

第四章 价值学习与 DQN

本章的内容是价值学习的基础。第 4.1 用神经网络近似最优动作价值函数 $Q^*(s, a)$ ，把这个神经网络称为深度 Q 网络 (DQN)。本章内容的难点在于训练 DQN 所用的时间差分算法 (TD)。为了帮助读者理解 TD 算法，第 4.2 节以“驾车时间估计”举例讲解 TD 算法，然后在第 4.3 节将 DQN 类比成“驾车时间估计”，推导出训练 DQN 的算法。第 4.5 节解释同策略 (On-policy) 与异策略 (Off-policy) 的区别；本章训练 DQN 所用的 TD 算法属于异策略。

4.1 DQN

在学习 DQN 之前，首先复习一些基础知识。在一局 (Episode) 游戏中，把从起始到结束的所有奖励记作：

$$R_1, \dots, R_t, \dots, R_n.$$

定义折扣率 $\gamma \in [0, 1]$ 。折扣回报的定义是：

$$U_t = R_t + \gamma \cdot R_{t+1} + \gamma^2 \cdot R_{t+2} + \dots + \gamma^{n-t} \cdot R_n.$$

在游戏尚未结束的 t 时刻， U_t 是一个未知的随机变量，其中的随机性来自于 t 时刻之后的所有状态与动作。动作价值函数的定义是：

$$Q_\pi(s_t, a_t) = \mathbb{E}[U_t \mid S_t = s_t, A_t = a_t],$$

公式中的期望消除了 t 时刻之后的所有状态 S_{t+1}, \dots, S_n 与所有动作 A_{t+1}, \dots, A_n 。最优动作价值函数用最大化消除策略 π ：

$$Q_*(s_t, a_t) = \max_\pi Q_\pi(s_t, a_t), \quad \forall s_t \in \mathcal{S}, \quad a_t \in \mathcal{A}.$$

可以这样理解 Q_* ：已知 s_t 和 a_t ，不论未来采取什么样的策略 π ，回报 U_t 的期望不可能超过 Q_* 。

最优动作价值函数的用途：假如我们知道 Q_* ，我们就能用它做控制。举个例子，超级玛丽游戏中的动作空间是 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。给定当前状态 s_t ，智能体该执行哪个动作呢？假设我们已知 Q_* 函数，那么我们就让 Q_* 给三个动作打分，比如：

$$Q_*(s_t, \text{左}) = 370, \quad Q_*(s_t, \text{右}) = -21, \quad Q_*(s_t, \text{上}) = 610.$$

这三个值是什么意思呢？ $Q_*(s_t, \text{左}) = 130$ 的意思是：如果现在智能体选择向左走，不论之后采取什么策略 π ，那么回报 U_t 的期望最多不会超过 370。同理，其他两个最优动作价值的也是回报的期望的上界。根据 Q_* 的评分，智能体应该选择向上跳，因为这样可以最大化回报 U_t 的期望。

我们希望知道 Q_* ，因为它就像是先知一般，可以预见未来，在 t 时刻就预见 t 到 n 时刻之间的累计奖励的期望。假如我们有 Q_* 这位先知，我们就遵照按照先知的指导，最大化未来的累计奖励。然而在实践中我们无法得到 Q_* 的函数表达式。是否有可能近似

出 Q_* 这位先知呢？对于超级玛丽这样的游戏，学出来一个“先知”并不难。假如让我重复玩超级玛丽一亿次，那我就像是先知一样：告诉我当前状态，我能准确判断出当前最优的动作是什么。这说明只要有足够多的“经验”，就能训练出超级玛丽中的“先知”。

最优动作价值函数的近似：在实践中，近似学习“先知” Q_* 最有效的办法是深度 Q 网络 (Deep Q Network)，缩写是 DQN，记作 $Q(s, a; \mathbf{w})$ ，其结构在图 4.1 中描述。其中的 \mathbf{w} 表示神经网络中的参数；一开始随机初始化 \mathbf{w} ，随后用“经验”去学习 \mathbf{w} 。学习的目标是：对于所有的 s 和 a ，DQN 的预测 $Q(s, a; \mathbf{w})$ 尽量接近 $Q_*(s, a)$ 。之后几节的内容都是如何学习 \mathbf{w} 。

