4 深度 Q 网络

本章将介绍的 DQN 算法全称为深度 Q 网络算法，是深度强化学习算法中最重要的算法之一。我们将从基于时间差分学习的 Q-Learning 算法入手，介绍 DQN 算法及其变体。在本章的最后，我们提供了代码示例，并对 DQN 及其变体进行实验比较。

强化学习最重要的突破之一是 Q-Learning 算法。它是一种离线策略（Off-Policy）的时间差分（Temporal Difference）算法，此前在第 2 章中有介绍。在使用表格（Tabular）的情况下或使用线性函数逼近 Q 函数时，Q-Learning 已被证明可以收敛于最优解。然而，当使用非线性函数逼近器（如神经网络）来表示 Q 函数时，Q-Learning 并不稳定，甚至是发散的 (Tsitsiklis et al., 1996)。随着深度神经网络技术的不断发展，**深度 Q 网络**（Deep Q-Networks，DQN）算法 (Mnih et al., 2015) 解决了这一问题，并点燃了深度强化学习的研究。在本章中，我们将先回顾 Q-Learning 的背景。之后介绍 DQN 算法及其变体，并给出详细的理论和解释。最后，在 4.8 节，我们将通过代码展示算法在雅达利游戏上的实现细节与实战表现，为读者提供快速上手的实战学习过程。每种算法的完整代码可以在随书提供的代码仓库中找到[1]。

无模型（Model-Free）方法为解决基于 MDP 的决策问题提供了一种通用的方法。其中“模型”是指显式地对 MDP 相关的转移概率分布和回报函数建模，而时间差分（Temporal Difference，TD）学习就是一类无模型方法。在 2.4 节中，我们讨论过，当拥有一个完美的 MDP 模型时，通过递归子问题的最优解，就可以得到动态规划的最优方案。TD 学习也遵循了这样一种思想，即使对子问题的估计并非一直是最优的，我们也可以通过自举（Bootstrapping）来估计子问题的值。

[1]代码链接见读者服务

子问题通过 MDP 中的状态表示。在策略 π 下，状态为 s 时的 value 值（V 值）$v_\pi(s)$ 被定义为从状态 s 开始，以策略 π 进行动作的预期回报：

$$v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1})|S_t = s], \tag{4.1}$$

此处的 $\gamma \in [0, 1]$ 是衰减率。TD 学习用自举法分解上述估计。给定价值函数 $V : \mathcal{S} \to \mathbb{R}$，TD(0) 是一个最简单的版本，它只应用一步自举，如下所示：

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)] \tag{4.2}$$

此处的 $R_t + \gamma V(S_{t+1})$ 和 $R_t + \gamma V(S_{t+1}) - V(S_t)$ 分别被称为 TD 目标和 TD 误差。

策略的评估值提供了一种对策略的动作质量（Quality）进行评估的方法。为了进一步了解如何选择某一特定状态下的动作，我们将通过 Q 值来评估状态-动作组合的效果。Q 值可以这样被估计：

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \tag{4.3}$$

有了 Q 值对策略进行评估之后，我们只需要找到一种能提升 Q 值的方法就能提升策略的效果。最简单的提升效果的方法就是通过贪心的方法执行动作：$\pi'(s) = \arg\max_{a'} q^\pi(s, a')$。由 $q_{\pi'}(s, a) = \max_{a'} q_\pi(s, a') \geqslant q_\pi(s, a)$ 我们可以知道，贪心的策略一定不会得到一个更差的解法。考虑到探索的必要性，我们可以用一种替代方案来提升策略的效果。在该方案中，多数情况下我们仍然选择贪心动作，但是同时会以一个小概率 ϵ，从所有动作中以相同概率随机选择一个动作。该方法被称为 ϵ-贪心。我们可以这样计算 ϵ-贪心策略中 π' 的 Q 值：

$$q_\pi(s, \pi'(s)) = (1 - \epsilon) \max_{a \in \mathcal{A}} q_\pi(s, a) + \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a). \tag{4.4}$$

值得注意的是，$\frac{\pi(s,a) - \epsilon/|\mathcal{A}|}{1-\epsilon}$ 在 $a \in \mathcal{A}$ 上的和为 1。由于最大值不小于加权平均值，所以可以得到：

$$\begin{aligned} q_\pi(s, \pi'(s)) &= (1 - \epsilon) \max_{a \in \mathcal{A}} q_\pi(s, a) \sum_{a \in \mathcal{A}} \frac{\pi(s, a) - \epsilon/|\mathcal{A}|}{1 - \epsilon} + \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) \\ &\geqslant (1 - \epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(s, a) - \epsilon/|\mathcal{A}|}{1 - \epsilon} q_\pi(s, a) + \frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, \pi(s)), \end{aligned} \tag{4.5}$$

由此得知，通过 ϵ-贪心策略 π' 进行动作产生的 Q 值并不会小于原始的策略 π。也就是说，ϵ-贪心方法能确保策略的优化。接下来，我们将在下一节中讨论如何使用 Q 函数进行策略优化。

4.1 Sarsa 和 Q-Learning

更新 Q 函数的方式也和 TD(0) 中更新 V 函数的方式相似，直接在每次发生非终结（Non-Terminal）状态下的状态转移之后，用此时的状态 S_t 对 Q 函数进行更新即可。

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4.6}$$

此处的 A_t 和 A_{t+1} 动作都是通过基于 Q 值的 ϵ-贪心方法来选择的。如果 S_{t+1} 是一个终结状态（Terminal State），则 $Q(S_{t+1}, A_{t+1})$ 将被设置为 0。我们能不断地估计行为策略 π 产生的 Q，同时让 π 趋近于基于 Q 的贪心策略。此算法就是 Sarsa 算法。值得注意的是，策略 π 在 Sarsa 中有两个职责：产生经验和提升策略。通常来说，用来产生行为的策略被称为行为策略，而用来评估和提升的策略被称为目标策略。当算法中的行为策略和目标策略是同一个策略时（例如 Sarsa），该算法就是一种在线策略（On-Policy）方法。

在线策略方法本质上是一种试错的过程，当前策略产生的经验仅会被直接用于进行策略提升。离线策略方法考虑一种反思的策略，使得反复使用过去的经验成为了可能。Q-Learning 就是一种离线策略方法。其最简单的形式，即单步（One-Step）Q-Learning 遵循如下更新规则：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4.7}$$

此处的 A_t 是通过基于 Q 的 ϵ-贪心方法采样得到的。注意 A_{t+1} 是通过贪心方式选择的，此处与 Sarsa 不同。也就是说，Q-Learning 中的行为策略也是 ϵ-贪心，但是目标策略是贪心（Greedy）策略。单步 Q-Learning 只考虑当前的状态转移，而我们可以选择多步（Multi-Steps）Q-Learning 方法，在近似情况下，通过使用多步奖励（Multi-Steps Rewards）来获得更加精准的 Q 值。要注意，多步 Q-Learning 中需要考虑后续奖励的不匹配问题，以保持 Q 函数对目标策略预期回报（参考公式 (4.3)）的近似。我们将在第 4.7 节中继续对多步 Q-Learning 展开讨论。

4.2 为什么使用深度学习：价值函数逼近

在使用表格方式表示 Q 函数的时候，Q 函数可以表示为一个大型二维表格。也就是说，每个离散的状态和动作都有一个单独的条目。然而该方法在处理具有大规模数据空间（如原始像素输入）的任务时将十分低效，更不用说具有连续数据的控制任务了。幸运的是，通过使用函数逼近从不同输入进行泛化的技术已经得到了广泛的研究，我们可以将其应用于基于价值（Value-based）的强化学习。

接下来，我们考虑 Q-Learning 中使用参数 θ 进行函数拟合。函数拟合器可以是线性模型、决

策树或者神经网络。之后，我们通过 (4.7) 式子进行更新，它可以被重写为

$$\theta_t \leftarrow \arg\min_{\theta} \mathcal{L}(Q(S_t, A_t; \theta), R_t + \gamma Q(S_{t+1}, A_{t+1}; \theta)) \tag{4.8}$$

