 actor，你会 learn 出一个 optimal actor。但 Inverse Reinforcement Learning 刚好是相反的，你没有 reward function，你只有一堆 expert 的 demonstration。但你还是有环境的。IRL 的做法是说假设我们现在有一堆 expert 的 demonstration，我们用 $\hat{\pi}$ 来代表 expert 的 demonstration。如果是在玩电玩的话，每一个 τ 就是一个很会玩电玩的人玩一场游戏的纪录，如果是自动驾驶汽车的话，就是人开自动驾驶汽车的纪录。这一边就是 expert 的 demonstration，每一个 τ 是一个 trajectory。

把所有 expert demonstration 收集起来，然后，使用 Inverse Reinforcement Learning 这个技术。使用 Inverse Reinforcement Learning 技术的时候，机器是可以跟环境互动的。但它得不到 reward。它的 reward 必须要从 expert 那边推出，有了环境和 expert demonstration 以后，去反推出 reward function 长什么样子。之前 reinforcement learning 是由 reward function 反推出什么样的 action、actor 是最好的。Inverse Reinforcement Learning 是反过来，我们有 expert 的 demonstration，我们相信它是不错的，我就反推说，expert 是因为什么样的 reward function 才会采取这些行为。你有了 reward function 以后，接下来，你就可以套用一般的 reinforcement learning 的方法去找出 optimal actor。所以 Inverse Reinforcement Learning 是先找出 reward function，找出 reward function 以后，再去用 Reinforcement Learning 找出 optimal actor。

把这个 reward function learn 出来，相较于原来的 Reinforcement Learning 有什么样好处。一个可能的好处是也许 reward function 是比较简单的。也许，虽然这个 expert 的行为非常复杂，但也许简单的 reward function 就可以导致非常复杂的行为。一个例子就是也许人类本身的 reward function 就只有活着这样，每多活一秒，你就加一分。但人类有非常复杂的行为，但是这些复杂的行为，都只是围绕着要从这个 reward function 里面得到分数而已。有时候很简单的 reward function 也许可以推导出非常复杂的行为。

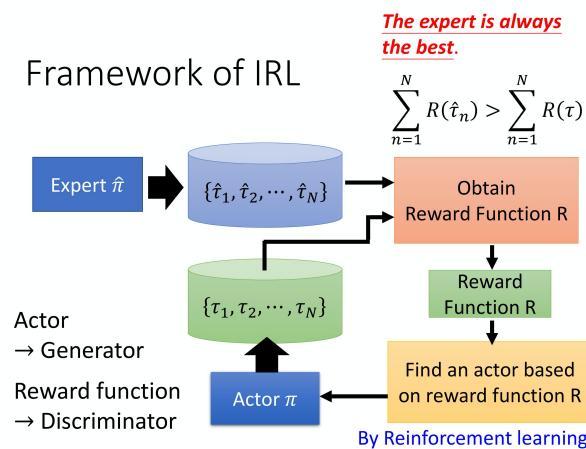


图 11.8

Inverse Reinforcement Learning 实际上是怎么做的呢？首先，我们有一个 expert $\hat{\pi}$ ，这个 expert 去跟环境互动，给我们很多 $\hat{\tau}_1$ 到 $\hat{\tau}_n$ 。如果是玩游戏的话，就让某一个电玩高手，去玩 n 场游戏。把 n 场游戏的 state 跟 action 的 sequence 都记录下来。接下来，你有一个 actor π ，一开始 actor 很烂，这个 actor 也去跟环境互动。它也去玩了 n 场游戏，它也有 n 场游戏的纪录。接下来，我们要反推出 reward function。怎么推出 reward function 呢？原则就是 expert 永远是最棒的，是先射箭，再画靶的概念。

Expert 去玩一玩游戏，得到这一些游戏的纪录，你的 actor 也去玩一玩游戏，得到这些游戏的纪录。接下来，你要定一个 reward function，这个 reward function 的原则就是 expert 得到的分数要比 actor 得到的分数高，先射箭，再画靶。所以我们就 learn 出一个 reward function。你就找出一个 reward function。这个 reward function 会使 expert 所得到的 reward 大于 actor 所得到的 reward。你有了新的 reward

function 以后，就可以套用一般 Reinforcement Learning 的方法去 learn 一个 actor，这个 actor 会针对 reward function 去 maximize 它的 reward。它也会采取一大堆的 action。但是，今天这个 actor 虽然可以 maximize 这个 reward function，采取一大堆的行为，得到一大堆游戏的纪录。

但接下来，我们就改 reward function。这个 actor 就会很生气，它已经可以在这个 reward function 得到高分。但是它得到高分以后，我们就改 reward function，仍然让 expert 可以得到比 actor 更高的分数。这个就是 [Inverse Reinforcement learning](#)。有了新的 reward function 以后，根据这个新的 reward function，你就可以得到新的 actor，新的 actor 再去跟环境做一下互动，它跟环境做互动以后，你又会重新定义你的 reward function，让 expert 得到的 reward 比 actor 大。

怎么让 expert 得到的 reward 大过 actor 呢？其实你在 learning 的时候，你可以很简单地做一件事就是，reward function 也许就是 neural network。这个 neural network 就是吃一个 τ ，output 就是应该要给这个 τ 多少的分数。或者说，你假设觉得 input 整个 τ 太难了。因为 τ 是 s 和 a 的一个很强的 sequence。也许它就是 input 一个 s 和 a 的 pair，然后 output 一个 real number。把整个 sequence，整个 τ 会得到的 real number 都加起来就得到 $R(\tau)$ 。在 training 的时候，对于 $\{\hat{\tau}_1, \hat{\tau}_2, \dots, \hat{\tau}_N\}$ ，我们希望它 output 的 R 越大越好。对于 $\{\tau_1, \tau_2, \dots, \tau_N\}$ ，我们就希望它 R 的值越小越好。

什么叫做一个最好的 reward function。最后你 learn 出来的 reward function 应该就是 expert 和 actor 在这个 reward function 都会得到一样高的分数。最终你的 reward function 没有办法分辨出谁应该会得到比较高的分数。

通常在 train 的时候，你会 iterative 的去做。那今天的状况是这样，最早的 Inverse Reinforcement Learning 对 reward function 有些限制，它是假设 reward function 是 linear 的。如果 reward function 是 linear 的话，你可以 prove 这个 algorithm 会 converge。但是如果不是 linear 的，你就没有办法 prove 说它会 converge。你有没有觉得这个东西，看起来还挺熟悉呢？其实你只要把它换个名字，说 actor 就是 generator，然后说 reward function 就是 discriminator，它就是 GAN。所以它会不会收敛这个问题就等于是问说 GAN 会不会收敛。如果你已经实现过，你会知道不一定会收敛。但除非你对 R 下一个非常严格的限制，如果你的 R 是一个 general 的 network 的话，你就会有很大的麻烦。

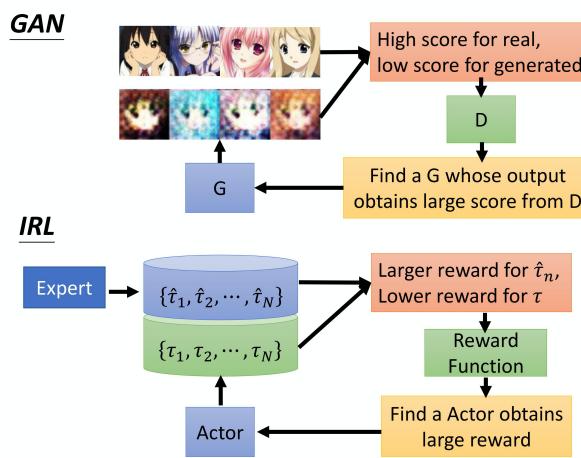


图 11.9

那怎么说它像是一个 GAN，我们来跟 GAN 比较一下。GAN 里面，你有一堆很好的图。然后你有一个 generator，一开始它根本不知道要产生什么样的图，它就乱画。然后你有一个 discriminator，discriminator 的工作就是给画的图打分，expert 画的图就是高分，generator 画的图就是低分。你有 discriminator 以后，generator 会想办法去骗过 discriminator。Generator 会希望 discriminator 也会给它画的图高分。整个 process 跟 Inverse Reinforcement Learning 是一模一样的。

- 画的图就是 expert 的 demonstration。generator 就是 actor，generator 画很多图，actor 会去跟环境互动，产生很多 trajectory。这些 trajectory 跟环境互动的记录，游戏的纪录其实就是等于 GAN 里

面的这些图。

- 然后你 learn 一个 reward function。Reward function 就是 discriminator。Rewards function 要给 expert 的 demonstration 高分，给 actor 互动的结果低分。
- 接下来，actor 会想办法，从这个已经 learn 出来的 reward function 里面得到高分，然后 iterative 地去循环。跟 GAN 其实是一模一样的，我们只是换个说法来而已。

Parking Lot Navigation



- Reward function:
 - Forward vs. reverse driving
 - Amount of switching between forward and reverse
 - Lane keeping
 - On-road vs. off-road
 - Curvature of paths

图 11.10

IRL 有很多的 application，比如可以用开来自动驾驶汽车，有人用这个技术来学开自动驾驶汽车的不同风格。每个人在开车的时候会有不同风格，举例来说，能不能够压到线，能不能够倒退，要不要遵守交通规则等等。每个人的风格是不同的，然后用 Inverse Reinforcement Learning 可以让自动驾驶汽车学会各种不同的开车风格。

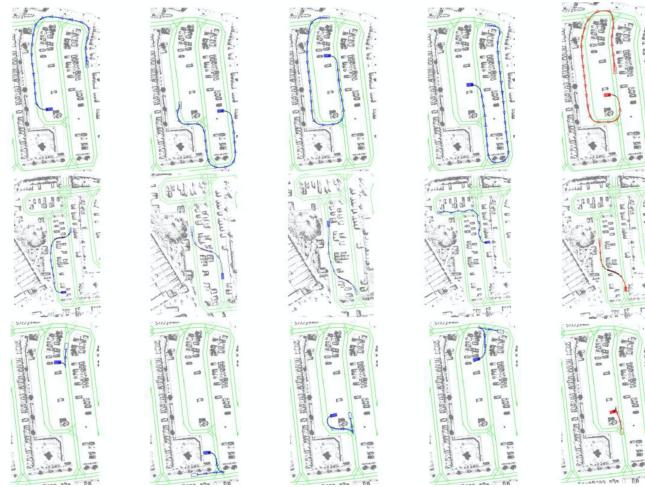


图 11.11

上图是文献上真实的例子。在这个例子里面，Inverse Reinforcement Learning 有一个有趣的地方，通常你不需要太多的 training data，因为 training data 往往都是个位数。因为 Inverse Reinforcement Learning 只是一种 demonstration，只是一种范例，实际上机器可以去跟环境互动非常多次。所以在 Inverse Reinforcement Learning 的文献，往往能看到说只用几笔 data 就训练出一些有趣的结果。