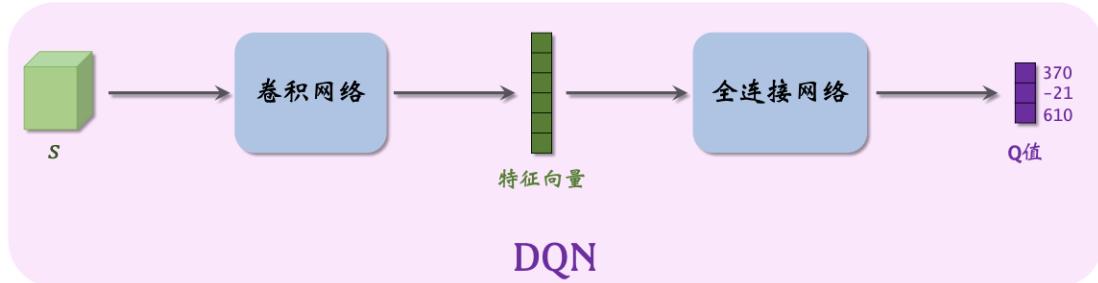


图 4.1：DQN 的神经网络结构。输入是状态 s ；输出是每个动作的 Q 值。。

可以这样理解 DQN 的表达式 $Q(s, a; \mathbf{w})$ 。DQN 的输出是离散动作空间 \mathcal{A} 上的每个动作的 Q 值（即给每个动作的评分，分数越高，动作越好）。举个例子，动作空间是 $\mathcal{A} = \{\text{左, 右, 上}\}$ ，那么动作空间的大小等于 $|\mathcal{A}| = 3$ ，DQN 的输出是 3 维的向量 $\hat{\mathbf{q}}$ ，向量每个元素对应一个动作。在图 4.1 中，DQN 的输出是

$$\hat{q}_1 = Q(s, \text{左}; \mathbf{w}) = 370,$$

$$\hat{q}_2 = Q(s, \text{右}; \mathbf{w}) = -21,$$

$$\hat{q}_3 = Q(s, \text{上}; \mathbf{w}) = 610.$$

总结一下，DQN 的输出是 $|\mathcal{A}|$ 维的向量 $\hat{\mathbf{q}}$ ，包含所有动作的价值。而我们常用的符号 $Q(s, a; \mathbf{w})$ 是标量，是动作 a 对应的动作价值。可以把动作 a 看做是一个序号，比如 $a = \text{“右”}$ 是 \mathcal{A} 中第二号动作，那么 $Q(s, a; \mathbf{w})$ 就是向量 $\hat{\mathbf{q}}$ 的第二号元素 $\hat{q}_2 = -21$ 。

DQN 的梯度：在训练 DQN 的时候，需要对 DQN 关于神经网络参数 \mathbf{w} 求梯度。用

$$\nabla_{\mathbf{w}} Q(s, a; \mathbf{w}) \triangleq \frac{\partial Q(s, a; \mathbf{w})}{\partial \mathbf{w}}$$

表示函数值 $Q(s, a; \mathbf{w})$ 关于参数 \mathbf{w} 的梯度。因为函数值 $Q(s, a; \mathbf{w})$ 是一个实数，所以梯度的形状与 \mathbf{w} 完全相同：如果 \mathbf{w} 是 $d \times 1$ 的向量，那么梯度也是 $d \times 1$ 的向量；如果 \mathbf{w} 是 $d_1 \times d_2$ 的矩阵，那么梯度也是 $d_1 \times d_2$ 的矩阵；如果 \mathbf{w} 是 $d_1 \times d_2 \times d_3$ 的张量，那么梯度也是 $d_1 \times d_2 \times d_3$ 的张量。

给定观测值 s 和 a （比如 $a = \text{“左”}$ ），可以用反向传播计算出梯度 $\nabla_{\mathbf{w}} Q(s, \text{“左”}; \mathbf{w})$ 。在编程实现的时候，TensorFlow 和 PyTorch 可以对 DQN 输出向量的一个元素（比如 $Q(s, \text{“左”}; \mathbf{w})$ ）关于变量 \mathbf{w} 自动求梯度，得到的梯度的形状与 \mathbf{w} 完全相同。