此处的 $\mathcal{L}$ 代表损失函数，如均方误差（Mean Squared Error）。对于上述的优化问题，可以通过批量采样构造出拟合 Q 迭代（Fitted Q Iteration）(Riedmiller, 2005)，其过程如算法 4.15 所示，其中 S_i' 是 S_i 的后继状态。该算法的一个在线随机的变种就是如算法 4.16 所示的在线 Q 迭代（Online Q Iteration）算法。

算法 4.15 拟合 Q 迭代

for 迭代数 $i = 1, T$ **do**
  收集 D 份采样 $\{(S_i, A_i, R_i, S_i')\}_{i=1}^{D}$
  for t $= 1, K$ **do**
    设置 $Y_i \leftarrow R_i + \gamma \max_a Q(S_i', a; \theta)$
    设置 $\theta \leftarrow \arg\min_{\theta'} \frac{1}{2} \sum_{i=1}^{D} (Q(S_i, A_i; \theta') - Y_i)^2$
  end for
end for

算法 4.16 在线 Q 迭代

for 迭代数 $= 1, T$ **do**
  选择动作 a 与环境交互，并得到观察数据 (s, a, r, s')
  设置 $y \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta)$
  设置 $\theta \leftarrow \theta - \alpha(Q(s, a; \theta) - y) \frac{\mathrm{d}Q(s,a;\theta)}{\mathrm{d}\theta}$
end for

值得注意的是，拟合 Q 迭代和在线 Q 迭代都是离线策略算法，因此，它们可以多次重用过去的经验。我们将在下一节对此进行深入讨论。

在 2.4.2 节中，我们通过贝尔曼最优回溯算子 $\mathcal{T}^*$ 介绍了值迭代的收敛性。我们定义一个新的运算符 $\mathcal{B}$，其函数近似为 $\mathcal{B}V = \arg\min_{V' \in \Omega} \mathcal{L}(V', V)$，其中 Ω 是所有可近似的值函数的集合。值得注意的是，$\mathcal{B}$ 中的 $\arg\min$ 可以看作是 $\mathcal{T}^*V$ 到 Ω 的映射。所以函数近似的回溯算子可以表示为 $\mathcal{B}\mathcal{T}^*$。而 $\mathcal{T}^*$ 在无穷范式（∞-norm）下收敛，$\mathcal{B}$ 则是在 L2 范式下的 MSE 损失下收敛。然而 $\mathcal{B}\mathcal{T}^*$ 不以任何形式收敛。因此，当用神经网络等非线性函数逼近器来表示数值函数时，数值迭代是不稳定的，甚至可能发散 (Tsitsiklis et al., 1997)。我们将在下一节讨论深度神经网络训练的稳定性。

4.3 DQN

在上一节中，我们介绍了近似学习状态-动作值函数的方法及其收敛不稳定性。为了在使用原始像素输入的复杂问题中实现端到端决策，DQN 通过两个关键技术结合 Q-Learning 和深度学习来解决不稳定性问题，并在雅达利游戏上取得了显著进展。

第一个关键技术被称为**回放缓存**（Replay Buffer）。这是一种被称为经验重演的生物学启发机制 (Lin, 1993; McClelland et al., 1995; O'Neill et al., 2010)。在每个时间步 t 中，DQN 先将智能体获得的经验 (S_t, A_t, R_t, S_{t+1}) 存入回放缓存中，然后从该缓存中均匀采样小批量样本用于 Q-Learning 更新。回放缓存相较于拟合 Q 迭代有几个优势。首先，它可以重用每个时间步的经验来学习 Q 函数，这样可以提高数据使用效率。其次，如果像拟合 Q 迭代那样没有回放缓存，那么一个批次中的样本将会是连续采集的，即样本高度相关。这样会增加更新的方差。最后，经验回放防止用于训练的样本只来自上一个策略，这样能平滑学习过程并减少参数的震荡或发散。在实践中，为了节省内存，我们往往只将最后 N 个经验存入回放缓存（FIFO 缓存）。

第二个关键技术是**目标网络**。它作为一个独立的网络，用来代替所需的 Q 网络来生成 Q-Learning 的目标，进一步提高神经网络的稳定性。此外，目标网络每 C 步将通过直接复制（硬更新）或者指数衰减平均（软更新）的方式与主 Q 网络同步。目标网络通过使用旧参数生成 Q-Learning 目标，使目标值的产生不受最新参数的影响，从而大大减少发散和震荡的情况。例如，在动作 (S_t, A_t) 上的更新使得 Q 值增加，此时 S_t 和 S_{t+1} 的相似性可能会导致所有动作 a 的 $Q(S_{t+1}, a)$ 值增加，从而使得由 Q 网络产生的训练目标值被过估计。但是如果使用目标网络产生训练目标，就能避免过估计的问题。

这两项关键技术在 5 个雅达利游戏的效果提升效果如表 4.1 所示。智能体进行了 1e7 次配备超参数搜索功能的训练。每 250000 次训练，会对各个智能体进行 135000 帧评估，并且记录最高的片段平均分。

表 4.1　分别使用回放缓存和目标 Q 网络的效果。数据来自文献 (Mnih et al., 2015).

游戏名称	使用回放缓存和目标 Q 网络	使用回放缓存，且不使用目标 Q 网络	不使用回放缓存，使用目标 Q 网络	不使用回放缓存和目标 Q 网络
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

由于将任意长度的历史数据作为神经网络的输入较为复杂，DQN 转而处理由函数 ϕ 生成的

固定长度表示的历史数据。准确来说，ϕ 集合了当前帧和前三帧的数据，这对于跟踪时间相关信息（如对象的移动）非常有用。完整的算法展示在算法 4.17 中。其中原始帧被调整为 84×84 的灰度图像。函数 ϕ 堆叠了最近 4 帧的数据作为神经网络的输入。此外，神经网络的结构由三个卷积层和两个完全连接的层组成，每个有效动作只有一个输出。我们将在 4.8 节讨论更多的训练细节。

算法 4.17 DQN

超参数：回放缓存容量 N，奖励折扣因子 γ，用于目标状态-动作值函数更新的延迟步长 C，ϵ-greedy 中的 ϵ。
输入: 空回放缓存 $\mathcal{D}$，初始化状态-动作值函数 Q 的参数 θ。
使用参数 $\hat{\theta} \leftarrow \theta$ 初始化目标状态-动作值函数 $\hat{Q}$。
for 片段 $= 0, 1, 2, \cdots$ **do**
　初始化环境并获取观测数据 O_0。
　初始化序列 $S_0 = \{O_0\}$ 并对序列进行预处理 $\phi_0 = \phi(S_0)$。
　for t $= 0, 1, 2, \cdots$ **do**
　　通过概率 ϵ 选择一个随机动作 A_t，否则选择动作 $A_t = \arg\max_a Q(\phi(S_t), a; \theta)$。
　　执行动作 A_t 并获得观测数据 O_{t+1} 和奖励数据 R_t。
　　如果本局结束，则设置 $D_t = 1$，否则 $D_t = 0$。
　　设置 $S_{t+1} = \{S_t, A_t, O_{t+1}\}$ 并进行预处理 $\phi_{t+1} = \phi(S_{t+1})$。
　　存储状态转移数据 $(\phi_t, A_t, R_t, D_t, \phi_{t+1})$ 到 $\mathcal{D}$ 中。
　　从 $\mathcal{D}$ 中随机采样小批量状态转移数据 $(\phi_i, A_i, R_i, D_i, \phi'_i)$。
　　若 $D_i = 0$，则设置 $Y_i = R_i + \gamma \max_{a'} \hat{Q}(\phi'_i, a'; \hat{\theta})$，否则，设置 $Y_i = R_i$。
　　在 $(Y_i - Q(\phi_i, A_i; \theta))^2$ 上对 θ 执行梯度下降步骤。
　　每 C 步对目标网络 $\hat{Q}$ 进行同步。
　　如果片段结束，则跳出循环。
　end for
end for