比如说，在这个例子里面是要让自动驾驶汽车学会在停车场里面停。这边的 demonstration 是这样，蓝色是终点，自动驾驶汽车要开到蓝色终点停车。给机器只看一行的四个 demonstration，然后让它去学怎么样开车，最后它就可以学出，在红色的终点位置，如果它要停车的话，它会这样开。今天给机器看不同的 demonstration，最后它学出来开车的风格就会不太一样。举例来说，上图第二行是不守规矩的开车

方式，因为它会开到道路之外，这边，它会穿过其它的车，然后从这边开进去。所以机器就会学到说，不一定要走在道路上，它可以走非道路的地方。上图第三行是倒退来停车，机器也会学会说，它可以倒退，



图 11.12

这种技术也可以拿来训练机器人。你可以让机器人，做一些你想要它做的动作，过去如果你要训练机器人，做你想要它做的动作，其实是比较麻烦的。怎么麻烦呢？过去如果你要操控机器的手臂，你要花很多力气去写 program 才让机器做一件很简单的事看。假设你有 Imitation Learning 的技术，你可以让人做一下示范，然后机器就跟着人的示范来进行学习，比如学会摆盘子，拉着机器人的手去摆盘子，机器自己动。让机器学会倒水，人只教它 20 次，杯子每次放的位置不太一样。用这种方法来教机械手臂。

11.3 Third Person Imitation Learning

Third Person Imitation Learning

- Ref: Bradly C. Stadie, Pieter Abbeel, Illya Sutskever, "Third-Person Imitation Learning", arXiv preprint, 2017



图 11.13

其实还有很多相关的研究，举例来说，你在教机械手臂的时候，要注意就是也许机器看到的视野跟人看到的视野是不太一样的。在刚才那个例子里面，人跟机器的动作是一样的。但是在未来的世界里面，也许机器是看着人的行为学的。刚才是人拉着，假设你要让机器学会打高尔夫球，在刚才的例子里面就是人拉着机器人手臂去打高尔夫球，但是在未来有没有可能机器就是看着人打高尔夫球，它自己就学会打高尔夫球了呢？但这个时候，要注意的事情是机器的视野跟它真正去采取这个行为的时候的视野是不一样的。机器必须了解到当它是第三人的视角的时候，看到另外一个人在打高尔夫球，跟它实际上自己去打高尔夫球的时候，看到的视野显然是不一样的。但它怎么把它是第三人的时间所观察到的经验把它 generalize 到它是第一人称视角的时候所采取的行为，这就需要用到 [Third Person Imitation Learning](#) 的技术。

Third Person Imitation Learning

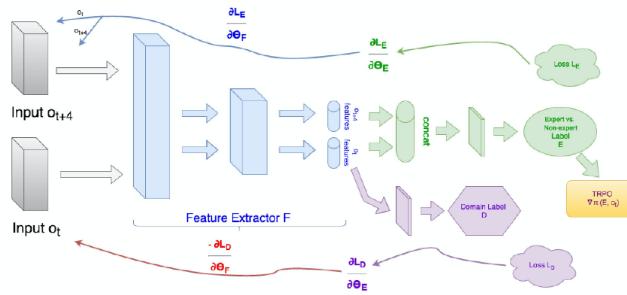


图 11.14

这个怎么做呢？它的技术其实也是不只是用到 Imitation Learning，它用到了 Domain-Adversarial Training。我们在讲 Domain-Adversarial Training 的时候，我们有讲说这也是一个 GAN 的技术。那我们希望今天有一个 extractor，有两个不同 domain 的 image，通过 feature extractor 以后，没有办法分辨出它来自哪一个 domain。其实第一人称视角和第三人称视角，Imitation Learning 用的技术其实也是一样的，希望 learn 一个 Feature Extractor，机器在第三人称的时候跟它在第一人称的时候看到的视野其实是一样的，就是把最重要的东西抽出来就好了。

11.4 Recap: Sentence Generation & Chat-bot

Recap: Sentence Generation & Chat-bot

<u>Sentence Generation</u>	<u>Chat-bot</u>
Expert trajectory:	Expert trajectory:
床前明月光	input: how are you
(s_1, a_1): ("<BOS>", "床")	(s_1, a_1): ("input, <BOS>", "I")
(s_2, a_2): ("床", "前")	(s_2, a_2): ("input, I", "am")
(s_3, a_3): ("床前", "明")	(s_3, a_3): ("input, I am", "fine")
⋮	⋮

Maximum likelihood is behavior cloning. Now we have better approach like SeqGAN.

图 11.15

在讲 Sequence GAN 的时候，我们有讲过 Sentence Generation 跟 Chat-bot。那其实 Sentence Generation 或 Chat-bot 也可以想成是 imitation learning。机器在 imitate 人写的句子，你在写句子的时候，你写下去的每一个 word 都想成是一个 action，所有的 word 合起来就是一个 episode。举例来说，sentence generation 里面，你会给机器看很多人类写的文字。你要让机器学会写诗，那你就要给它看唐诗三百首。人类写的文字其实就是 expert 的 demonstration。每一个词汇其实就是一个 action。你让机器做 Sentence Generation 的时候，其实就是在 imitate expert 的 trajectory。Chat-bot 也是一样，在 Chat-bot 里面你会收集到很多人互动对话的纪录，那些就是 expert 的 demonstration。

如果我们单纯用 maximum likelihood 这个技术来 maximize 会得到 likelihood，这个其实就是 behavior

cloning。我们做 behavior cloning 就是看到一个 state，接下来预测我们会得到什么样的 action。看到一个 state，然后有一个 ground truth 告诉机器说什么样的 action 是最好的。在做 likelihood 的时候也是一样，given sentence 已经产生的部分。接下来 machine 要 predict 说接下来要写哪一个 word 才是最好的。所以，其实 maximum likelihood 在做 sequence generation 的时候，它对应到 imitation learning 里面就是 behavior cloning。只有 maximum likelihood 是不够的，我们想要用 Sequence GAN。其实 Sequence GAN 就是对应到 Inverse Reinforcement Learning，Inverse Reinforcement Learning 就是一种 GAN 的技术。你把 Inverse Reinforcement Learning 的技术放在 sentence generation，放到 Chat-bot 里面，其实就是 Sequence GAN 跟它的种种的变形。

11.5 Keywords

- Imitation learning: 其讨论我们没有 reward 或者无法定义 reward 但是有与 environment 进行交互时怎么进行 agent 的学习。这与我们平时处理的问题中的情况有些类似，因为通常我们无法从环境中得到明确的 reward。Imitation learning 又被称为 learning from demonstration (示范学习) , apprenticeship learning (学徒学习), learning by watching (观察学习) 等。
- Behavior Cloning: 类似于 ML 中的监督学习，通过收集 expert 的 state 与 action 的对应信息，训练我们的 network (actor)。在使用时 input state 时，得到对应的 output action。
- Dataset Aggregation: 用来应对在 Behavior Cloning 中 expert 提供不到的 data，其希望收集 expert 在各种极端 state 下 expert 的 action。
- Inverse Reinforcement learning (IRL): Inverse Reinforcement Learning 是先找出 reward function，再去用 Reinforcement Learning 找出 optimal actor。这么做是因为我们没有环境中 reward，但是我们有 expert 的 demonstration，使用 IRL，我们可以推断 expert 是因为什么样的 reward function 才会采取这些 action。有了 reward function 以后，接下来，就可以套用一般的 reinforcement learning 的方法去找出 optimal actor。
- Third Person Imitation Learning: 一种把第三人称视角所观察到的经验 generalize 到第一人称视角的经验的技术。

11.6 Questions

- 对于 Imitation Learning 的方法有哪些？

答：Behavior Cloning、Inverse Reinforcement Learning (IRL) 或者称为 Inverse Optimal Control。

- Behavior Cloning 存在哪些问题呢？我们可以如何处理呢？

答：

1. 首先，如果只收集 expert 的 data (看到某一个 state 输出的 action)，你可能看过的 observation 会是非常 limited。所以我们要收集 expert 在各种极端 state 下的 action，或者说是要收集更多的、复杂的 data，可以使用教程中提到的 Dataset Aggregation。

2. 另外，使用传统意义上的 Behavior Cloning 的话，机器会完全 copy expert 的行为，不管 expert 的行为是否有道理，就算没有道理，没有什么用的，这是 expert 本身的习惯，机器也会硬把它记下来。我们的 agent 是一个 machine，它是一个 network，network 的 capacity 是有限的。就算给 network training data，它在 training data 上得到的正确率往往也不是 100%，他有些事情是学不起来的。这个时候，什么该学，什么不该学就变得很重要。不过极少数 expert 的行为是没有意义的，但是至少也不会产生较坏的影响。

3. 还有，在做 Behavior Cloning 的时候，training data 跟 testing data 是 mismatch 的。我们可以用 Dataset Aggregation 的方法来缓解这个问题。这个问题是，在 training 跟 testing 的时候，data distribution 其实是不一样的。因为在 reinforcement learning 里面，action 会影响到接下来所看到的 state。我们是先有 state s_1 ，然后采取 action a_1 ，action a_1 其实会决定接下来你看到什么样的 state

s_2 。所以在 reinforcement learning 里面有一个很重要的特征，就是你采取了 action 会影响你接下来所看到的 state。如果做了 Behavior Cloning 的话，我们只能观察到 expert 的一堆 state 跟 action 的 pair。然后我们希望可以 learn 一个 π^* ，我们希望 π^* 跟 $\hat{\pi}$ 越接近越好。如果 π^* 可以跟 $\hat{\pi}$ 一模一样的话，你 training 的时候看到的 state 跟 testing 的时候所看到的 state 会是一样的，这样模型的泛化性能就会变得比较差。而且，如果你的 π^* 跟 $\hat{\pi}$ 有一点误差。这个误差在一般 supervised learning problem 里面，每一个 example 都是 independent 的，也许还好。但对 reinforcement learning 的 problem 来说，可能在某个地方，也许 machine 没有办法完全复制 expert 的行为，也许最后得到的结果就会差很多。所以 Behavior Cloning 并不能够完全解决 Imatation learning 这件事情，我们可以使用另外一个比较好的做法叫做 Inverse Reinforcement Learning。

- Inverse Reinforcement Learning 是怎么运行的呢？

答：首先，我们有一个 expert $\hat{\pi}$ ，这个 expert 去跟环境互动，给我们很多 $\hat{\tau}_1$ 到 $\hat{\tau}_n$ ，我们需要将其中的 state、action 这个序列都记录下来。然后对于 actor π 也需要进行一样的互动和序列的记录。接着我们需要指定一个 reward function，并且保证 expert 对应的分数一定要比 actor 的要高，用过这个 reward function 继续 learning 更新我们的训练并且套用一般条件下的 RL 方法进行 actor 的更新。在这个过程中，我们也要同时进行我们一开始制定的 reward function 的更新，使得 actor 得得分越来越高，但是不超过 expert 的得分。最终的 reward function 应该让 expert 和 actor 对应的 reward function 都达到比较高的分数，并且从最终的 reward function 中无法分辨出谁应该得到比较高的分数。