4.2 时间差分 (TD) 算法

训练 DQN 最常用的算法是时间差分 (Temporal Difference)，缩写 TD。为了帮助大家理解 TD 算法，本书用驾车时间类比 DQN。假设我们有一个模型 $Q(s, d; \mathbf{w})$ ，其中 s 是起点， d 是终点， \mathbf{w} 是参数，它可以预测出开车出行的时间开销。这个模型一开始不准确，甚至是纯随机的。但是随着很多人用这个模型，得到更多数据、更多训练，这个模型就会越来越准，会像谷歌地图一样准。我们该如何训练这个模型呢？

4.2.1 在线梯度下降算法

在介绍 TD 算法之前，我们先来学习一种更简单的算法——在线梯度下降 (Online Gradient Descent)——并用驾车时间的例子来讲解算法。

我要北京驾车去上海。从北京出发之前，我让模型做预测，模型告诉我总车程是 14 小时。

$$\hat{q} \triangleq Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) = 14.$$

当我到达上海，我知道自己在路上花的时间是 16 小时，并将结果反馈给模型；见图 4.2。



图 4.2：模型估计驾驶时间是 $Q(s, d; \mathbf{w}) = 14$ ，而实际花费时间 $y = 16$ 。

可以用在线梯度下降对模型做一次更新，具体做法如下。把我的这次旅程作为一组训练数据：

$$s = \text{“北京”}, \quad d = \text{“上海”}, \quad \hat{q} = 14, \quad y = 16.$$

我们希望估计值 \hat{q} 尽量接近真实值 y ，所以用两者的平方差作为损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [Q(s, d; \mathbf{w}) - y]^2.$$

用链式法则计算损失函数的梯度，得到：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = (\hat{q} - y) \cdot \nabla_{\mathbf{w}} Q(s, d; \mathbf{w}),$$

然后做一次梯度下降更新模型参数 \mathbf{w} ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \nabla_{\mathbf{w}} L(\mathbf{w}),$$

此处的 α 是学习率，需要手动调。如果现在再让模型做一次预测，那么模型的预测值

$$Q(\text{“北京”}, \text{“上海”}; \mathbf{w})$$

会比原先更接近 $y = 16$ 。

4.2.2 TD 算法

接着上文驾车时间的例子。出发前模型估计全程时间为 $\hat{q} = 14$ 小时；模型建议的路线会途径济南。我从北京出发，过了 $r = 4.5$ 小时，我到达济南。此时我再让模型做一次预测，模型告诉我

$$\hat{q}' \triangleq Q(\text{“济南”}, \text{“上海”}; \mathbf{w}) = 11.$$

见图 4.3 的描述。假如此时我的车坏了，必须要在济南修理，我不得不取消此次行程。我没有完成旅途，那么我的这组数据是否能帮助训练模型呢？其实是可以的，用到的算法不再是在线梯度下降，而是时间差分 (Temporal Difference)，缩写为 TD。



图 4.3：紫色的数字 14 和 11 是模型的估计值；蓝色的数字 4.5 是实际观测值。

下面解释 TD 算法的原理。回顾一下我们已有的数据：模型估计从北京到上海一共需要 $\hat{q} = 14$ 小时，我实际用了 $r = 4.5$ 小时到达济南，模型估计还需要 $\hat{q}' = 11$ 小时从济南到上海。到达济南时，根据模型最新估计，整个旅程的总时间为：

$$\hat{y} \triangleq r + \hat{q}' = 4.5 + 11 = 15.5.$$

TD 算法将 $\hat{y} = 15.5$ 称为 **TD 目标** (TD Target)，它比最初的估计 $\hat{q} = 14$ 更可靠。最初的估计 $\hat{q} = 14$ 纯粹是估计的，没有任何事实的成分。TD 目标 $\hat{y} = 15.5$ 也是个估计，但其中有事实的成分：其中的 $r = 4.5$ 就是实际的观测。

基于以上讨论，我们认为 TD 目标 $\hat{y} = 15.5$ 比模型最初的估计值

$$\hat{q} = Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) = 14$$

更可靠，所以可以用 \hat{y} 对模型做“修正”。我们希望估计值 \hat{q} 尽量接近真实值 \hat{y} ，所以用两者的平方差作为损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [Q(\text{“北京”}, \text{“上海”}; \mathbf{w}) - \hat{y}]^2.$$