4.4 Double DQN

Double DQN 是对 DQN 在减少过拟合方面的改进 (Van Hasselt et al., 2016)。在进一步讨论算法之前，我们先在经典的 DQN 算法上说明一下过拟合问题。我们注意到 Q-Learning 目标 $R_t + \gamma \max_a Q(S_{t+1}, a)$ 包含一个最大化算子 max 的操作。而 Q 又由于环境、非稳态、函数近似或者其他原因，可能带有噪声。需注意的是，最大噪声的期望值并不会小于噪声的最大期望，即 $\mathbb{E}[\max(\epsilon_1, \cdots, \epsilon_n)] \geqslant (\max(\mathbb{E}[\epsilon_1], \cdots, \mathbb{E}[\epsilon_n]))$。因此，下一个 Q 值往往被过估计了。文献 (Thrun et al., 1993) 对此提供了进一步的理论分析和实验结果。

通过增加对网络参数 θ 的关注，标准 DQN 的学习目标可以被重写为如下式子：

$$R_t + \gamma \hat{Q}(S_{t+1}, \arg\max_a \hat{Q}(S_{t+1}, a; \hat{\theta}); \hat{\theta}), \tag{4.9}$$

在式子中可以注意到一个问题：$\hat{\theta}$ 既用于估计 Q 值，又用于对估计过程中的下一个动作 a 进行选择。而 Double DQN 的核心思想是在这两个阶段使用两个不同的网络，以去除选择和评价中噪声的相关性。因此，需要一个额外的网络完成这项工作，而 DQN 结构中的 Q 网络则是一个很自然能想到的选择。（回顾一下 DQN 结构中有 Q 网络和目标网络这两个网络，并通过目标网络进行评估来进一步提高稳定性。）因此，Double DQN 中使用的 Q 学习目标是

$$R_t + \gamma \hat{Q}(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta); \hat{\theta}). \tag{4.10}$$

在 Wang 等人 (Wang et al., 2016) 的工作之上，我们通过如下公式计算智能体分数相对人类和基准智能体分数的提升百分比（有正有负）：

$$\frac{\text{Score}_{\text{Agent}} - \text{Score}_{\text{Baseline}}}{\max(\text{Score}_{\text{Baseline}}, \text{Score}_{\text{Human}}) - \text{Score}_{\text{Random}}} \tag{4.11}$$

Double DQN 相比于 DQN 的效果提升情况如图 4.1 所示。

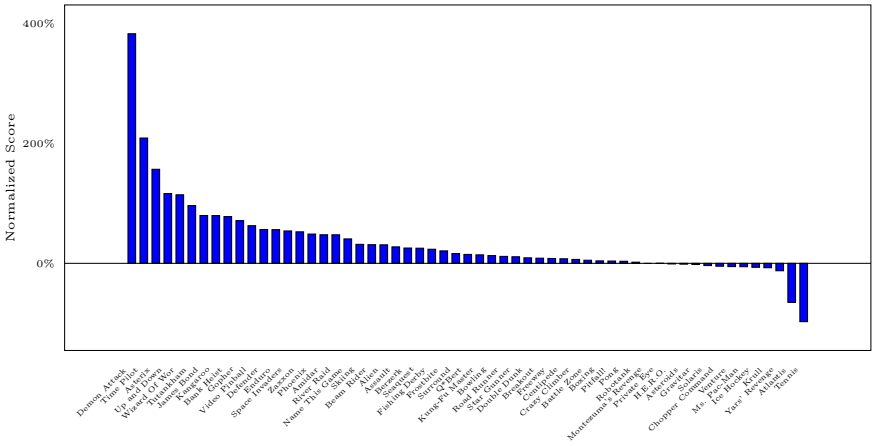

图 4.1 Double DQN (Van Hasselt et al., 2016) 相比于 DQN (Mnih et al., 2015) 在雅达利基准上的效果提升情况。计算标准参考公式 (4.11)。所有数据来自文献 (Wang et al., 2016)

。

4.5 Dueling DQN

对于某些状态来说，不同的动作与预期值无关，因此我们不需要学习各个动作对该状态的影响。例如，假想我们在山上看日出，美丽的景色令人陶醉，这是一个很高的奖励。此时，你即使在这里继续做不同的动作也不会对 Q 值产生影响。因此，将动作无关的状态值与 Q 值进行解耦，可以获得更加鲁棒的学习效果。

Dueling DQN 提出了一种新的网络结构来实现这一思想 (Wang et al., 2016)。更准确地说，Q 值可以被分为状态值和动作优势这两部分：

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a) \tag{4.12}$$

然后，Dueling DQN 通过如下方法将这两部分的表示分开：

$$Q(s,a;\theta,\theta_v,\theta_a) = V(s;\theta,\theta_v) + (A(s,a;\theta,\theta_a) - \max_{a'} A(s,a';\theta,\theta_a)) \tag{4.13}$$

其中 θ_v 和 θ_a 是两个全连接层的参数，θ 表示卷积层的参数。注意公式 (4.13) 中的 max 函数确保了 Q 值能唯一地对应状态值和动作优势。否则，训练将忽略状态值项，并只会使优势函数收敛到 Q 值。此外，文献 (Wang et al., 2016) 还提出使用取平均代替取最大值的方法，以获得更好的稳定性：

$$Q(s,a;\theta,\theta_v,\theta_a) = V(s;\theta,\theta_v) + (A(s,a;\theta,\theta_a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\theta_a)) \tag{4.14}$$

其中，优势函数只需要向平均优势方向靠近，而不必追求最大优势。

训练 Dueling 结构和训练标准 DQN 一样，它只需要更多的网络层。实验表明，Dueling 结构在许多价值相似的动作中，能获得更好的策略评估效果。Dueling DQN 相比于 DQN 的效果提升效果如图 4.2 所示。

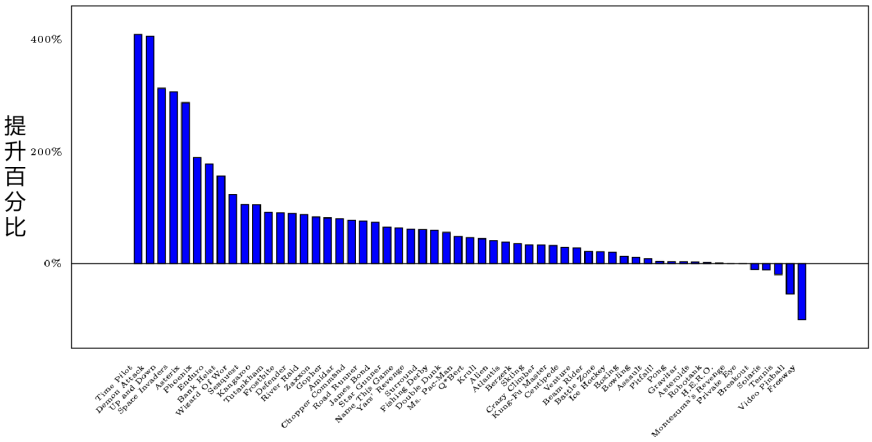

图 4.2　Dueling DQN (Wang et al., 2016) 相比于 DQN (Mnih et al., 2015) 在雅达利基准上的效果提升，计算标准参考公式 (4.11)。所有数据来自文献 (Wang et al., 2016)

4.6 优先经验回放

标准 DQN 中还剩下的一个可改进的地方就是，使用更好经验回放采样策略。**优先经验回放**（Prioritized Experience Replay，PER）是一种将经验进行优先排序的技术。通过该技术可以使重要的状态转移经验被更加频繁地回放 (Schaul et al., 2015)。PER 的核心思想是通过 TD 误差 δ 来考虑不同状态转移数据的重要性。TD 误差 δ 是一个令人惊喜的衡量标准。该方法之所以能有效，是由于某些经验数据相较于其他经验数据，可能包含更多值得学习的信息，所以给予这些包含更丰富信息量的经验更多的回放机会，有助于使得整个学习进度更为快速和高效。

当然，直接使用 TD 误差做优先排序是最直接能想到的方法。然而这种方法有一些问题。首先，扫描整个回放缓存空间非常低效。其次，这种方法对近似误差和随机回报的噪声十分敏感。最后，这种贪心的方法会使误差收敛缓慢，可能导致刚开始训练时有着高误差的状态转移被频繁地回放。为了克服这些问题，文献 (Schaul et al., 2015) 提出了使用如下方法计算状态转移 i 的采样概率：