- Inverse Reinforcement Learning 方法与 GAN 在图像生成中有什么异曲同工之处？

答：在 GAN 中，我们有一些比较好的图片数据集，也有一个 generator，一开始他根本不知道要产生什么样的图，只能随机生成。另外我们有一个 discriminator，其用来给生成的图打分，expert 生成的图得分高，generator 生成的图得分低。有了 discriminator 以后，generator 会想办法去骗过 discriminator。Generator 会希望 discriminator 也会给它生成得图高分。整个 process 跟 IRL 的过程是类似的。我们一一对应起来看：

- 生成的图就是 expert 的 demonstration，generator 就是 actor，generator 会生成很多的图并让 actor 与环境进行互动，从而产生很多 trajectory。这些 trajectory 跟环境互动的记录等价于 GAN 里面的生成图。
- 在 IRL 中 learn 的 reward function 就是 discriminator。Rewards function 要给 expert 的 demonstration 高分，给 actor 互动的结果低分。
- 考虑两者的过程，在 IRL 中，actor 会想办法，从这个已经 learn 出来的 reward function 里面得到高分，然后 iterative 地去循环这其实是与 GAN 的过程是一致的。

References

- 机器学习

第 12 章 DDPG

12.1 Discrete Action vs. Continuous Action

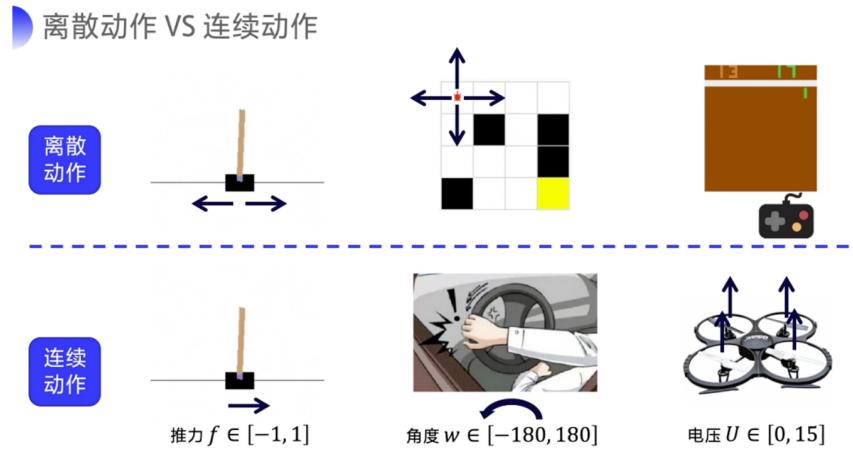


图 12.1

离散动作与连续动作是相对的概念，一个是可数的，一个是不可数的。

离散动作有如下几个例子：

- 在 CartPole 环境中，可以有向左推小车、向右推小车两个动作。
- 在 Frozen Lake 环境中，小乌龟可以有上下左右四个动作。
- 在 Atari 的 Pong 游戏中，游戏有 6 个按键的动作可以输出。

但在实际情况中，经常会遇到连续动作空间的情况，也就是输出的动作是不可数的。比如：

- 推小车力的大小，
- 选择下一时刻方向盘的转动角度，
- 四轴飞行器的四个螺旋桨给的电压的大小。

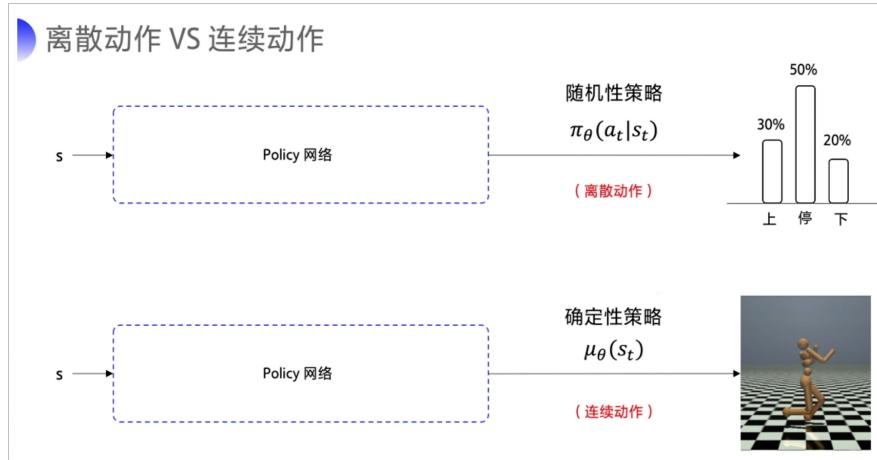


图 12.2

对于这些连续的动作控制空间，Q-learning、DQN 等算法是没有办法处理的。那我们怎么输出连续的动作呢，这个时候，万能的神经网络又出现了。

在离散动作的场景下，比如说我输出上、下或是停止这几个动作。有几个动作，神经网络就输出几个概率值，我们用 $\pi_\theta(a_t|s_t)$ 来表示这个随机性的策略。

在连续的动作场景下，比如说我要输出这个机器人手臂弯曲的角度，这样子的一个动作，我们就输出一个具体的浮点数。我们用 $\mu_\theta(s_t)$ 来代表这个确定性的策略。

我们再解释一下随机性策略跟确定性策略。

对随机性的策略来说，输入某一个状态 s ，采取某一个 action 的可能性并不是百分之百，而是有一个概率 P 的，就好像抽奖一样，根据概率随机抽取一个动作。

而对于确定性的策略来说，它没有概率的影响。当神经网络的参数固定下来了之后，输入同样的 state，必然输出同样的 action，这就是确定性的策略。

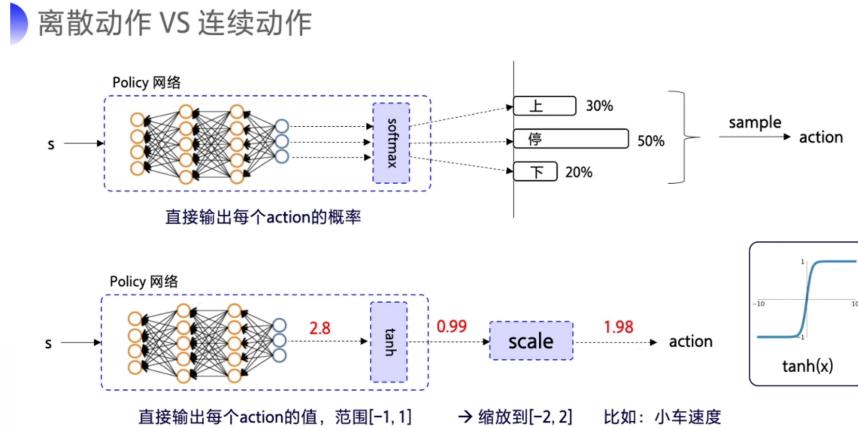


图 12.3

- 要输出离散动作的话，我们就是加一层 softmax 层来确保说所有的输出是动作概率，而且所有的动作概率加和为 1。
- 要输出连续动作的话，一般可以在输出层这里加一层 tanh。
 - tanh 的图像的像右边这样子，它的作用就是把输出限制到 $[-1, 1]$ 之间。
 - 我们拿到这个输出后，就可以根据实际动作的范围再做一下缩放，然后再输出给环境。
 - 比如神经网络输出一个浮点数是 2.8，然后经过 tanh 之后，它就可以被限制在 $[-1, 1]$ 之间，它输出 0.99。假设小车速度的动作范围是 $[-2, 2]$ 之间，那我们就按比例从 $[-1, 1]$ 扩放到 $[-2, 2]$ ，0.99 乘 2，最终输出的就是 1.98，作为小车的速度或者说推小车的力输出给环境。

12.2 DDPG(Deep Deterministic Policy Gradient)

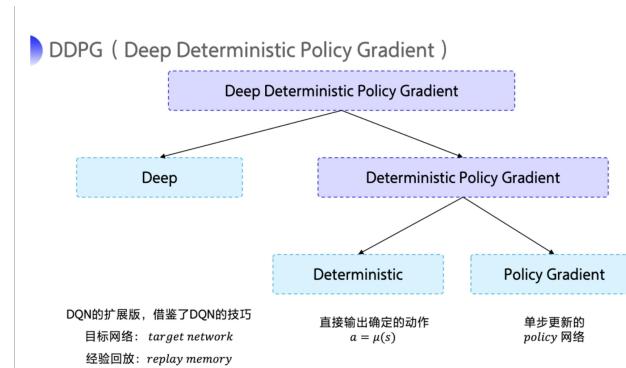


图 12.4

在连续控制领域，比较经典的强化学习算法就是 [深度确定性策略梯度 \(Deep Deterministic Policy Gradient\)](#)

Gradient, 简称 DDPG)。DDPG 的特点可以从它的名字当中拆解出来，拆解成 Deep、Deterministic 和 Policy Gradient。

- Deep 是因为用了神经网络；
- Deterministic 表示 DDPG 输出的是一个确定性的动作，可以用于连续动作的一个环境；
- Policy Gradient 代表的是它用到的是策略网络。REINFORCE 算法每隔一个 episode 就更新一次，但 DDPG 网络是每个 step 都会更新一次 policy 网络，也就是说它是一个单步更新的 policy 网络。DDPG 是 DQN 的一个扩展的版本。
- 在 DDPG 的训练中，它借鉴了 DQN 的技巧：目标网络和经验回放。
- 经验回放这一块跟 DQN 是一样的，但 target network 这一块的更新跟 DQN 有点不一样。

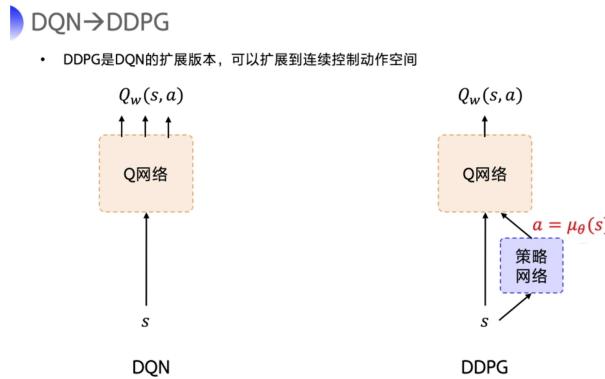


图 12.5

提出 DDPG 是为了让 DQN 可以扩展到连续的动作空间，就是我们刚才提到的小车速度、角度和电压的电流量这样的连续值。

- DDPG 直接在 DQN 基础上加了一个策略网络来直接输出动作值，所以 DDPG 需要一边学习 Q 网络，一边学习策略网络。
- Q 网络的参数用 w 来表示。策略网络的参数用 θ 来表示。
- 我们称这样的结构为 Actor-Critic 的结构。