把 \hat{y} 看做常数，尽管它依赖于 \mathbf{w} 。¹ 计算损失函数的梯度：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q} - \hat{y})}_{\text{记作 } \delta} \cdot \nabla_{\mathbf{w}} Q(\text{“北京”}, \text{“上海”}; \mathbf{w}),$$

此处的 $\delta = \hat{q} - \hat{y}$ 称作 **TD 误差** (TD Error)。做一次梯度下降更新模型参数 \mathbf{w} ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta \cdot \nabla_{\mathbf{w}} Q(\text{“北京”}, \text{“上海”}; \mathbf{w}).$$

TD 算法指的是用此公式更新模型参数 \mathbf{w} 。

¹根据定义，TD 目标 $\hat{y} = r + \hat{q}'$ ，而 $\hat{q}' = Q(\text{“济南”}, \text{“上海”}; \mathbf{w})$ 依赖于 \mathbf{w} 。因此， \hat{y} 其实是 \mathbf{w} 的函数。然而 TD 算法忽视这一点，在求梯度的时候，将 \hat{y} 视为常数，而非 \mathbf{w} 的函数。

如果你仍然不理解 TD 算法，那么请换个角度来思考问题。模型估计从北京到上海全程需要 $\hat{q} = 14$ 小时，模型还估计从济南到上海需要 $\hat{q}' = 11$ 小时。这就相当于模型做了这样的估计：从北京到济南需要的时间为

$$\hat{q} - \hat{q}' = 14 - 11 = 3.$$

而我真实花费 $r = 4.5$ 小时。模型的估计与我的真实观测之差为

$$\delta = 3 - 4.5 = -1.5.$$

这就是 TD 误差！以上分析说明 TD 误差 δ 就是模型估计与真实观测之差。TD 算法的目的是通过更新参数 w 使得目标函数 $L(w) = \frac{1}{2}\delta^2$ 减小。

4.3 推导训练 DQN 的 TD 算法

本节推导训练 DQN 的 TD 算法。² 回忆一下回报的定义: $U_t = \sum_{k=t}^n \gamma^{k-t} \cdot R_k$ 。由这个定义可得:

$$U_t = R_t + \gamma \cdot \underbrace{\sum_{k=t+1}^n \gamma^{k-t} \cdot R_k}_{= U_{t+1}}.$$

回忆一下，最优动作价值函数可以写成

$$Q_*(s_t, a_t) = \max_{\pi} \mathbb{E}[U_t \mid S_t = s_t, A_t = a_t].$$

从上面两个公式出发，经过一系列数学推导（见附录 A），可以得到下面的定理。这个定理是最优贝尔曼方程 (Optimal Bellman Equations) 的一种形式。

定理 4.1. 最优贝尔曼方程

$$\underbrace{Q_*(s_t, a_t)}_{U_t \text{ 的期望}} = \mathbb{E}_{S_{t+1} \sim p(\cdot | s_t, a_t)} \left[R_t + \gamma \cdot \underbrace{\max_{A \in \mathcal{A}} Q_*(S_{t+1}, A)}_{U_{t+1} \text{ 的期望}} \mid S_t = s_t, A_t = a_t \right].$$



贝尔曼方程的右边是个期望，我们可以对期望做蒙特卡洛近似。期望是关于随机变量 S_{t+1} 求的，所以需要从环境得到 S_{t+1} 的一个观测值。当智能体执行动作 a_t 之后，环境通过状态转移函数 $p(s_{t+1}|s_t, a_t)$ 计算出新状态 s_{t+1} ，并将其反馈给智能体。奖励 R_t 最多只依赖于 S_t 、 A_t 、 S_{t+1} 。那么当我们观测到 s_t 、 a_t 、 s_{t+1} ，则奖励 R_t 也被观测到，记作 r_t 。有了这个四元组：

$$(s_t, a_t, r_t, s_{t+1}),$$

我们就有了贝尔曼方程右边期望的一个蒙特卡洛近似。可以把贝尔曼方程写作：

$$Q_*(s_t, a_t) \approx r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q_*(s_{t+1}, a). \quad (4.1)$$