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{4.15}$$

其中，p_i 指状态转移 i 的优先级，它是一个正数，即 $p_i > 0$。α 是一个指数超参数，$\alpha = 0$ 对应均匀采样情况，而 k 表示对采样的状态转移进行枚举。p_i 有两种变体。第一种是按比例优先：$p_i = |\delta_i| + \epsilon$。其中 δ_i 是状态转移 i 的 TD 误差，而 ϵ 是一个用于数值稳定的小正数。第二种变体是基于顺序的优先：$p_i = \frac{1}{\text{rank}(i)}$。其中 $\text{rank}(i)$ 是状态转移 i 基于 $|\delta_i|$ 的等级评定。

回想起在回放缓存中，正是因为随机采样而有助于消除样本之间的相关性的。然而在使用优先采样时，又放弃了纯随机采样。因此，减少高优先级状态转移数据的训练权重也有一定的道理。PER 使用了重要性采样（Importance-Sampling）权重来修正状态转移 i 的偏差。

$$w_i = (NP(i))^{-\beta} \tag{4.16}$$

其中，N 指回放缓存的容量大小，而 P 是按照公式 (4.15) 定义的概率。β 是训练过程中将会退火（Anneal）[2]到 1 的超参数，这么设置是由于随着训练增加，更新会趋近于无偏。此权重通常被折叠进损失函数来构造加权学习。

为了更有效地实现上述方法，我们将使用一个分段线性函数逼近采样概率的累积密度函数，该函数具有 k 段。更准确地说，优先级存储在一个称为线段树的高效查询数据结构中。在运行期间，首先对线段范围进行采样，然后在该线段范围内的样本进行均匀采样。对 DQN 的改进如图 4.3 所示。

[2]指模拟退火法中的退火，一种简单的实现是线性退火，例如，若设置初始值 0.6，终止值 1.0，最大迭代步数为 100，则第 $0 \leqslant t < 100$ 步取 $\beta = 0.6 + t(1.0 - 0.6)/99$

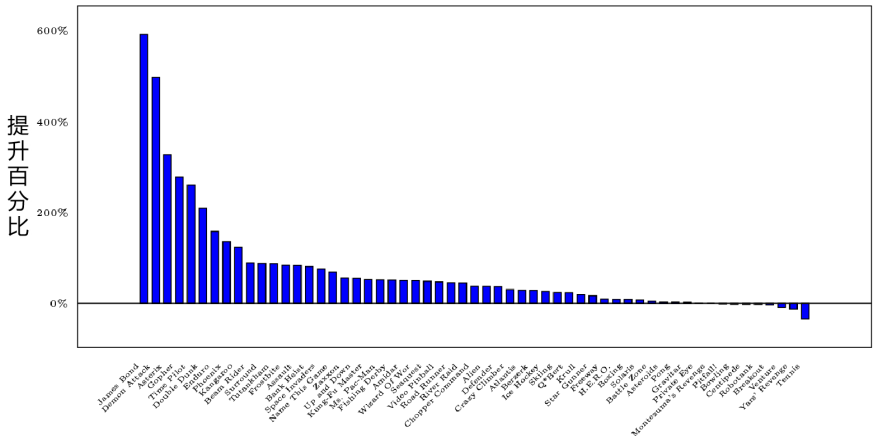

图 4.3　使用基于等级优先排序的优先经验回放算法 (Schaul et al., 2015) 相比于 DQN (Mnih et al., 2015) 在雅达利基准上的效果提升，计算标准参考公式 (4.11)。所有数据来自文献 (Wang et al., 2016)

4.7　其他改进内容：多步学习、噪声网络和值分布强化学习

Rainbow 在包含 Double Q-Learning、Dueling 结构和 PER 之外，还包含了另外 3 个 DQN 的扩展，并在雅达利游戏上取得了显著的成果 (Hessel et al., 2018)。在本节中，我们将对此展开讨论，并进一步讨论它们的延伸内容。

第一个扩展是多步学习（Multi-Step Learning）。使用 n 步回报将使估计更加准确，也被证明可以通过适当调整 n 值来加快学习速度 (Sutton et al., 2018)。然而，在离线策略学习过程中，目标策略和行为策略在多个步骤中的行为选择可能并不匹配。我们可以在文献 (Hernandez-Garcia et al., 2019) 中找到一个系统性的研究方法来纠正此类错配问题。Rainbow 直接使用了来自给定状态 S_t 的截断的 n 步回报 $R_t^{(k)}$ (Castro et al., 2018; Hessel et al., 2018)，其中 $R_t^{(k)}$ 由以下公式定义。

$$R_t^{(k)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k} \tag{4.17}$$

接着，Q-Learning 多步学习变体的目标通过下式定义。

$$R_t^{(k)} + \gamma^k \max_a Q(S_{t+k}, a) \tag{4.18}$$

第二个扩展是噪声网络 (Fortunato et al., 2017)。它是另一种 ϵ 贪心的探索算法，对于像《蒙特祖玛的复仇》这样需要大量探索的游戏十分有效。我们使用一个额外的噪声流将噪声加入线性层 $\boldsymbol{y} = (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ 中。

$$\boldsymbol{y} = (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + ((\boldsymbol{W}_{\text{noisy}} \odot \epsilon_w)\boldsymbol{x} + \boldsymbol{b}_{\text{noisy}} \odot \epsilon_b) \tag{4.19}$$

其中，$\odot$ 表示元素间的乘积，$\boldsymbol{W}_{\text{noisy}}$ 和 $\boldsymbol{b}_{\text{noisy}}$ 都是可训练的参数，而 ϵ_w 和 ϵ_b 是将退火到 0 的随机的标量。实验表明，噪声网络相比于许多基线算法，使得众多雅达利游戏的得分有了大幅提升。

最后一个扩展是值分布强化学习 (Bellemare et al., 2017)。该方法为值估计提供了一个新的视角。文献 (Bellemare et al., 2017) 提出了分布式贝尔曼算子 $\mathcal{T}^\pi$ 用于估计回报 Z 的分布，以改进过去只考虑 Z 的期望的做法:

$$\mathcal{T}^\pi Z = R + \gamma P^\pi Z. \tag{4.20}$$

图 4.4 展示了 $\mathcal{T}^\pi$ 的一种连续分布的情况。

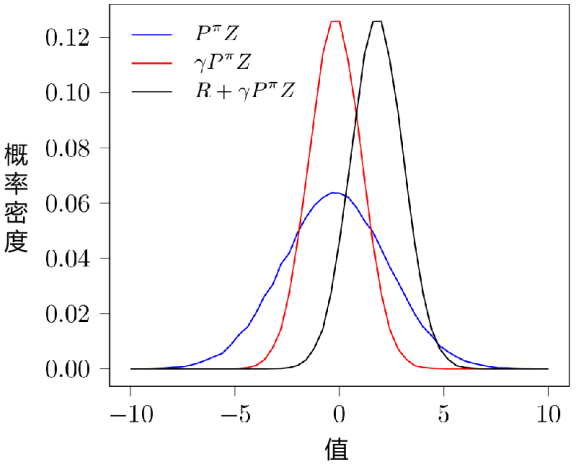

图 4.4 一种分布式贝尔曼算子在连续分布上的情况。它提供了在策略 π 下，下个状态的回报分布。它将先被折扣因子 γ 折损，然后被当前时间步中的奖励移动

Rainbow 中使用的值分布 DQN 变体被称为离散 DQN (Bellemare et al., 2017)，它通过一个离散分布来对状态-动作值分布进行建模，该分布由一个有 N 个元素（也被称为原子）的向量 $\boldsymbol{z}$ 参数化而来。该向量表示为 $z_i = V_{\min} + (i-1)\Delta z$，其中 $[V_{\min}, V_{\max}]$ 是状态-动作值分布的范围，并且 $\Delta z = \frac{V_{\max} - V_{\min}}{N-1}$。在实践中，$N$ 值通常设置为 51，因此，有时该算法也被称为 **C51**。C51 的参数模型 θ 输出每个原子概率 $p_i(s,a) = \mathrm{e}^{\theta_i(s,a)} / \sum_j \mathrm{e}^{\theta_j(s,a)}$ 组成分布 Z_θ。其中值得注意的是，离散化的近似会导致贝尔曼更新 $\mathcal{T}^\pi Z$ 与参数化的 Z_θ 脱节。而 C51 通过将目标分布 $\mathcal{T}^\pi Z_{\hat{\theta}}$ 投影到 Z_θ 上来解决这个问题。更加准确地说，若给定一个转移数据 (S_t, A_t, R_t, S_{t+1})，则使用 Double Q-Learning 的投影目标 $\Phi\mathcal{T}^\pi Z_{\hat{\theta}}(S_t, A_t)$ 的第 i 个分量由以下公式算出：

$$\sum_{j=1}^{N} p_j(S_{t+1}, \arg\max_a \boldsymbol{z}^\intercal p(S_{t+1}, a; \theta); \hat{\theta})[1 - \frac{|[R_t + \gamma z_j]_{V_{\min}}^{V_{\max}} - z_i|}{\Delta z}]_0^1 \tag{4.21}$$