图 12.6

通俗地解释一下 Actor-Critic 的结构，

- 策略网络扮演的就是 actor 的角色，它负责对外展示输出，输出舞蹈动作。
- Q 网络就是评论家 (critic)，它会在每一个 step 都对 actor 输出的动作做一个评估，打一个分，估计一下 actor 的 action 未来能有多少收益，也就是去估计这个 actor 输出的这个 action 的 Q 值大概是多少，即 $Q_w(s, a)$ 。Actor 就需要根据舞台目前的状态来做出一个 action。

- 评论家就是评委，它需要根据舞台现在的状态和演员输出的 action 对 actor 刚刚的表现去打一个分数 $Q_w(s, a)$ 。
 - Actor 根据评委的打分来调整自己的策略，也就是更新 actor 的神经网络参数 θ ，争取下次可以做得更好。
 - Critic 则是要根据观众的反馈，也就是环境的反馈 reward 来调整自己的打分策略，也就是要更新 critic 的神经网络的参数 w ，它的目标是要让每一场表演都获得观众尽可能多的欢呼声跟掌声，也就是要最大化未来的总收益。
- 最开始训练的时候，这两个神经网络参数是随机的。所以 critic 最开始是随机打分的，然后 actor 也跟着乱来，就随机表演，随机输出动作。但是由于我们有环境反馈的 reward 存在，所以 critic 的评分会越来越准确，也会评判的那个 actor 的表现会越来越好。
- 既然 actor 是一个神经网络，是我们希望训练好的策略网络，那我们就需要计算梯度来去更新优化它里面的参数 θ 。简单的说，我们希望调整 actor 的网络参数，使得评委打分尽可能得高。注意，这里的 actor 是不管观众的，它只关注评委，它就是迎合评委的打分 $Q_w(s, a)$ 而已。

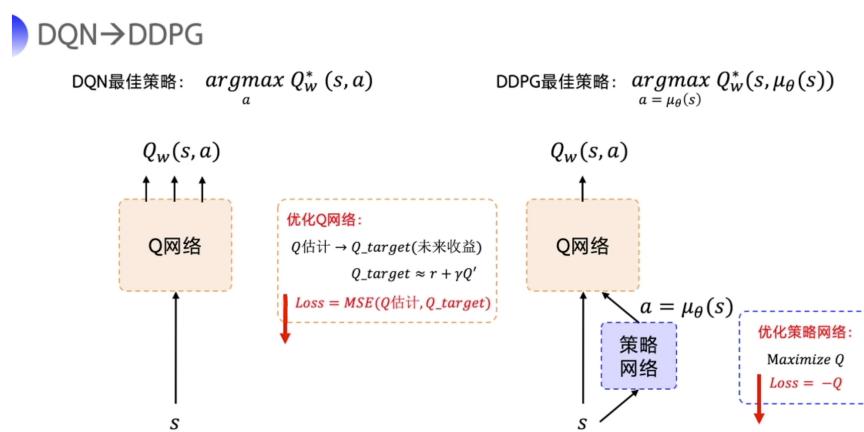


图 12.7

接下来就是类似 DQN。

- DQN 的最佳策略是想要学出一个很好的 Q 网络，学好这个网络之后，我们希望选取的那个动作使你的 Q 值最大。
- DDPG 的目的也是为了求解让 Q 值最大的那个 action。
 - Actor 只是为了迎合评委的打分而已，所用用来优化策略网络的梯度就是要最大化这个 Q 值，所以构造的 loss 函数就是让 Q 取一个负号。
 - 我们写代码的时候就是把这个 loss 函数扔到优化器里面，它就会自动最小化 loss，也就是最大化 Q。

这里要注意，除了策略网络要做优化，DDPG 还有一个 Q 网络也要优化。

- 评委一开始也不知道怎么评分，它也是在一步一步的学习当中，慢慢地去给出准确的打分。
- 我们优化 Q 网络的方法其实跟 DQN 优化 Q 网络的方法是一样的，我们用真实的 reward r 和下一步的 Q 即 Q' 来去拟合未来的收益 Q_{target} 。
- 然后让 Q 网络的输出去逼近这个 Q_{target} 。
 - 所以构造的 loss function 就是直接求这两个值的均方差。
 - 构造好 loss 后，我们就扔进去那个优化器，让它自动去最小化 loss 就好了。

我们可以把两个网络的 loss function 构造出来。

策略网络的 loss function 是一个复合函数。我们把 $a = \mu_\theta(s)$ 代进去，最终策略网络要优化的是策略

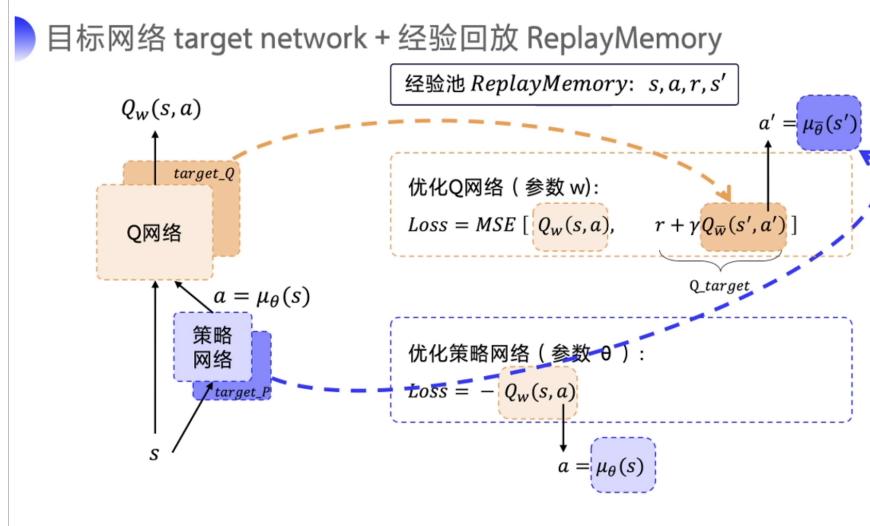


图 12.8

网络的参数 θ 。Q 网络要优化的是 $Q_w(s, a)$ 和 Q_{target} 之间的一个均方差。

但是 Q 网络的优化存在一个和 DQN 一模一样的问题就是它后面的 Q_{target} 是不稳定的。此外，后面的 $Q_{\bar{w}}(s', a')$ 也是不稳定的，因为 $Q_{\bar{w}}(s', a')$ 也是一个预估的值。

为了稳定这个 Q_{target} ，DDPG 分别给 Q 网络和策略网络都搭建了 target network。

- target_Q 网络就为了来计算 Q_{target} 里面的 $Q_{\bar{w}}(s', a')$ 。
- $Q_{\bar{w}}(s', a')$ 里面的需要的 next action a' 就是通过 target_P 网络来去输出，即 $a' = \mu_{\bar{\theta}}(s')$ 。
- 为了区分前面的 Q 网络和策略网络以及后面的 target_Q 网络和 target_P 策略网络，前面的网络的参数是 w ，后面的网络的参数是 \bar{w} 。
- DDPG 有四个网络，策略网络的 target 网络和 Q 网络的 target 网络就是颜色比较深的这两个，它只是为了让计算 Q_{target} 的时候能够更稳定一点而已。因为这两个网络也是固定一段时间的参数之后再跟评估网络同步一下最新的参数。

这里面训练需要用到的数据就是 s, a, r, s' ，我们只需要用到这四个数据。我们就用 Replay Memory 把这些数据存起来，然后再 sample 进来训练就好了。这个经验回放的技巧跟 DQN 是一模一样的。注意，因为 DDPG 使用了经验回放这个技巧，所以 DDPG 是一个 off-policy 的算法。

12.2.1 Exploration vs. Exploitation

DDPG 通过 off-policy 的方式来训练一个确定性策略。因为策略是确定的，如果 agent 使用同策略来探索，在一开始的时候，它会很可能不会尝试足够多的 action 来找到有用的学习信号。为了让 DDPG 的策略更好地探索，我们在训练的时候给它们的 action 加了噪音。DDPG 的原作者推荐使用时间相关的 OU noise，但最近的结果表明不相关的、均值为 0 的 Gaussian noise 的效果非常好。由于后者更简单，因此我们更喜欢使用它。为了便于获得更高质量的训练数据，你可以在训练过程中把噪声变小。

在测试的时候，为了查看策略利用它学到的东西的表现，我们不会在 action 中加噪音。

12.3 Twin Delayed DDPG(TD3)

虽然 DDPG 有时表现很好，但它在超参数和其他类型的调整方面经常很敏感。DDPG 常见的问题是已经学习好的 Q 函数开始显著地高估 Q 值，然后导致策略被破坏了，因为它利用了 Q 函数中的误差。

我们可以拿实际的 Q 值跟这个 Q-network 输出的 Q 值进行对比。实际的 Q 值可以用 MC 来算。根据当前的 policy 采样 1000 条轨迹，得到 G 后取平均，得到实际的 Q 值。

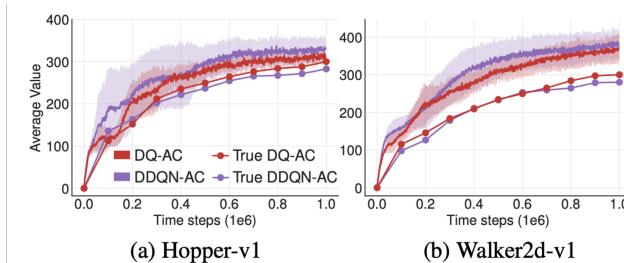


Figure 2. Measuring overestimation bias in the value estimates of actor critic variants of Double DQN (DDQN-AC) and Double Q-learning (DQ-AC) on MuJoCo environments over 1 million time steps.