这是不是很像驾驶时间预测问题？左边的 $Q_*(s_t, a_t)$ 就像是模型预测“北京到上海”的总时间， r_t 像是实际观测的“北京到济南”的时间， $\gamma \cdot \max_{a \in \mathcal{A}} Q_*(s_{t+1}, a)$ 相当于模型预测剩余路程“济南到上海”的时间。见图 4.4 中的类比。

把公式 4.1 中的最优动作价值函数 $Q_*(s_t, a_t)$ 替换成神经网络 $Q(s_t, a_t; \mathbf{w})$ ，得到：

$$\underbrace{Q(s_t, a_t; \mathbf{w})}_{\text{预测 } \hat{q}_t} \approx \underbrace{r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w})}_{\text{TD 目标 } \hat{y}_t}.$$

左边的 $\hat{q}_t \triangleq Q(s_t, a_t; \mathbf{w})$ 是神经网络在 t 时刻做出的预测，其中没有任何事实成分。右边的 TD 目标 \hat{y}_t 是神经网络在 $t+1$ 时刻做出的预测，它部分基于真实观测到的奖励 r_t 。 \hat{q}_t 和 \hat{y}_t 两者都是对最优动作价值 $Q_*(s_t, a_t)$ 的估计，但是 \hat{y}_t 部分基于事实，因此更可信。

²严格地讲，本节推导的是“Q 学习算法”，它属于 TD 算法的一种。本节就称其为 TD 算法；下一章再具体介绍 Q 学习算法。

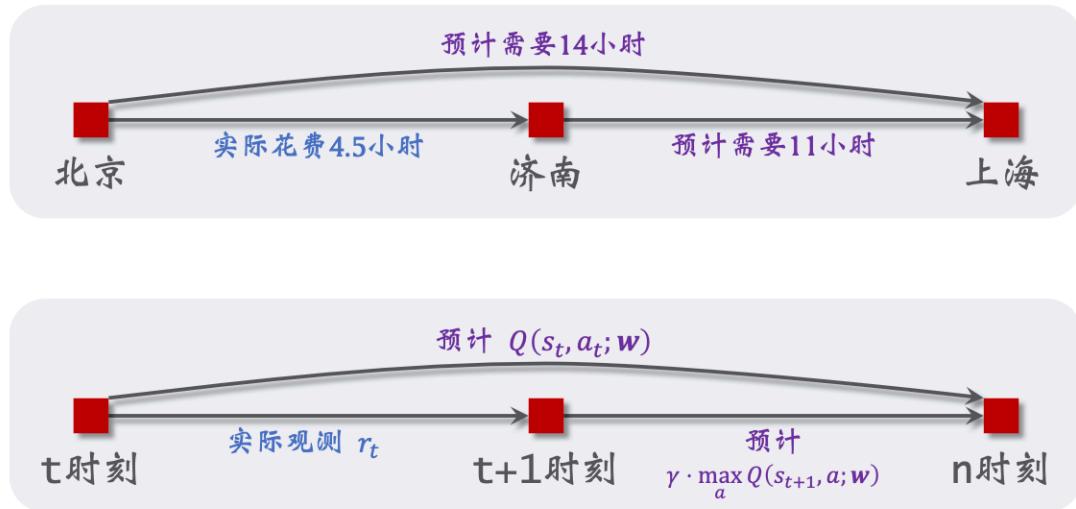


图 4.4: 用“驾车时间”类比 DQN。

应当鼓励 $\hat{q}_t \triangleq Q(s_t, a_t; \mathbf{w})$ 接近 \hat{y}_t 。定义损失函数：

$$L(\mathbf{w}) = \frac{1}{2} [Q(s_t, a_t; \mathbf{w}) - \hat{y}_t]^2.$$

假装 \hat{y} 是常数，计算 L 关于 \mathbf{w} 的梯度：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q}_t - \hat{y}_t)}_{\text{TD 误差 } \delta_t} \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

做一步梯度下降，可以让 \hat{q}_t 更接近 \hat{y}_t ：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

这个公式就是训练 DQN 的 TD 算法。

4.4 训练流程

首先回顾上一节的结论。给定一个四元组 (s_t, a_t, r_t, s_{t+1}) ，我们可以计算出 DQN 的预测值

$$\hat{q}_t = Q(s_t, a_t; \mathbf{w}),$$

以及 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \mathbf{w}) \quad \text{和} \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