其中，$[\cdot]_a^b$ 将其参数限制在 $[a, b]$ 范围内。由于 TD 误差无法度量值分布之间的差异，因此 C51 提出使用如下的 Kullbeck-Leibler 散度作为训练损失：

$$D_{\mathrm{KL}}(\Phi \mathcal{T}^\pi Z_{\hat{\theta}}(S_t, A_t) \| Z_\theta(S_t, A_t)). \tag{4.22}$$

另外，用于经验回放的优先级也被 KL 散度所代替。对于 Dueling 结构，输出分布也将分为价值数据流和优势数据流，并且总分布估计如下所示：

$$p_i(s, a) = \frac{\exp(V_i(s) + A_i(s, a) - \bar{A}_i(s, a))}{\sum_j \exp(V_j(s) + A_j(s, a) - \bar{A}_j(s, a))} \tag{4.23}$$

其中 $\bar{A}_j(s, a)$ 由 $\frac{1}{|\mathcal{A}|} \sum_{a'} A_j(s, a')$ 定义。

通过 C51 实现的值分布强化学习的主要缺点是，它只能在一个固定的离散集上估计值。文献 (Dabney et al., 2018b) 提出了**分位数回归 DQN**（Quantile Regression DQN，QR-DQN），通过分位数回归估计完整分布的分位数来解决这个问题。在介绍 QR-DQN 之前，我们先来看看这个分位数回归（Quantile Regression）。回想一下，对绝对损失函数进行经验风险最小化，能使预测符合中值（50% 分位数）。具体来说，给定随机变量 x 及其标签 y，对于估计函数 f，经验平均绝对误差为 $\mathcal{L}_{\mathrm{mae}} = \mathbb{E}[|f(x) - y|]$。接着用如下的偏微分：

$$\begin{aligned} \frac{\partial \mathcal{L}_{\mathrm{mae}}}{\partial f(x)} &= \frac{\partial}{\partial f(x)}(P(f(x) > y)(f(x) - y) + P(f(x) \leqslant y)(y - f(x))) \\ &= P(f(x) > y) - P(f(x) \leqslant y) = 0, \end{aligned} \tag{4.24}$$

我们能得到 $F(x) = 0.5$，其中 F 是 f 的原函数。通常来说，对于分位数 τ，其分位数损失定义为 $\mathcal{L}_{\mathrm{quantile}}(\tau) = \mathbb{E}[\rho_\tau(f(x) - y)]$，其中

$$\rho_\tau(\alpha) = \begin{cases} \tau\alpha, & \text{若} \alpha > 0 \\ (\tau - 1)\alpha, & \text{其他} \end{cases} \tag{4.25}$$

与之类似，通过 $\frac{\partial \mathcal{L}_{\mathrm{quantile}}}{\partial f(x)}$，我们能得到 $F(x) = 1 - \tau$，即 $f(x)$ 是随机变量 y 的 τ 分位数值。

具体来说，QR-DQN 考虑将 N 个均匀的分位数 $q_i = \frac{1}{N}$ 作为值分布。对于一个 QR-DQN 模型 $\theta : \mathcal{S} \to \mathbb{R}^{N \times |\mathcal{A}|}$，在采样期间，$Q$ 值的状态 s 和动作 a 是 N 个估计的平均值：$Q(s, a) = \sum_{i=1}^{N} q_i \theta_i(s, a)$。在训练过程中，基于 Q 值的贪心策略在下个状态提供 $a^* = \arg\max_{a'} Q(s', a')$，并且根据公式 (4.20)，分布式贝尔曼目标为 $\mathcal{T}\theta_j = r + \gamma\theta_j(s', a^*)$。文献 (Dabney et al., 2018b) 中

的引理 2 指出下式的和可以最小化近似值分布与真实值之间的 1-Wasserstein 距离：

$$\sum_{i=1}^{N} \mathbb{E}_j[\rho_{\hat{\tau}_i}(\mathcal{T}\theta_j - \theta_i(s,a))]. \tag{4.26}$$

其中 $\hat{\tau}_i = \frac{i}{N} - \frac{1}{2N}$。

图 4.5 展示了 DQN、C51 和 QR-DQN 的对比。接下来在值分布强化学习上，其参数化分布的灵活性和鲁棒性上还有更多的工作要做。读者对这方面感兴趣的话可以从文献 (Dabney et al., 2018a; Mavrin et al., 2019; Yang et al., 2019) 中找到相关资源。

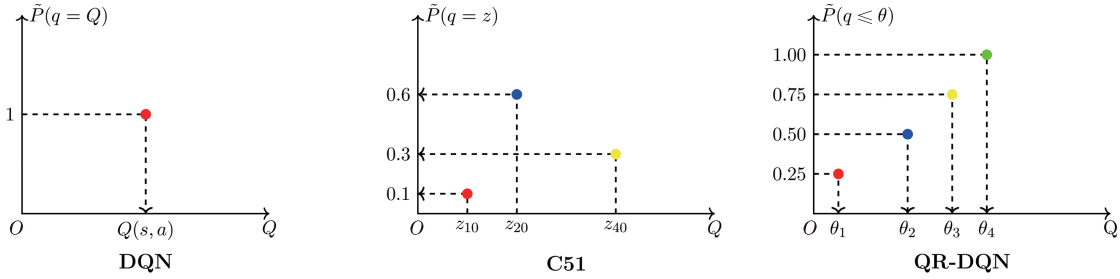

图 4.5 对比 s 和动作 a 下的 DQN，C51 和 QR-DQN。其中箭头指向的是估计值。QR-DQN 中分位数的数量指定为 4。DQN 的结构只输出实际 Q 值的近似值。对于值分布强化学习，C51 估计了多个 Q 值，而 QR-DQN 提供了 Q 值的分位数

4.8 DQN 代码实例

本节中，我们将围绕 DQN 及其变体算法讨论更多训练细节。首先演示雅达利环境的设置过程，以及如何实现一些十分有用的装饰器（Wrapper）。高效地使用装饰器能使训练更加简单和稳定。

Gym 环境相关

OpenAI Gym 是一个用于开发和对比强化学习算法的开源工具包。它包含了如图 4.6 显示的一系列环境。它可以直接从 PyPI 安装，默认安装包不带有雅达利组件，需要使用雅达利扩展安装：

```
pip install gym[atari]
```

也可以直接从源安装。

```
git clone https://github.com/openai/gym.git
cd gym
```

```
pip install -e .
```

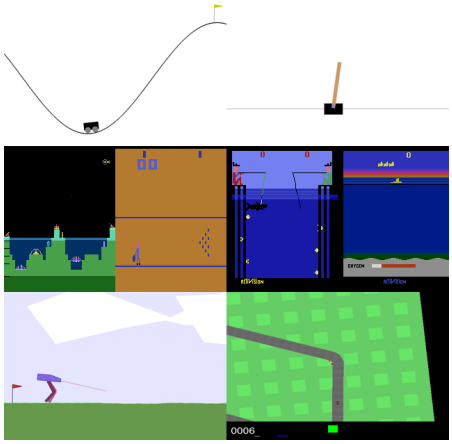

图 4.6　OpenAI Gym 的一些环境

可以通过以下代码建立环境实例 env：

```
import gym
env = gym.make(env_id)
```

其中 env_id 是环境名称的字符串。所有可用的 env_id 可以在网址（链接见读者服务）上查到。

env 实例中有以下重要的方法：

1. env.reset() 重启环境并返回初始的观测数据。
2. env.render(mode) 根据所给的 mode 模式呈现环境图像。默认为 human 模式，它将呈现当前显示画面或者终端窗口，并不返回任何内容。你可以指定 rgb_array 模式来使 env.render 函数返回 numpy.ndarray 对象，这些数据可用于生成视频。
3. env.step(action) 在环境中执行动作 action，并运行一个时间步。之后返回 (observation, reward, done, info) 的数据元组，其中 observation 为当前环境的观测数据，reward 是状态转移的奖励，done 指出当前片段是否结束，info 则包含一些辅助信息。
4. env.seed(seed) 手动设置随机种子。该函数在复现效果时非常有用。