图 12.9

双延迟深度确定性策略梯度 (Twin Delayed DDPG, 简称 TD3) 通过引入三个关键技巧来解决这个问题：

- 截断的双 Q 学习 (Clipped Double Q-learning)。TD3 学习两个 Q-function (因此名字中有“twin”)。TD3 通过最小化均方差来同时学习两个 Q-function: Q_{ϕ_1} 和 Q_{ϕ_2} 。两个 Q-function 都使用一个目标，两个 Q-function 中给出较小的值会被作为如下的 Q-target:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,targ}}(s', a_{TD3}(s'))$$

- 延迟的策略更新 (“Delayed” Policy Updates)。相关实验结果表明，同步训练动作网络和评价网络，却不使用目标网络，会导致训练过程不稳定；但是仅固定动作网络时，评价网络往往能够收敛到正确的结果。因此 TD3 算法以较低的频率更新动作网络，较高频率更新评价网络，通常每更新两次评价网络就更新一次策略。
- 目标策略平滑 (Target Policy smoothing)。TD3 引入了 smoothing 的思想。TD3 在目标动作中加入噪音，通过平滑 Q 沿动作的变化，使策略更难利用 Q 函数的误差。

这三个技巧加在一起，使得性能相比基线 DDPG 有了大幅的提升。

目标策略平滑化的工作原理如下：

$$a_{TD3}(s') = \text{clip}(\mu_{\theta,targ}(s') + \text{clip}(\epsilon, -c, c), a_{\text{low}}, a_{\text{high}})$$

其中 ϵ 本质上是一个噪音，是从正态分布中取样得到的，即 $\epsilon \sim N(0, \sigma)$ 。

目标策略平滑化是一种正则化方法。

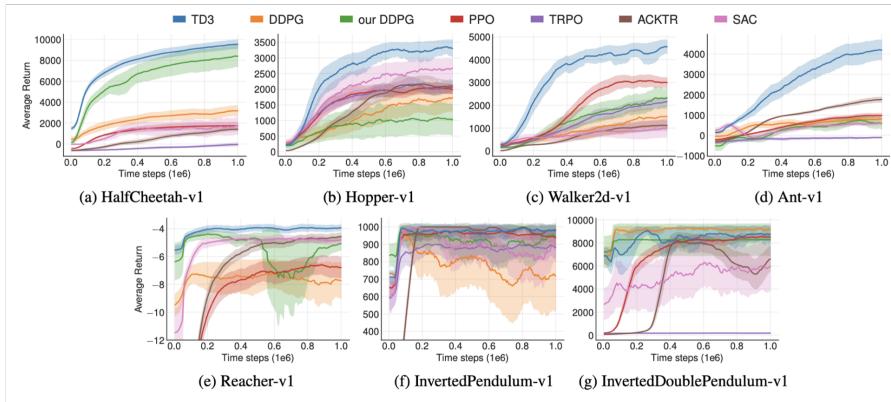


图 12.10

我们可以将 TD3 跟其他算法进行对比。这边作者自己实现的 DDPG(our DDPG) 和官方实现的 DDPG 的表现不一样，这说明 DDPG 对初始化和调参非常敏感。TD3 对参数不是这么敏感。在 TD3 的论文中，TD3 的性能比 SAC(Soft Actor-Critic) 高。但在 SAC 的论文中，SAC 的性能比 TD3 高，这是因为强化学习的很多算法估计对参数和初始条件敏感。

TD3 的作者给出了对应的实现：TD3 Pytorch implementation，代码写得很棒，我们可以将其作为一个强化学习的标准库来学习。

12.3.1 Exploration vs. Exploitation

TD3 以 off-policy 的方式训练确定性策略。由于该策略是确定性的，因此如果智能体要探索策略，则一开始它可能不会尝试采取足够广泛的动作来找到有用的学习信号。为了使 TD3 策略更好地探索，我们在训练时在它们的动作中添加了噪声，通常是不相关的均值为零的高斯噪声。为了便于获取高质量的训练数据，你可以在训练过程中减小噪声的大小。

在测试时，为了查看策略对所学知识的利用程度，我们不会在动作中增加噪音。

12.4 Keywords

- DDPG(Deep Deterministic Policy Gradient)：在连续控制领域经典的 RL 算法，是 DQN 在处理连续动作空间的一个扩充。具体地，从命名就可以看出，Deep 是使用了神经网络；Deterministic 表示 DDPG 输出的是一个确定性的动作，可以用于连续动作的一个环境；Policy Gradient 代表的是它用到的是策略网络，并且每个 step 都会更新一次 policy 网络，也就是说它是一个单步更新的 policy 网络。其与 DQN 都有目标网络和经验回放的技巧，在经验回放部分是一致的，在目标网络的更新有些许不同。

12.5 Questions

- 请解释随机性策略和确定性策略。答：
 - 对于随机性的策略 $\pi_\theta(a_t|s_t)$ ，我们输入某一个状态 s ，采取某一个 action 的可能性并不是百分之百，而是有一个概率 P 的，就好像抽奖一样，根据概率随机抽取一个动作。
 - 对于确定性的策略 $\mu_\theta(s_t)$ ，其没有概率的影响。当神经网络的参数固定下来了之后，输入同样的 state，必然输出同样的 action，这就是确定性的策略。
- 对于连续动作的控制空间和离散动作的控制空间，如果我们都采取使用 Policy 网络的话，分别应该如何操作？

答：首先需要说明的是，对于连续的动作控制空间，Q-learning、DQN 等算法是没有办法处理的，所以我们需要使用神经网络进行处理，因为其可以既输出概率值 $\pi_\theta(a_t|s_t)$ ，也可以输出确定的策略 $\mu_\theta(s_t)$ 。

- 要输出离散动作的话，最后的 output 的激活函数使用 softmax 就可以实现。其可以保证输出是的动作概率，而且所有的动作概率加和为 1。
 - 要输出连续的动作的话，可以在输出层这里加一层 tanh 激活函数。其作用可以把输出限制到 [-1,1] 之间。我们拿到这个输出后，就可以根据实际动作的一个范围再做一下缩放，然后再输出给环境。比如神经网络输出一个浮点数是 2.8，然后经过 tanh 之后，它就可以被限制在 [-1,1] 之间，它输出 0.99。然后假设说小车的一个速度的那个动作范围是 [-2,2] 之间，那我们就按比例从 [-1,1] 扩放到 [-2,2]，0.99 乘 2，最终输出的就是 1.98，作为小车的速度或者说推小车的力输出给环境。

12.6 Something About Interview

- 高冷的面试官：请简述一下 DDPG 算法？

答：深度确定性策略梯度 (Deep Deterministic Policy Gradient, 简称 DDPG) 使用 Actor Critic 结构，但是输出的不是行为的概率，而是具体的行为，用于连续动作的预测。优化的目的是为了将 DQN 扩展到连续的动作空间。另外，其字如其名：

- Deep 是因为用了神经网络；
- Deterministic 表示 DDPG 输出的是一个确定性的动作，可以用于连续动作的一个环境；
- Policy Gradient 代表的是它用到的是策略网络。REINFORCE 算法每隔一个 episode 就更新一次，但 DDPG 网络是每个 step 都会更新一次 policy 网络，也就是说它是一个单步更新的 policy 网络。
- 高冷的面试官：你好，请问 DDPG 是 on-policy 还是 off-policy，原因是什么呀？
答：off-policy。解释方法一，DDPG 是优化的 DQN，其使用了经验回放，所以为 off-policy 方法；解释方法二，因为 DDPG 为了保证一定的探索，对于输出动作加了一定的噪音，也就是说行为策略不再是优化的策略。
- 高冷的面试官：你是否了解过 D4PG 算法呢？描述一下吧。
答：分布的分布式 DDPG (Distributed Distributional DDPG，简称 D4PG)，相对于 DDPG 其优化部分为：
 - 分布式 critic: 不再只估计 Q 值的期望值，而是去估计期望 Q 值的分布，即将期望 Q 值作为一个随机变量来进行估计。
 - N 步累计回报: 当计算 TD 误差时，D4PG 计算的是 N 步的 TD 目标值而不仅仅只有一步，这样就可以考虑未来更多步骤的回报。
 - 多个分布式并行 actor: D4PG 使用 K 个独立的演员并行收集训练样本并存储到同一个 replay buffer 中。
 - 优先经验回放 (Prioritized Experience Replay, PER): 使用一个非均匀概率 π 从 replay buffer 中采样。

12.7 Solve Pendulum with DDPG

使用 Policy-Based 方法比如 DDPG 等实现 Pendulum-v0 环境

12.7.1 Pendulum-v0

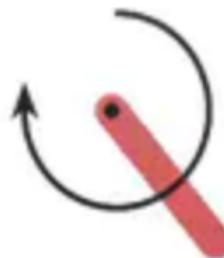


图 12.11

钟摆以随机位置开始，目标是将其摆动，使其保持向上直立。动作空间是连续的，值的区间为 [-2,2]。每个 step 给的 reward 最低为 -16.27，最高为 0。

环境建立如下：

```
env = gym.make('Pendulum-v0')
```

```
env.seed(1) # 设置env随机种子
n_states = env.observation_space.shape[0] # 获取总的状态数
```

12.7.2 强化学习基本接口

```

rewards = [] # 记录总的rewards
moving_average_rewards = [] # 记录总的经滑动平均处理后的rewards
ep_steps = []
for i_episode in range(1, cfg.max_episodes+1): # cfg.max_episodes为最大训练的episode数
    state = env.reset() # reset环境状态
    ep_reward = 0
    for i_step in range(1, cfg.max_steps+1): # cfg.max_steps为每个episode的补偿
        action = agent.select_action(state) # 根据当前环境state选择action
        next_state, reward, done, _ = env.step(action) # 更新环境参数
        ep_reward += reward
        agent.memory.push(state, action, reward, next_state, done) # 将state等这些transition存入memory
        state = next_state # 跳转到下一个状态
        agent.update() # 每步更新网络
        if done:
            break
    # 更新target network, 复制DQN中的所有weights and biases
    if i_episode % cfg.target_update == 0: # cfg.target_update为target_net的更新频率
        agent.target_net.load_state_dict(agent.policy_net.state_dict())
    print('Episode:', i_episode, ' Reward: {:.2f} %'.format(
        int(ep_reward), 'n_steps:', i_step, 'done: ', done, ' Explore: {:.2f}' % agent.epsilon))
    ep_steps.append(i_step)
    rewards.append(ep_reward)
    # 计算滑动窗口的reward
    if i_episode == 1:
        moving_average_rewards.append(ep_reward)
    else:
        moving_average_rewards.append(
            0.9*moving_average_rewards[-1]+0.1*ep_reward)
```

12.7.3 任务要求

训练并绘制 reward 以及滑动平均后的 reward 随 episode 的变化曲线图并记录超参数写成报告，如图 12.12 所示。

同时也可以绘制测试 (eval) 模型时的曲线，如图 12.13 所示。

也可以tensorboard查看结果，如图 12.14 所示。

12.7.4 注意

1. 本次环境 action 范围在 [-2,2] 之间，而神经网络中输出的激活函数 tanh 在 [0,1]，可以使用 NormalizedActions(gym.ActionWrapper) 的方法解决
2. 由于本次环境为惯性系统，建议增加 Ornstein-Uhlenbeck 噪声提高探索率，可参考知乎文章
3. 推荐多次试验保存 rewards，然后使用 searborn 绘制，可参考CSDN