TD 算法用这个公式更新 DQN 的参数：

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} Q(s_t, a_t; \mathbf{w}).$$

注意，算法所需数据为 (s_t, a_t, r_t, s_{t+1}) 这个四元组，与控制智能体运动的策略 π 无关。这就意味着可以用任何策略控制智能体，同时记录下算法运动轨迹，作为 DQN 的训练数据。因此，DQN 的训练可以分割成两个独立的部分：收集训练数据、更新参数 \mathbf{w} 。

收集训练数据：我们可以用任何策略函数 π 去控制智能体与环境交互，这个 π 就叫做**行为策略 (Behavior Policy)**。比较常用的是 ϵ -greedy 策略：

$$a_t = \begin{cases} \operatorname{argmax}_a Q(s_t, a; \mathbf{w}), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

把智能体在一局游戏中的轨迹记作：

$$s_1, a_1, r_1, s_2, a_2, r_2, \dots, s_n, a_n, r_n.$$

把一条轨迹划分成 n 个 (s_t, a_t, r_t, s_{t+1}) 这种四元组，存入数组，这个数组叫做**经验回放数组 (Replay Buffer)**。

更新 DQN 参数 \mathbf{w} ：随机从经验回放数组中取出一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。设 DQN 当前的参数为 \mathbf{w}_{now} ，执行下面的步骤对参数做一次更新，得到新的参数 \mathbf{w}_{new} 。

1. 对 DQN 做正向传播，得到 Q 值：

$$\hat{q}_j = Q(s_j, a_j; \mathbf{w}_{\text{now}}) \quad \text{和} \quad \hat{q}_{j+1} = \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}_{\text{now}}).$$

2. 计算 TD 目标和 TD 误差：

$$\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1} \quad \text{和} \quad \delta_j = \hat{q}_j - \hat{y}_j.$$

3. 对 DQN 做反向传播，得到梯度：

$$\mathbf{g}_j = \nabla_{\mathbf{w}} Q(s_j, a_j; \mathbf{w}_{\text{now}}).$$

4. 做梯度下降更新 DQN 的参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_j \cdot \mathbf{g}_j.$$

智能体收集数据、更新 DQN 参数这两者可以同时进行。可以在智能体每执行一个动作之前（或之后），对 \mathbf{w} 做几次更新。

4.5 同策略 (On-policy) 与异策略 (Off-policy)

在强化学习中经常会遇到两个专业术语：同策略 (On-policy) 和异策略 (Off-policy)。为了解释同策略和异策略，我们要从行为策略 (Behavior Policy) 和目标策略 (Target Policy) 讲起。

在强化学习中，我们让智能体与环境交互，记录下观测到的状态、动作、奖励，用这些数据来学习一个策略函数。在这一过程中，控制智能体与环境交互的策略被称作行为策略。行为策略的作用是收集经验 (Experience)，即观测的环境、动作、奖励。

训练的目的是得到一个策略函数，在结束训练之后，用这个策略函数来控制智能体；这个策略函数就叫做目标策略。在本章中，目标策略是一个确定性的策略，即用 DQN 控制智能体：

$$a_t = \operatorname{argmax}_a Q(s_t, a; \mathbf{w}).$$

本章的 TD 算法用任意的行为策略收集 (s_t, a_t, r_t, s_{t+1}) 这样的四元组，然后拿它们训练目标策略，即 DQN。

行为策略和目标策略可以相同，也可以不同。同策略是指用相同的行为策略和目标策略；我们暂时还没有学到同策略。异策略是指用不同的行为策略和目标策略；本章的 DQN 是异策略。对于 DQN，行为策略可以不同于目标策略，用任意的行为策略都可以，比如最常用的 ϵ -greedy：

$$a_t = \begin{cases} \operatorname{argmax}_a Q(s_t, a; \mathbf{w}), & \text{以概率 } (1 - \epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作,} & \text{以概率 } \epsilon. \end{cases}$$

让行为策略带有随机性的好处在于能探索更多没见过的状态。

异策略的好处是可以用行为策略收集经验，把 (s_t, a_t, r_t, s_{t+1}) 这样的四元组记录到一个数组里，在事后反复利用这些经验去更新目标策略。这个数组被称作经验回放数组 (Replay Buffer)，这种训练方式被称作经验回放 (Experience Replay)。注意，经验回放只适用于异策略，不适用于同策略，其原因是收集经验时用的行为策略不同于事后训练的目标策略。