这里展示了一个经典游戏 Breakout（打砖块）的例子。我们将先运行一个 BreakoutNoFrameskip-v4 环境的实例直到本片段结束。游戏过程的一个样帧图像如图 4.7 所示。

```
import gym
env = gym.make('BreakoutNoFrameskip-v4')
```

```
o = env.reset()
while True:
    env.render()
    # take a random action
    a = env.action_space.sample()
    o, r, done, _ = env.step(a)
    if done:
        break
env.close() # close and clean up
```

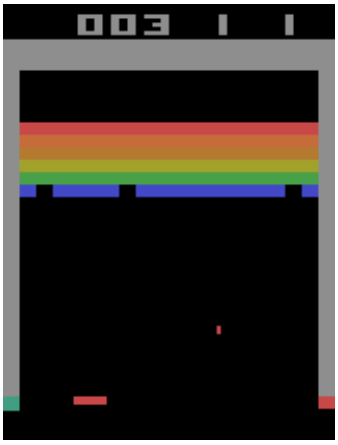

图 4.7 Breakout 游戏的一个样帧图像。在屏幕上方有几行需要被破坏的砖块。智能体可以控制屏幕下方的挡板，并控制角度弹射小球到想要的位置来撞毁砖块。该游戏的观测数据是形状为 $(210, 160, 3)$ 的 RGB 屏幕图像

需要注意的是，游戏 id 中的 `NoFrameskip` 意味着没有跳帧和动作重复，而 `v4` 意思是当前为第 4 个版本，也是本书写稿时的最新版本。我们将在接下来的例子中使用该环境。

OpenAI Gym 的另一个十分有用的特性是环境装饰器。它可以对环境对象进行装饰，使训练代码更加简洁。如下代码展示了一个用于限制每个回合片段最大长度的时间限制装饰器，这也是雅达利游戏的一个默认装饰器。

```
class TimeLimit(gym.Wrapper):
    def __init__(self, env, max_episode_steps=None):
        super(TimeLimit, self).__init__(env)
        self._max_episode_steps = max_episode_steps
        self._elapsed_steps = 0
```

```
    def step(self, ac):
        o, r, done, info = self.env.step(ac)
        self._elapsed_steps += 1
        if self._elapsed_steps >= self._max_episode_steps:
            done = True
            info['TimeLimit.truncated'] = True
        return o, r, done, info

    def reset(self, **kwargs):
        self._elapsed_steps = 0
        return self.env.reset(**kwargs)
```

为了更加高效地训练，`gym.vector.AsyncVectorEnv` 提供了一个用来并行运行 n 个环境的矢量化装饰器的实现。所有的接口将统一收到并返回 n 个变量。此外，还可以实现一个带有缓存的矢量化装饰器，其接口也接受和返回 n 个变量，但会在后台保持 $m > n$ 个线程。这样将更为高效地运行某些状态转移耗时较长的环境。

Gym 提供一系列雅达利 2600 游戏的标准接口。这些游戏可以以游戏内存数据或者屏幕图像数据作为输入，使用街机学习环境 (Bellemare et al., 2013) 运行。在这 2600 款雅达利游戏中，有些游戏最多包含 18 个不同的按键组合：

1. 移动按键：空动作、上移、右移、左移、下移、右上键组合、左上键组合、右下键组合、左下键组合。
2. 攻击按键：开火、上移开火组合、右移开火组合、左移开火组合、下移开火组合、右上开火组合、左上开火组合、右下开火组合、左下开火组合。

此处的空动作表示什么都不做。然后开火键可能被作为开始游戏的按键。为了方便起见，我们后续将以按键名称称呼其对应的动作。

DQN

DQN 还有三个额外的训练技巧。首先，依次使用如下的装饰器可以让训练更加稳定高效。

1. `NoopResetEnv` 在重置游戏时，会随机地进行几步空动作，以确保初始化的状态更为随机。默认的最大空动作数量为 30。这个装饰器将有助于智能体收集更多的初始状态，提供更为鲁棒的学习。
2. `MaxAndSkipEnv` 重复每个动作 4 次，以提供更为高效的学习。为了进一步对观测数据降噪，返回的图像帧是在最近 2 帧上对像素进行最大池化的结果。
3. `Monitor` 记录原始奖励数据。我们可以在这个装饰器中实现一些有用的函数，比如速度跟踪器。

4. EpisodicLifeEnv 使得本条命结束的时候，相当于本片段结束。这样不用等到玩家所有命都消耗完才能结束本片段，对价值估计很有帮助 (Roderick et al., 2017)。
5. FireResetEnv 在环境重置的时候触发开火动作。很多游戏需要这个开火动作来开始游戏。这是快速开始游戏的先验知识。
6. WarpFrame 将观测数据转换为 84×84 的灰度图像。
7. ClipRewardEnv 将奖励通过符号进行装饰，只根据奖励数据的符号输出 -1、0、1 三种奖励值。这样防止任何一个单独的小批量更新而大幅改变参数，可以进一步提高稳定性。
8. FrameStack 堆叠最后 4 帧。我们回忆一下，DQN 为了捕捉运动信息，通过堆叠当前帧和前 3 帧来用函数 ϕ 对观测数据进行预处理。FrameStack 和 WarpFrame 实现了 ϕ 的功能。需要注意的是，我们可以通过只在观测值之间存储一次公共帧来优化内存使用，这也称为延迟帧技术（Lazy-Frame Trick）。

其次，为避免梯度爆炸，DQN (DeepMind, 2015; Mnih et al., 2015) 使用了对平方误差进行了裁剪，这等同于将均方差替换成了 $\delta = 1$ 情况下的 Huber 损失 (Huber, 1992)。Huber 损失如下所示：

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & |x| \leqslant \delta \\ \delta\left(|x| - \frac{1}{2}\delta\right) & \text{其他} \end{cases} \tag{4.27}$$

最终，回放缓存采样了大批有放回的抽样。在能够有个稳定的开始之前，最后还需要完成一些热启动步骤。

注意到上述所说的全部三个技巧都用于本节中所有的实验。现在我们将展示如何建立一个能玩 Breakout 游戏的智能体。首先，为了实验的可复现性，我们将手动设置相关库的随机种子。

```
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

接着，我们通过 tf.keras.Model 创建一个 Q 网络：

```
class QFunc(tf.keras.Model):
    def __init__(self, name):
        super(QFunc, self).__init__(name=name)
        self.conv1 = tf.keras.layers.Conv2D(
            32, kernel_size=(8, 8), strides=(4, 4),
            padding='valid', activation='relu')
        self.conv2 = tf.keras.layers.Conv2D(
            64, kernel_size=(4, 4), strides=(2, 2),
```

```
            padding='valid', activation='relu')
        self.conv3 = tf.keras.layers.Conv2D(
            64, kernel_size=(3, 3), strides=(1, 1),
            padding='valid', activation='relu')
        self.flat = tf.keras.layers.Flatten()
        self.fc1 = tf.keras.layers.Dense(512, activation='relu')
        self.fc2 = tf.keras.layers.Dense(action_dim, activation='linear')

    def call(self, pixels, **kwargs):
        # scale observation
        pixels = tf.divide(tf.cast(pixels, tf.float32), tf.constant(255.0))
        # extract features by convolutional layers
        feature = self.flat(self.conv3(self.conv2(self.conv1(pixels))))
        # calculate q-value
        qvalue = self.fc2(self.fc1(feature))

        return qvalue
```

DQN 对象的定义由 Q 网络、目标 Q 网络、训练时间步数目和优化器、同步 Q 网络、目标 Q 网络这些属性组成，代码如下所示。