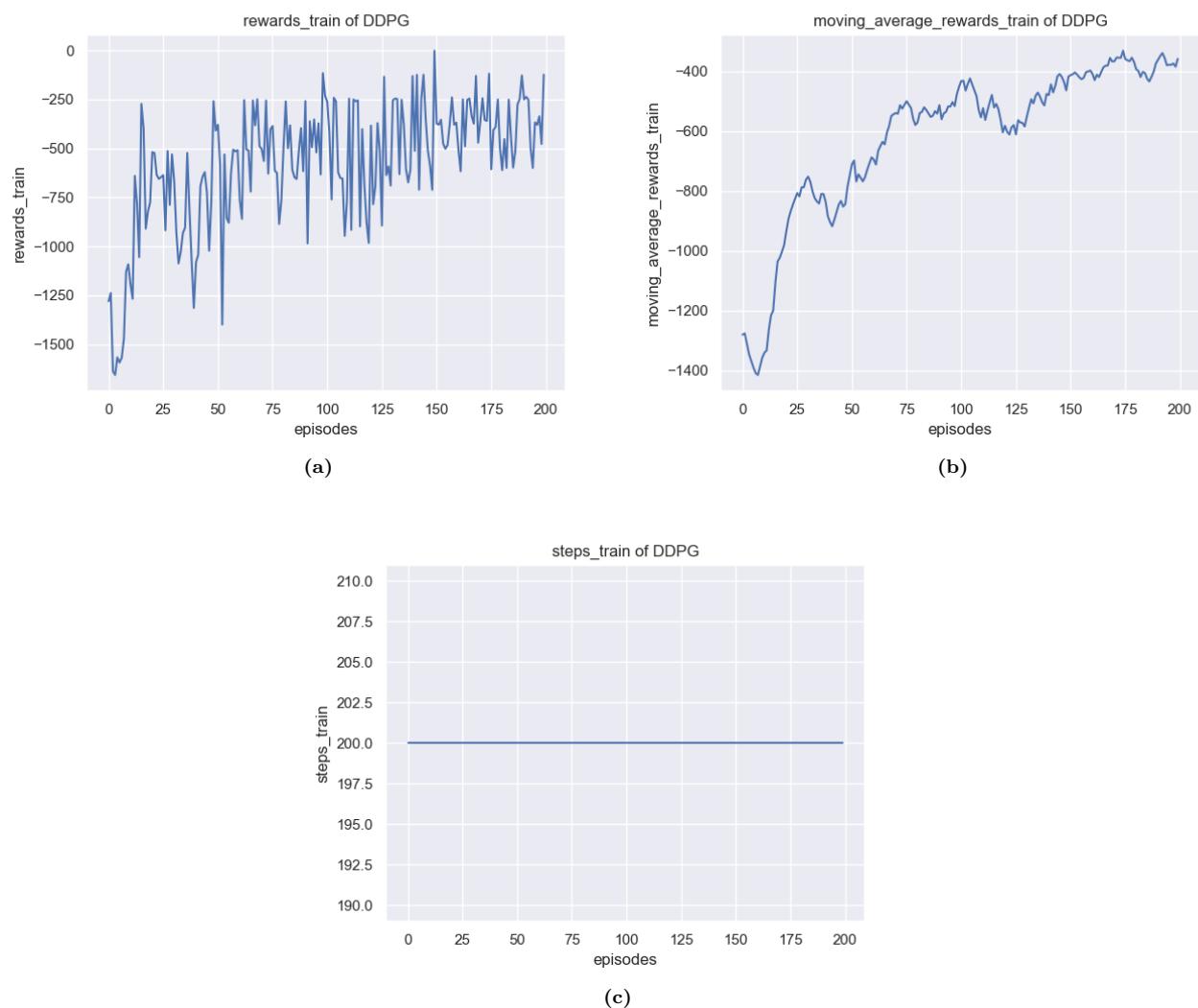


图 12.12

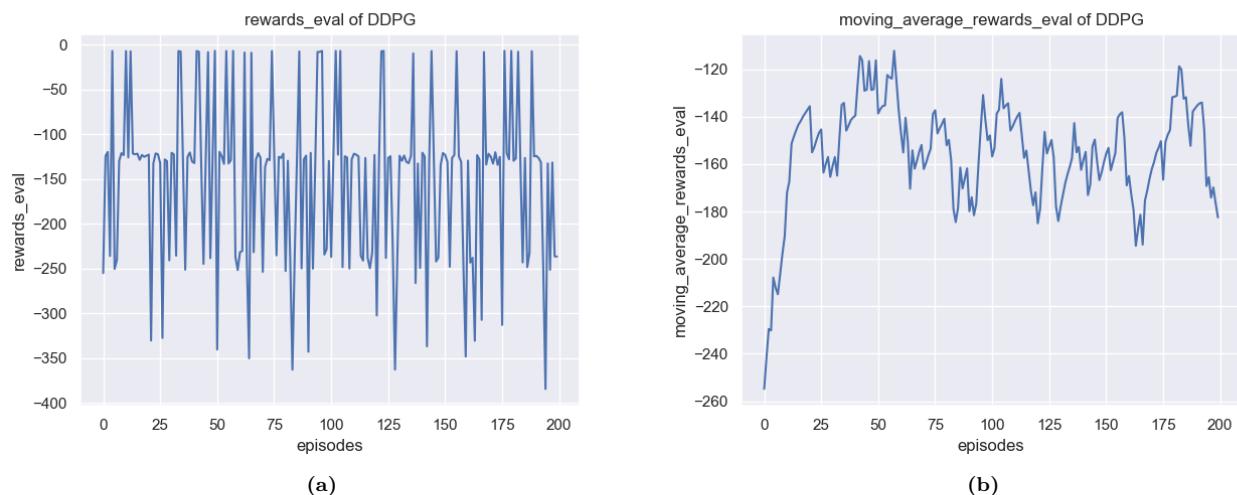


图 12.13

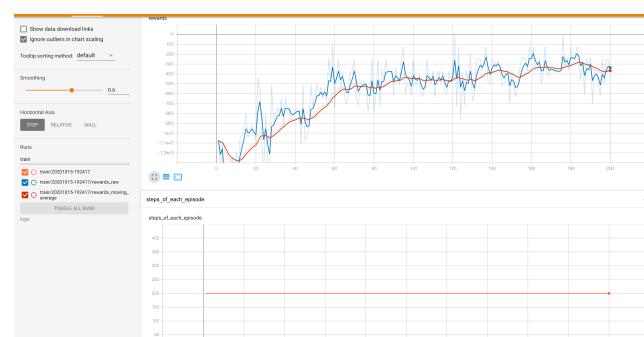


图 12.14

12.7.5 代码清单

main.py: 保存强化学习基本接口，以及相应的超参数，可使用 argparse

model.py: 保存神经网络，比如全链接网络

ddpg.py: 保存算法模型，主要包含 select_action 和 update 两个函数

memory.py: 保存 Replay Buffer

plot.py: 保存相关绘制函数

noise.py: 保存噪声相关

参考代码

References

- 百度强化学习
- OpenAI Spinning Up
- Intro to Reinforcement Learning (强化学习纲要)
- 天授文档

第 13 章 AlphaStar 论文解读

13.1 AlphaStar 以及背景简介

相比于之前的深蓝和 go，对于星际争霸 2 等策略对战型游戏，使用 AI 与人类对战难度更大。比如在星际争霸 2 中，操作枯燥是众所周知的，要想在 PVP 中击败对方，就得要学会各种战术，各种微操和 Timing。在游戏中你还得侦查对方的发展，做出正确判断进行转型，甚至要欺骗对方以达到战术目的。总而言之，想要上手这款游戏是非常困难的，对不起，DeepMind 就做到了。

AlphaStar 是 DeepMind 公司与暴雪使用深度强化学习技术进行 PC 与星际争霸 2 人类玩家进行对战的产品，其在近些年在星际争霸 2 中打败了职业选手以及 99.8% 的欧服玩家而被人所熟知。北京时间 2019 年 1 月 25 日凌晨 2 点，暴雪与谷歌 DeepMind 团队合作研究的星际争霸人工智能“AlphaStar”正式通过直播亮相。按照直播安排，AlphaStar 与两位《星际争霸 2》人类职业选手进行了 5 场比赛对决演示。加上并未在直播中演示的对决，在人类 vs AlphaStar 人工智能的共计 11 场比赛中，人类只取得了一场胜利。DeepMind 也将研究工作发表在了 2019 年 10 月的《Nature》杂志上。我们也将对于这篇 Paper 进行深入的分析，下面是论文的链接：Vinyals, Oriol, et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning." Nature (2019): 1-5.

13.2 AlphaStar 的模型输入输出是什么？——环境设计

构建 DRL 模型的第一部分就是构建输入输出，对于星际争霸 2 这个复杂的环境，paper 第一步做的就是将游戏的环境抽象成为许多的数据信息。

13.2.1 状态（网络的输入）

AlphaStar 将星际争霸 2 的环境状态分为四部分，分别为实体信息（Entities）、地图信息（Map）、玩家数据信息（Player data）、游戏统计信息（Game statistics）。

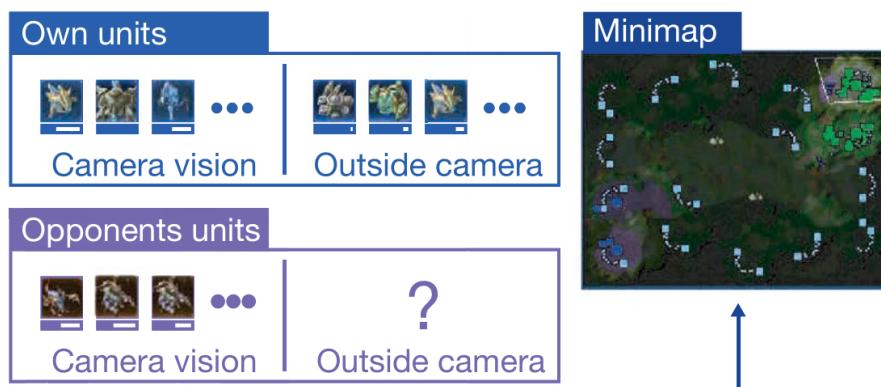


图 13.1

- 第一部分：实体信息，例如当前时刻环境中有什么建筑、兵种等等，并且我们将每一个实体的属性信息以向量的形式表示，例如对于一个建筑，其当前时刻的向量中包含此建筑的血量、等级、位置以及冷却时间等等信息。所以对于当前帧的全部实体信息，环境会给神经网络 N 个长度为 K 的向量，各表示此刻智能体能够看见的 N 个实体的具体信息。（向量信息）
- 第二部分：地图信息，这个比较好理解，也就是将地图中的信息以矩阵的形式送入神经网络中，来表示当前状态全局地图的信息。（向量信息或者说是图像信息）
- 第三部分：玩家数据信息，也就是当前状态下，玩家的等级、种族等等信息。（标量信息）
- 第四部分：游戏统计信息，相机的位置（小窗口的位置，区别于第二部分的全局地图信息），还有当前游戏的开始时间等等信息。（标量信息）