相关文献

DQN 首先由 Mnih 等人在 2013 年提出 [1]，其训练用的算法与本章介绍的基本一致，这种简单的训练算法实践中效果不佳。这篇论文用 Atari 游戏评价 DQN 的表现，虽然 DQN 的表现优于已有方法，但是它还是比人类的表现差一截。同样的作者在 2015 年发表了 DQN 的改进版本 [2]，其主要改进在于使用“目标网络”(Target Network)；这个版本的 DQN 在 Atari 游戏上的表现超越了人类玩家。

DQN 的本质是对最优动作价值函数 Q_* 的函数近似。早在 1995 年和 1997 年发表的论文 [3-4] 就把函数近似用于价值学习中。本章使用的 TD 算法叫做 Q 学习算法，它是由 Watkins 在 1989 年在博士论文 [5] 提出的。Watkins 和 Dayan 发表在 1992 年的论文 [6] 分析了 Q 学习的收敛。1994 年的论文 [7-8] 改进了 Q 学习算法的收敛分析。训练 DQN 用到的经验回放是由 Lin 在 1993 年的博士论文 [9] 中提出的。

参考文献

- [1] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.
- [2] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning [J]. nature, 2015, 518(7540): 529-533.
- [3] BAIRD L. Residual algorithms: reinforcement learning with function approximation[M]//Machine Learning Proceedings 1995. [S.l.]: Elsevier, 1995: 30-37.
- [4] TSITSIKLIS J N, VAN ROY B. An analysis of temporal-difference learning with function approximation[J]. IEEE transactions on automatic control, 1997, 42(5): 674-690.
- [5] WATKINS C J C H. Learning from delayed rewards[J]. 1989.
- [6] WATKINS C J, DAYAN P. Q-learning[J]. Machine learning, 1992, 8(3-4): 279-292.
- [7] JAAKKOLA T, JORDAN M I, SINGH S P. On the convergence of stochastic iterative dynamic programming algorithms[J]. Neural computation, 1994, 6(6): 1185-1201.
- [8] TSITSIKLIS J N. Asynchronous stochastic approximation and Q-learning[J]. Machine learning, 1994, 16(3): 185-202.
- [9] LIN L J. Reinforcement learning for robots using neural networks[R]. [S.l.]: Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.
- [10] SCHULMAN J, LEVINE S, ABBEEL P, et al. Trust region policy optimization[C]//International Conference on Machine Learning (ICML). [S.l.: s.n.], 2015.
- [11] NOCEDAL J, WRIGHT S. Numerical optimization[M]. [S.l.]: Springer Science & Business Media, 2006.
- [12] CONN A R, GOULD N I, TOINT P L. Trust region methods[M]. [S.l.]: SIAM, 2000.
- [13] BERTSEKAS D P. Constrained optimization and lagrange multiplier methods[M]. [S.l.]: Academic press, 2014.
- [14] BOYD S, BOYD S P, VANDENBERGHE L. Convex optimization[M]. [S.l.]: Cambridge university press, 2004.
- [15] SILVER D, LEVER G, HEES N, et al. Deterministic policy gradient algorithms[C]//International Conference on Machine Learning (ICML). [S.l.: s.n.], 2014.
- [16] LILlicrap T P, HUNT J J, PRITZEL A, et al. Continuous control with deep reinforcement learning.[C]// International Conference on Learning Representations (ICLR). [S.l.: s.n.], 2016.
- [17] HAFNER R, RIEDMILLER M. Reinforcement learning in feedback control[J]. Machine learning, 2011, 84 (1-2): 137-169.
- [18] PROKHOROV D V, WUNSCH D C. Adaptive critic designs[J]. IEEE transactions on Neural Networks, 1997, 8(5): 997-1007.
- [19] HAUSKNECHT M, STONE P. Deep recurrent Q-learning for partially observable MDPs[C]//AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents. [S.l.: s.n.], 2015.
- [20] RASHID T, SAMVELYAN M, SCHROEDER C, et al. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning[C]//International Conference on Machine Learning (ICML). [S.l.: s.n.], 2018.