```
class DQN(object):
    def __init__(self):
        self.qnet = QFunc('q')
        self.targetqnet = QFunc('targetq')
        sync(self.qnet, self.targetqnet)
        self.niter = 0
        self.optimizer = tf.optimizers.Adam(lr, epsilon=1e-5, clipnorm=clipnorm)
```

申明一个内部方法，以装饰 Q 网络，之后再给 DQN 对象添加一个 `get_action` 方法来执行 ϵ-贪心的行动。

```
@tf.function
def _qvalues_func(self, obv):
    return self.qnet(obv)

def get_action(self, obv):
    eps = epsilon(self.niter)
    if random.random() < eps:
```

```
        return int(random.random() * action_dim)
    else:
        obv = np.expand_dims(obv, 0).astype('float32')
        return self._qvalues_func(obv).numpy().argmax(1)[0]
```

其中，这里的 epsilon 函数是一个在前 10% 训练时间步中，将 ϵ 线性地从 1.0 退火到 0.01 的函数。为了更好地训练，我们为 DQN 及其变体提供了 3 个通用接口，即 train、_train_func、_tderror_func。

```
def train(self, b_o, b_a, b_r, b_o_, b_d):
    self._train_func(b_o, b_a, b_r, b_o_, b_d)

    self.niter += 1
    if self.niter
        sync(self.qnet, self.targetqnet)

@tf.function
def _train_func(self, b_o, b_a, b_r, b_o_, b_d):
    with tf.GradientTape() as tape:
        td_errors = self._tderror_func(b_o, b_a, b_r, b_o_, b_d)
        loss = tf.reduce_mean(huber_loss(td_errors))

    grad = tape.gradient(loss, self.qnet.trainable_weights)
    self.optimizer.apply_gradients(zip(grad, self.qnet.trainable_weights))

    return td_errors

@tf.function
def _tderror_func(self, b_o, b_a, b_r, b_o_, b_d):
    b_q_ = (1 - b_d) * tf.reduce_max(self.targetqnet(b_o_), 1)
    b_q = tf.reduce_sum(self.qnet(b_o) * tf.one_hot(b_a, action_dim), 1)

    return b_q - (b_r + reward_gamma * b_q_)
```

其中 train 调用了 _train_func 并每 target_q_update_freq 个时间步将目标 Q 网络与 Q 网络进行同步。

最终，我们构建主要训练步骤：

```
dqn = DQN()
buffer = ReplayBuffer(buffer_size)

o = env.reset()
nepisode = 0
t = time.time()
for i in range(1, number_time steps + 1):
    a = dqn.get_action(o)

    # execute action and feed to replay buffer
    # note that '_' tail in var name means next
    o_, r, done, info = env.step(a)
    buffer.add(o, a, r, o_, done)

    if i >= warm_start and i
        transitions = buffer.sample(batch_size)
        dqn.train(*transitions)

    if done:
        o = env.reset()
    else:
        o = o_

    # episode in info is real (unwrapped) message
    if info.get('episode'):
        nepisode += 1
        reward, length = info['episode']['r'], info['episode']['l']
        print(
            'Time steps so far: {}, episode so far: {}, '
            'episode reward: {:.4f}, episode length: {}'
                .format(i, nepisode, reward, length)
        )
```

我们在 3 个随机种子上运行了 Breakout 游戏 10^7 个时间步（4×10^7 帧）。为了更好地可视化，我们将训练时的片段奖励进行平滑处理。之后通过如下代码绘制均值和标准差，输出效果如图 4.8 所示的红色区域。

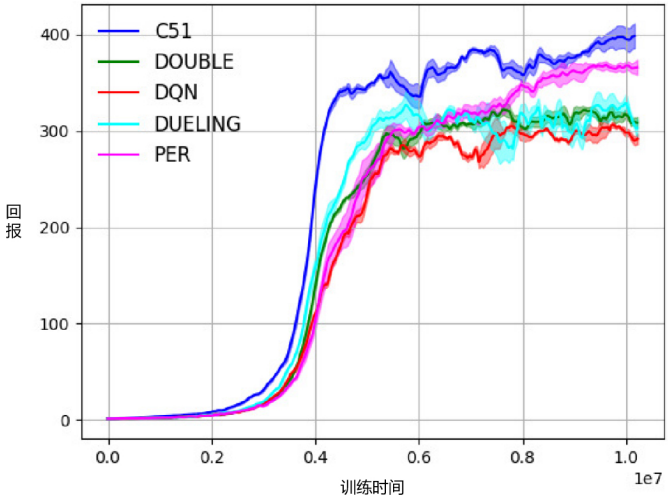

图 4.8 DQN 及其变体在 Breakout 游戏中的效果（见彩插）

```
from matplotlib import pyplot as plt
plt.plot(xs, mean, color=color)
plt.fill_between(xs, mean - std, mean + std, color=color, alpha=.4)
```

Double DQN

Double DQN 可以通过更新 Double Q 的估计来简单地实现。在智能体的 `_tderror_func` 中使用如下 Double Q 估计的代码进行替换即可。

```
# double Q estimation
b_a_ = tf.one_hot(tf.argmax(qnet(b_o_), 1), out_dim)
b_q_ = (1 - b_d) * tf.reduce_sum(targetqnet(b_o_) * b_a_, 1)
```

我们也在 Breakout 游戏上，使用 3 个随机种子运行了 10^7 个时间步。输出效果显示在图 4.8 上的绿色区域。

Dueling DQN

Dueling 架构只对 Q 网络进行了修改，它可以通过如下方式实现：

```
class QFunc(tf.keras.Model):
    def __init__(self, name):
        super(QFunc, self).__init__(name=name)
```

```
        self.conv1 = tf.keras.layers.Conv2D(
            32, kernel_size=(8, 8), strides=(4, 4),
            padding='valid', activation='relu')
        self.conv2 = tf.keras.layers.Conv2D(
            64, kernel_size=(4, 4), strides=(2, 2),
            padding='valid', activation='relu')
        self.conv3 = tf.keras.layers.Conv2D(
            64, kernel_size=(3, 3), strides=(1, 1),
            padding='valid', activation='relu')
        self.flat = tf.keras.layers.Flatten()
        self.fc1q = tf.keras.layers.Dense(512, activation='relu')
        self.fc2q = tf.keras.layers.Dense(action_dim, activation='linear')
        self.fc1v = tf.keras.layers.Dense(512, activation='relu')
        self.fc2v = tf.keras.layers.Dense(1, activation='linear')

    def call(self, pixels, **kwargs):
        # scale observation
        pixels = tf.divide(tf.cast(pixels, tf.float32), tf.constant(255.0))
        # extract features by convolutional layers
        feature = self.flat(self.conv3(self.conv2(self.conv1(pixels))))
        # calculate q-value
        qvalue = self.fc2q(self.fc1q(feature))
        svalue = self.fc2v(self.fc1v(feature))

        return svalue + qvalue - tf.reduce_mean(qvalue, 1, keepdims=True)
```

我们同样在 Breakout 游戏上，使用 3 个随机种子运行了 10^7 个时间步。在图 4.8 上的青色区域是该方法的输出效果。

经验优先回放

PER 相较于标准的 DQN 有三个变化。首先，回放缓存维持了 2 个线段树进行取小和求和操作，来高效地计算最小优先级和优先级之和。更具体地说，`_it_sum` 属性是具备两个接口的求和操作线段树对象，`sum` 用于获得指定区间内的元素之和，而 `find_prefixsum_idx` 用于查找更高的索引 i，以使最小的 i 个元素比输入值要小。

其次，为了代替原本的均匀采样，考虑比例信息的采样策略如下所示：

```
res = []
p_total = self._it_sum.sum(0, len(self._storage) - 1)
every_range_len = p_total / batch_size
for i in range(batch_size):
    mass = random.random() * every_range_len + i * every_range_len
    idx = self._it_sum.find_prefixsum_idx(mass)
    res.append(idx)
return res
```

最后，不同于普通的回放缓存，PER 必须返回采样经验的索引和标准化的权重。权重用于计算加权 Huber 损失，而索引则用于更新优先级。采样步骤将被修改为