13.2.2 动作（网络的输出）

AlphaStar 的动作信息主要分为六个部分，分别为动作类型（Action type）、选中的单元（Selected units）、目标（Target）、执行动作的队列（Queued）、是否重复（Repeat）、延时（Delay），每一个部分间是有关联的。

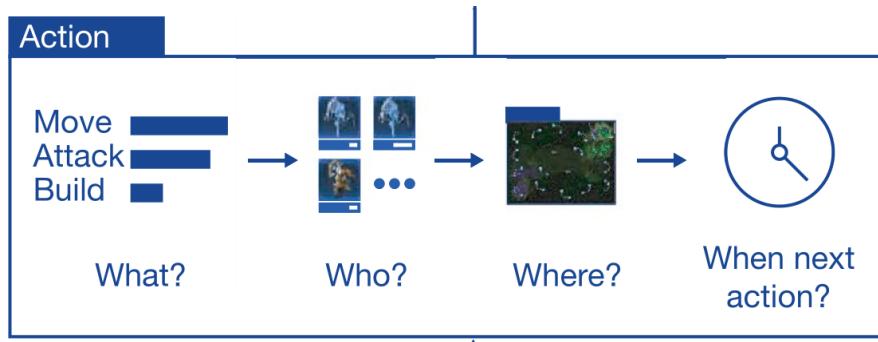
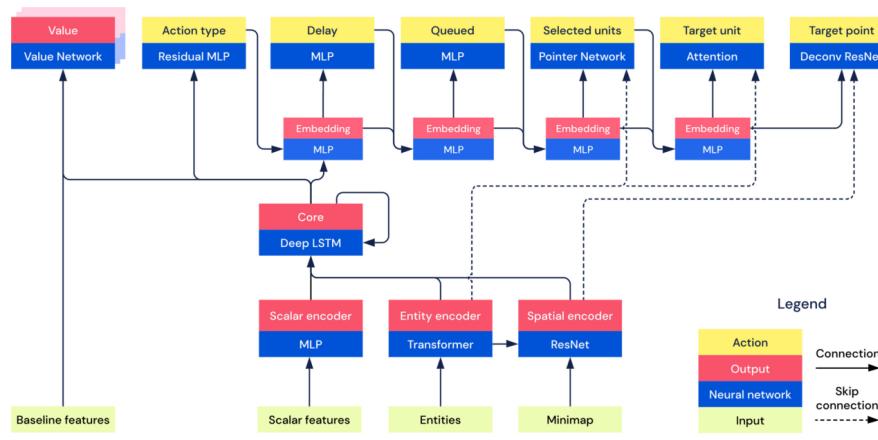


图 13.2

- 第一部分：动作类型，即下一次要进行的动作的类型是移动小兵、升级建筑还是移动小窗口的位置等等
- 第二部分：选中的单元，即承接第一部分，例如我们要进行的动作类型是移动小兵，那么我们就应该选择具体“操作”哪一个小兵
- 第三部分：目标，承接第二部分，我们操作小兵 A 后，是要去地图的某一个位置还是去攻击对手的哪一个目标等等，即选择目的地和攻击的对象
- 第四部分：执行动作的队列，具体说是是否立即执行动作，对于小兵 A，我们是到达目的地后直接进行攻击还是等待
- 第五部分：是否重复做动作，如果需要小兵 A 持续攻击，那么就不需要再通过网络计算得到下一个的动作了，直接重复以上一个动作的相同动作即可。
- 第六部分：延时，也就是等候多久才接收网络的输入，可以理解为我们人类玩家的一个操作的延迟等等

13.3 AlphaStar 的计算模型是什么呢？——网络结构

上面我们说明了 AlphaStar 网络的输入和输出，即状态和动作，那么从状态怎么得到动作呢？其网络结构是怎么样的呢？



Extended Data Fig. 3 | Overview of the architecture of AlphaStar. A detailed description is provided in the Supplementary Data, Detailed Architecture.

图 13.3

13.3.1 输入部分

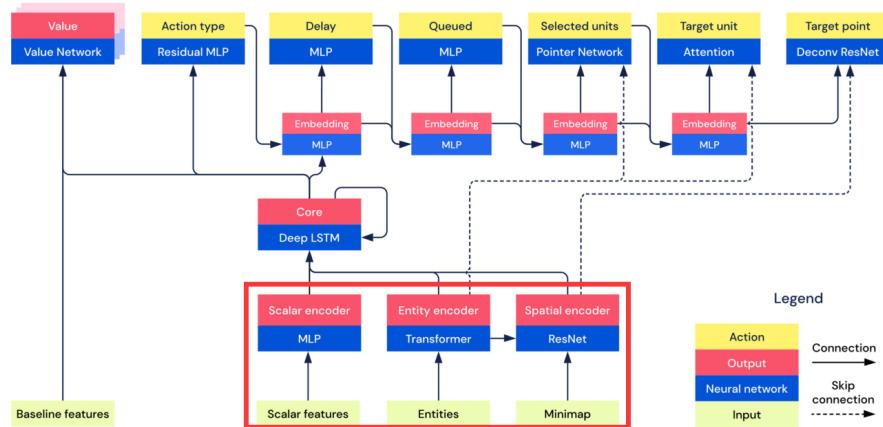


图 13.4

从上图的红框可以看出，模型的输入框架中主要有三个部分，即 Scalar features（标量特征），例如前面叙述的玩家的等级、小窗口的位置等等信息、Entities（实体），是向量即前面所叙述的一个建筑一个兵的当前的所有属性信息、Minimap（地图），即上面说的图像的数据。

- 对于 Scalar features（标量特征），使用多层感知器（MLP），就可以得到对应的向量，或者说是一个 embedding 的过程。
- 对于 Entities，使用 NLP 中常用的 transformer 作为 encoder
- 对于 Minimap，使用图像中常用的 Resnet 作为 encoder，得到一个定长的向量。

13.3.2 中间过程

中间过程比较简单，即通过一个 deep LSTM 进行融合三种当前状态下的 embedding 进行下一时刻的 embedding 输出，并且将该结果分别送入 ValueNetwork、Residual MLP 以及 Action type 的后续的 MLP 中。

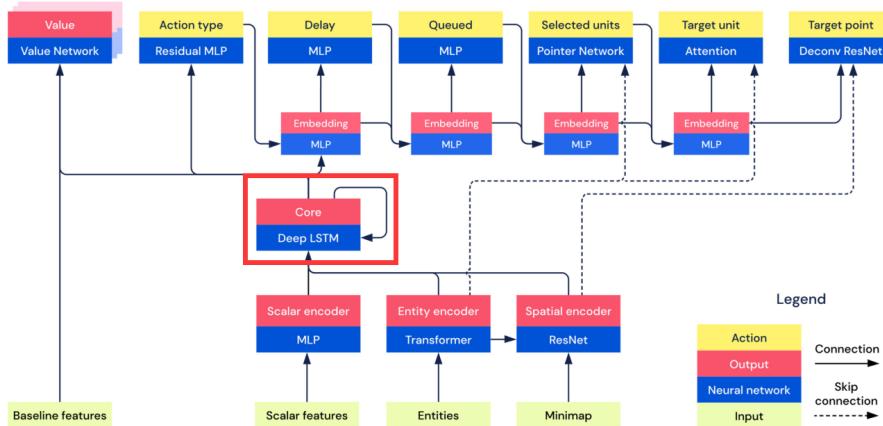


图 13.5

13.3.3 输出部分

正如前面介绍的，输出的动作是前后有关联的，按照顺序

- 首先是动作类型 (Action type)：使用 Deep LSTM 的 embedding 的向量作为输入，使用 residual MLP 得到 Action type 的 softmax 的输出结果，并传给下一个子模型进行 embedding。

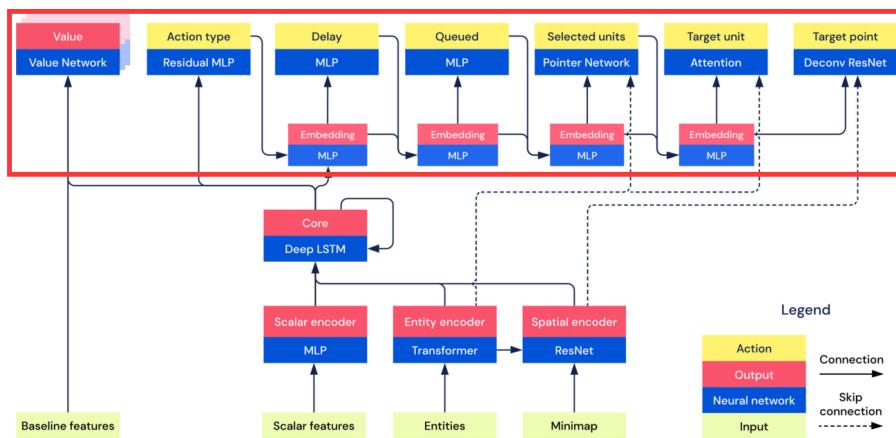


图 13.6

- 然后是延时 (Delay): 使用上一个上面的 embedding 的结果以及 Deep LSTM 的结果一起输入 MLP 后得到结果，并传给下一个子模型进行 embedding。
- 接下来是执行动作的队列 (Queued): 使用 delay 的结果以及 embedding 的结果一起输入 MLP 后得到结果，并传给下一个子模型进行 embedding。
- 然后是选中的单元 (Selected units): 使用 queued 的结果、embedding 的结果以及 Entity encoder 的全部结果（非平均的结果）一起送入到 Pointer Network 中得到结果，并传给下一个子模型进行 embedding。这里的 Pointer Netowrk 为指针网络，即输入的是一个序列，输出是另外一个序列，并且，输出序列的元素来源于输入的序列，主要用于 NLP 中，在这里很适合与我们的 Selected units 的计算。
- 接着是目标单元 (Target unit) 和目标区域 (Target point) 两者二选一进行，对于 Target unit，使用 attention 机制得到最优的动作作用的一个对象，对于 target point，使用反卷积神经网络，将 embedding 的向量，反卷积为 map 的大小，从而执行目标移动到某一点的对应动作。

13.4 庞大的 AlphaStar 如何训练呢？——学习算法

对于上面复杂的模型，AlphaStar 究竟如何来进行训练呢？总结下来一共分为 4 个部分，即监督学习（主要是解决训练的初始化问题）、强化学习、模仿学习（配合强化学习）以及多智能体学习和自学习（面向对战的具体问题），下面我们一一分析：