```
*transitions, idxs = buffer.sample(batch_size)
priorities = dqn.train(*transitions)
priorities = np.clip(np.abs(priorities), 1e-6, None)
buffer.update_priorities(idxs, priorities)
```

`_train_func` 可修改为

```
@tf.function
def _train_func(self, b_o, b_a, b_r, b_o_, b_d, b_w):
    with tf.GradientTape() as tape:
        td_errors = self._tderror_func(b_o, b_a, b_r, b_o_, b_d)
        loss = tf.reduce_mean(huber_loss(td_errors) * b_w)

    grad = tape.gradient(loss, self.qnet.trainable_weights)
    self.optimizer.apply_gradients(zip(grad, self.qnet.trainable_weights))

    return td_errors
```

我们还是在 Breakout 游戏上，使用 3 个随机种子运行了 10^7 个时间步。图 4.8 上的洋红色区域是该方法的输出效果。

深度 Q 分布网络

值分布强化学习对 Q 值进行估计。在本节中，我们将通过演示如何实现其中的 C51 技术，来实现一种值分布强化学习方法。在 Breakout 游戏中，奖励都是正数。因此，我们将文献 (Bellemare et al., 2017) 中值的范围 $[-10, 10]$ 换成 $[-1, 19]$，其中 -1 是为了允许一些近似误差。实现 C51 首

先要做的是让 Q 网络给每个动作输出 51 个估计值，这点可以通过在最后的全连接层增加更多的输出单元来实现。接着，为了替代 TD 误差，需要使用目标 Q 分布和估计分布之间的 KL 散度作为误差：

```
@tf.function
def _kl_divergence_func(self, b_o, b_a, b_r, b_o_, b_d):
    b_r = tf.tile(
        tf.reshape(b_r, [-1, 1]),
        tf.constant([1, atom_num])
    ) # batch_size * atom_num
    b_d = tf.tile(
        tf.reshape(b_d, [-1, 1]),
        tf.constant([1, atom_num])
    )

    z = b_r + (1 - b_d) * reward_gamma * vrange # shift value distribution
    z = tf.clip_by_value(z, min_value, max_value) # clip the shifted distribution
    b = (z - min_value) / deltaz
    index_help = tf.expand_dims(tf.tile(
        tf.reshape(tf.range(batch_size), [batch_size, 1]),
        tf.constant([1, atom_num])
    ), -1)

    b_u = tf.cast(tf.math.ceil(b), tf.int32) # upper
    b_uid = tf.concat([index_help, tf.expand_dims(b_u, -1)], 2) # indexes
    b_l = tf.cast(tf.math.floor(b), tf.int32)
    b_lid = tf.concat([index_help, tf.expand_dims(b_l, -1)], 2) # indexes

    b_dist_ = self.targetqnet(b_o_) # whole distribution
    b_q_ = tf.reduce_sum(b_dist_ * vrange_broadcast, axis=2)
    b_a_ = tf.cast(tf.argmax(b_q_, 1), tf.int32)
    b_adist_ = tf.gather_nd( # distribution of b_a_
        b_dist_,
        tf.concat([tf.reshape(tf.range(batch_size), [-1, 1]),
                   tf.reshape(b_a_, [-1, 1])], axis=1)
    )
    b_adist = tf.gather_nd( # distribution of b_a
        self.qnet(b_o),
```

```
        tf.concat([tf.reshape(tf.range(batch_size), [-1, 1]),
                   tf.reshape(b_a, [-1, 1])], axis=1)
    ) + 1e-8

    b_l = tf.cast(b_l, tf.float32)
    mu = b_adist_ * (b - b_l) * tf.math.log(tf.gather_nd(b_adist, b_uid))
    b_u = tf.cast(b_u, tf.float32)
    ml = b_adist_ * (b_u - b) * tf.math.log(tf.gather_nd(b_adist, b_lid))
    kl_divergence = tf.negative(tf.reduce_sum(mu + ml, axis=1))

    return kl_divergence
```

当然我们在 Breakout 游戏上，使用 3 个随机种子运行了 10^7 个时间步。图 4.8 上的蓝色区域是该方法的输出效果。

参考文献

BELLEMARE M G, NADDAF Y, VENESS J, et al., 2013. The Arcade Learning Environment: An evaluation platform for general agents[J]. Journal of Artificial Intelligence Research, 47: 253-279.

BELLEMARE M G, DABNEY W, MUNOS R, 2017. A distributional perspective on reinforcement learning[C]//Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org: 449-458.

CASTRO P S, MOITRA S, GELADA C, et al., 2018. Dopamine: A research framework for deep reinforcement learning[J].

DABNEY W, OSTROVSKI G, SILVER D, et al., 2018a. Implicit quantile networks for distributional reinforcement learning[C]//International Conference on Machine Learning. 1104-1113.

DABNEY W, ROWLAND M, BELLEMARE M G, et al., 2018b. Distributional reinforcement learning with quantile regression[C]//Thirty-Second AAAI Conference on Artificial Intelligence.

DEEPMIND, 2015. Lua/Torch implementation of DQN[J]. GitHub repository.

FORTUNATO M, AZAR M G, PIOT B, et al., 2017. Noisy networks for exploration[J]. arXiv preprint arXiv:1706.10295.

HERNANDEZ-GARCIA J F, SUTTON R S, 2019. Understanding multi-step deep reinforcement learning: A systematic study of the DQN target[C]//Proceedings of the Neural Information Processing Systems (Advances in Neural Information Processing Systems) Workshop.

HESSEL M, MODAYIL J, VAN HASSELT H, et al., 2018. Rainbow: Combining improvements in deep reinforcement learning[C]//Thirty-Second AAAI Conference on Artificial Intelligence.

HUBER P J, 1992. Robust estimation of a location parameter[M]//Breakthroughs in statistics. Springer: 492-518.

LIN L J, 1993. Reinforcement learning for robots using neural networks[R]. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

MAVRIN B, YAO H, KONG L, et al., 2019. Distributional reinforcement learning for efficient exploration[C]//International Conference on Machine Learning. 4424-4434.

MCCLELLAND J L, MCNAUGHTON B L, O'REILLY R C, 1995. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory.[J]. Psychological review, 102(3): 419.

MNIH V, KAVUKCUOGLU K, SILVER D, et al., 2015. Human-level control through deep reinforcement learning[J]. Nature.

O'NEILL J, PLEYDELL-BOUVERIE B, DUPRET D, et al., 2010. Play it again: reactivation of waking experience and memory[J]. Trends in neurosciences, 33(5): 220-229.

RIEDMILLER M, 2005. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method[C]//European Conference on Machine Learning. Springer: 317-328.

RODERICK M, MACGLASHAN J, TELLEX S, 2017. Implementing the deep Q-network[J]. arXiv preprint arXiv:1711.07478.

SCHAUL T, QUAN J, ANTONOGLOU I, et al., 2015. Prioritized experience replay[C]//arXiv preprint arXiv:1511.05952.

SUTTON R S, BARTO A G, 2018. Reinforcement learning: An introduction[M]. MIT press.

THRUN S, SCHWARTZ A, 1993. Issues in using function approximation for reinforcement learning[C]// Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum.

TSITSIKLIS J, VAN ROY B, 1996. An analysis of temporal-difference learning with function approximationtechnical[J]. Report LIDS-P-2322). Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Tech. Rep.

TSITSIKLIS J N, VAN ROY B, 1997. Analysis of temporal-diffference learning with function approximation[C]//Advances in Neural Information Processing Systems. 1075-1081.

VAN HASSELT H, GUEZ A, SILVER D, 2016. Deep reinforcement learning with double Q-learning[C]// Thirtieth AAAI conference on artificial intelligence.

WANG Z, SCHAUL T, HESSEL M, et al., 2016. Dueling network architectures for deep reinforcement learning[C]//International Conference on Machine Learning. 1995-2003.

YANG D, ZHAO L, LIN Z, et al., 2019. Fully parameterized quantile function for distributional reinforcement learning[C]//Advances in Neural Information Processing Systems. 6190-6199.