13.4.1 监督学习

在训练一开始首先使用监督学习利用人类的数据进行一个比较好的初始化。模型的输入是收集到的人类的对局的信息，输出是训练好的神经网络。具体的做法是，对于收集到了人类的对局数据，在对于每一个时刻解码游戏的状态，将每一时刻的状态送入网络中得到以上每一个动作的概率分布，最终计算模型的输出以及人类数据的 KL Loss，并以此进行网络的优化，其中在 KL Loss 中需要使用不同的 Loss 函数，例如，Action 类型的输出，即分类问题的 loss 就需要使用 Cross Entropy。而对于 target location 等类似于回归问题的就需要计算 MSE。当然还有一些细节，大家可以自行阅读 paper。总之，经过监督学习，我们的模型输出的概率分布就可以与人类玩家输出的概率分布类似。

13.4.2 强化学习

这里的目标就是通过优化策略使得期望的 reward 最大，即

$$J(\pi_\theta) = E_{\pi_\theta} \sum_{t=0} r(s_t, a_t)$$

但 AlphaStar 的训练的模型使用不是采样的模型，即 off-policy 的模型，这是因为其使用的架构为类似于 IMPALA 的结构，即 Actor 负责与环境交互并采样，learner 负责优化网络并更新参数，而 Actor 和 learner 通常是异步进行计算的，并且由于前面介绍的输出的动作的类型空间复杂，所以导致我们的 value function 的拟合比较困难。

这里 AlphaStar 利用了以下的方式进行强化学习模型的构建：

- 首先是采取了经典的 Actor-critic 的结构，使用策略网络给出当前状态下的智能体的动作，即计算 $\pi(a_t|s_t)$ ，使用价值网络计算当前状态下的智能体的期望收益，即计算 $V(s_t) = E \sum_{t'=t} r_{t'} = E_{a_t}[r(s_t, a_t) + V(s_{t+1})]$ 。具体的计算方法是：

– 对于当前的状态 s ，计算当前计算出的动作 a 相对于“平均动作”所能额外获得的奖励。 $A(s_t, a_t) = [r(s_t, a_t) + V(s_{t+1})] - V(s_t)$ ，即当前动作的预期收益减去当前状态的预期收益。在 AlphaStar 中，UPGO (Upgoing Policy Update) 也得到了应用，即 UPGO 使用了一个迭代变量 G_t 来取代原来的动作预期收益的 $r(s_t, a_t) + V(s_{t+1})$ ，即把未来乐观的信息纳入到我们额外奖励中，上式可改写为：

$$A(s_t, a_t) = G_t - V(s_t)$$

$$G_t = \begin{cases} r_t + G_{t+1} & Q(s_{t+1}, a_{t+1}) \geq V(s_{t+1}) \\ r_t + V(s_{t+1}) & \text{otherwise} \end{cases}$$

- 在基于上面计算得到的 action，更新策略梯度，即 $\nabla_\theta J = A(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)$ ，在我们之前的笔记中也介绍了，如果基于 π_θ 的分布不好求解，或者说学习策略 π_θ 与采集策略 π_μ 不同的话，我们需要使用重要性采样的方法，即 $\nabla_\theta J = E_{\pi_\mu} \frac{\pi_\theta(a_t|s_t)}{\pi_\mu(a_t|s_t)} A^{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)$ 。当然我们还需防止 $\frac{\pi_\theta(a_t|s_t)}{\pi_\mu(a_t|s_t)}$ 出现无穷大的情况，我们需要使用 V-trace 限制重要性系数。这也是用于 off-policy 的一个更新方法，在 IMPALA 论文中的 4.1 小节有所体现。即将重要性系数的最大值限制为 1，公式可表达如下：

$$\nabla_\theta J = E_{\pi_\mu} \rho_t A^{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$\rho_t = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_\mu(a_t|s_t)}, 1\right)$$

- 利用了 TD(λ) 来优化价值网络，并同时输入对手的数据。对于我们的价值函数

$$V^{\pi_\theta}(s_t) = E_{\pi_\theta} \sum_{t'=t} \gamma^{t'-t} r(s_t, a_t) = E_{a_t \sim \pi_\theta(\cdot|s_t)} [r(s_t, a_t) + \gamma V(s_{t+1})]$$

，可以使用 TD 的方法计算 MSE 损失，有如下几种：

- TD(0)，表达式为 $L = [(r_t + \gamma V_{t+1}) - V_t]^2$ ，即当前 step 的信息，有偏小方差
- TD(1) 也就是 MC 方法，表达式为 $L = [(\sum_{t'=t}^\infty \gamma^{t'-t} r_{t'}) - V_t]^2$ ，即未来无穷个 step 的信息，无偏大方差
- TD(λ)，以上两个方法的加权平均。即平衡当前 step、下一个 step 到无穷个 step 后的结果
 - * 已知对于 $\lambda \in (0, 1)$, $(1 - \lambda) + (1 - \lambda)\lambda + (1 - \lambda)\lambda^2 + \dots = 1$
 - * $R_t = \lim_{T \rightarrow \infty} (1 - \lambda)(r_t + V_{t+1}) + (1 - \lambda)\lambda(r_t + \gamma r_{t+1} + \gamma^2 V_{t+2}) + \dots$

13.4.3 模仿学习

使用模仿学习额外引入了监督学习 Loss 以及人类的统计量 Z ，即对于 Build order (建造顺序)、Build Units (建造单元)、Upgrades (升级)、Effects (技能) 等信息进行了奖励。对于统计量 Z ，本质来说是一系列的数据，将其作为输入信息输入到策略网络和价值网络中。另外对于人类信息的利用还体现在前面介绍的使用监督学习进行网络的预训练工作。

13.4.4 多智能体学习/自学习

自学习在 AlphaGo 中得到了应用也就是自己和自己玩，Alpha 对此做了一些更新，即有优先级的虚拟自学习策略，对于虚拟自学习就是在训练过程中，每一些时间就进行存档，并随机均匀的从存档中选出对手与正在训练的智能体对战。而有优先级的虚拟自学习指的是优先挑选能击败我的或者说常能打败智能体的对手进行训练对战，评判指标就是概率。对于 AlphaStar 中，其训练的 agent 分为了三种，

- Main Agent (主智能体)，即正在训练的智能体及其祖先；其中有 50% 的概率从联盟中的所有人中挑选，使用有优先级的虚拟自学习策略，即能打败我的概率高，不能打败我的概率低，有 35% 的概率与自己对战，有 15% 的概率与能打败我的联盟利用者或者老的主智能体对战，通过利用了有优先级的虚拟自学习策略。
- League Exploiter (联盟利用者)：能打败联盟中的所有智能体的 agent；其按照有优先级的虚拟自学习策略计算的概率与全联盟的对手训练，在以 70% 的胜率打败所有的 agent 或者距离上次存档 2×10^9 step 后就保存，并且在存档的时候，有 25% 概率把场上的联盟利用者的策略重设成监督学习给出的初始化。
- Main Exploiter (主利用者)：能打败训练中的所有 agent，在训练的过程中，随机从 3 个中挑 1 个主智能体，如果可以以高于 10% 的概率打败该 agent 就与其进行训练，如果不能就从其他的的老主智能体中再挑选对手，当以 70% 的胜率打败全部三个正在学习的策略主智能体，或者距上次存档 4×10^9 个 step 之后就存，并且进行重设初始化的操作。

他们的区别在于：

- 如何选取训练过程中对战的对象
- 在什么情况下存档 (snapshot) 现在的策略
- 以多大的概率将策略的参数重设为监督训练给出的初始化

13.5 AlphaStar 实验结果如何呢？——实验结果

13.5.1 宏观结果

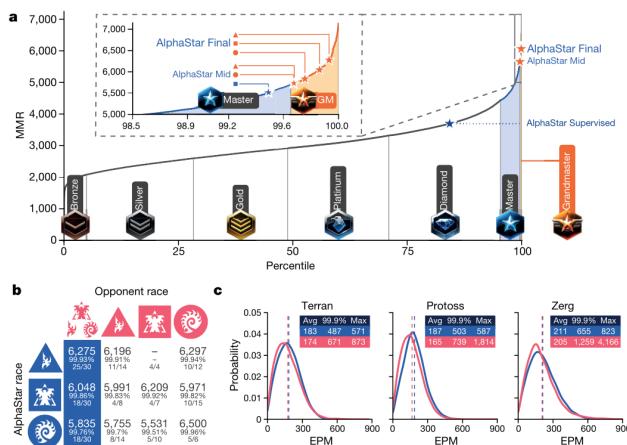


图 13.7

图 A 为训练后的 agent 与人类对战的结果（天梯图），具体地，刚刚结束监督学习后的 AlphaStar 可以达到钻石级别，而训练到一半（20 天）以及训练完结（40 天）的 AlphaStar 可以达到 GM 的级别。AlphaStar 已经可以击败绝大多数的普通玩家。

图 B 为不同种族间对战的胜率。

图 C 为《星际争霸 II》报告的每分钟有效行动分布情况（EPM），其中蓝色为 AlphaStar Final 的结果，红色为人类选手的结果虚线显示平均值。

13.5.2 其他实验（消融实验）

AlphaStar 的论文中也使用了消融实验，即控制变量法，来进一步分析每一个约束条件对于对战结果的影响。下面举一个特别的例子：

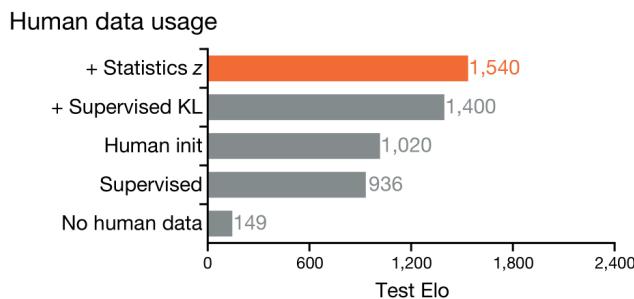


图 13.8

上面的图片表示的是人类对局数据的使用的情况。可以看到如果没有人类对局数据的情况下，数值仅仅为 149，但是只要经过了简单的监督学习，对应的数值就可以达到 936，当然使用人类初始化后的强化学习可以达到更好的效果，利用强化学习加监督学习的 KL Loss 的话可以达到接近于完整的利用人类统计量 Z 的效果。可以分析出，AlphaStar 中人类对局的数据对于整个 model 的表现是很重要的，其并没有完全像 AlphaGo 一样，可以不使用人类数据的情况。

13.6 关于 AlphaStar 的总结

13.6.1 总结

- AlphaStar 设计了一个高度可融合图像、文本、标量等信息的神经网络架构，并且对于网络设计使用了 Autoregressive 解耦了结构化的 action space。
- 模仿学习和监督学习的内容，例如人类统计量 Z 的计算方法
- 复杂的 DRL 方法以及超复杂的训练策略
- 当然了，大量的计算资源 (Each agent was trained using 32 third-generation tensor processing units (TPUs 23) over 44 days. During league training almost 900 distinct players were created.)