

可以用链式法则求出梯度 \mathbf{g}_j 。复习一下链式法则。如果有这样的函数关系: $\theta \rightarrow a \rightarrow q$, 那么 q 关于 θ 的导数可以写成

$$\frac{\partial q}{\partial \theta} = \frac{\partial a}{\partial \theta} \cdot \frac{\partial q}{\partial a}$$

价值网络的输出与 θ 的函数关系如图 10.6 所示。应用链式法则, 我们得到下面的定理。

定理 10.1. 确定策略梯度

$$\nabla_{\theta} q(s_j, \mu(s_j; \theta); \mathbf{w}) = \nabla_{\theta} \mu(s_j; \theta) \cdot \nabla_{\mathbf{a}} q(s_j, \hat{\mathbf{a}}_j; \mathbf{w}), \quad \text{其中 } \hat{\mathbf{a}}_j = \mu(s_j; \theta).$$

由此我们得到更新 θ 的算法。每次从经验回放数组里随机抽取一个状态, 记作 s_j 。计算 $\hat{\mathbf{a}}_j = \mu(s_j; \theta)$ 。用梯度上升更新一次 θ :

$$\theta \leftarrow \theta + \beta \cdot \nabla_{\theta} \mu(s_j; \theta) \cdot \nabla_{\mathbf{a}} q(s_j, \hat{\mathbf{a}}_j; \mathbf{w}).$$

此处的 β 是学习率, 需要手动调。这样做梯度上升, 可以逐渐让目标函数 $J(\theta)$ 增大, 也就是让评委给演员的平均打分更高。

训练价值网络: 首先通俗解释训练价值网络的原理。训练价值网络的目标是让价值网络 $q(s, \mathbf{a}; \mathbf{w})$ 的预测越来越接近真实价值函数 $Q_{\pi}(s, \mathbf{a})$ 。如果把价值网络看做评委, 那么训练评委的目标就是让他的打分越来越准确。每一轮训练都要用到一个实际观测的奖励 r , 可以把 r 看做“真理”, 用它来校准评委的打分。

训练价值网络要用 TD 算法。这里的 TD 算法与之前学过的标准 Actor-Critic 类似, 都是让价值网络去拟合 TD 目标。每次从经验回放数组中取出一个四元组 $(s_j, \mathbf{a}_j, r_j, s_{j+1})$, 用它更新一次参数 \mathbf{w} 。首先让价值网络做预测:

$$\hat{q}_j = q(s_j, \mathbf{a}_j; \mathbf{w}) \quad \text{和} \quad \hat{q}_{j+1} = q(s_{j+1}, \mu(s_{j+1}; \theta); \mathbf{w}).$$

计算 TD 目标 $\hat{y}_j = r_j + \gamma \cdot \hat{q}_{j+1}$ 。定义损失函数

$$L(\mathbf{w}) = \frac{1}{2} \left[q(s_j, \mathbf{a}_j; \mathbf{w}) - \hat{y}_j \right]^2,$$

计算梯度

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \underbrace{(\hat{q}_j - \hat{y}_j)}_{\text{TD 误差 } \delta_j} \cdot \nabla_{\mathbf{w}} q(s_j, \mathbf{a}_j; \mathbf{w}),$$

做一轮梯度下降更新参数 \mathbf{w} :

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

这样可以让损失函数 $L(\mathbf{w})$ 减小, 也就是让价值网络的预测 $\hat{q}_j = q(s, \mathbf{a}; \mathbf{w})$ 更接近 TD 目标 \hat{y}_j 。公式中的 α 是学习率, 需要手动调。

训练流程: 做训练的时候, 可以同时对价值网络和策略网络做训练。每次从经验回放数组中抽取一个四元组, 记作 $(s_j, \mathbf{a}_j, r_j, s_{j+1})$ 。把神经网络当前参数记作 \mathbf{w}_{now} 和 θ_{now} 。执行以下步骤更新策略网络和价值网络:

1. 让策略网络做预测:

$$\hat{\mathbf{a}}_j = \mu(s_j; \theta_{\text{now}}) \quad \text{和} \quad \hat{\mathbf{a}}_{j+1} = \mu(s_{j+1}; \theta_{\text{now}}).$$

注 计算动作 \hat{a}_j 用的是当前的策略网络 $\mu(s_j; \theta_{\text{now}})$, 用 \hat{a}_j 来更新 θ_{now} ; 而从经验回放数组中抽取的 a_j 则是用过时的策略网络 $\mu(s_j; \theta_{\text{old}})$ 算出的, 用 a_j 来更新 w_{now} 。请注意 \hat{a}_j 与 a_j 的区别。

2. 让价值网络做预测:

$$\hat{q}_j = q(s_j, a_j; w_{\text{now}}) \quad \text{和} \quad \hat{q}_{j+1} = q(s_{j+1}, \hat{a}_{j+1}; w_{\text{now}}).$$

3. 计算 TD 目标和 TD 误差:

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

4. 更新价值网络:

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \delta_j \cdot \nabla_w q(s_j, a_j; w_{\text{now}}).$$

5. 更新策略网络:

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \nabla_\theta \mu(s_j; \theta_{\text{now}}) + \nabla_a q(s_j, \hat{a}_j; w_{\text{now}}).$$

在实践中, 上述算法的表现并不好; 读者应当采用第 10.4 节介绍的技巧训练策略网络和价值网络。

10.3 深入分析 DPG

上一节介绍的 DPG 是一种“四不像”的方法。DPG 乍看起来很像第 7 章中介绍的策略学习方法，因为 DPG 的目的是学习一个策略 μ ，而价值网络 q 只起辅助作用。然而 DPG 又很像第 4 章中介绍的 DQN，两者都是异策略 (Off-policy)，而且两者存在高估问题。鉴于 DPG 的重要性，我们更深入分析 DPG。

10.3.1 从策略学习的角度看待 DPG

问题 10.1

DPG 中有一个确定策略网络 $\mu(s; \theta)$ 和一个价值网络 $q(s, a; w)$ 。请问价值网络 $q(s, a; w)$ 是对动作价值函数 $Q_\pi(s, a)$ 的近似，还是对最优动作价值函数 $Q_*(s, a)$ 的近似？



答案是动作价值函数 $Q_\pi(s, a)$ 。上一节 DPG 的训练流程中，更新价值网络用到 TD 目标：

$$\hat{y}_j = r_j + \gamma \cdot q(s_{j+1}, \mu(s_{j+1}; \theta_{\text{now}}); w_{\text{now}}).$$

很显然，当前的策略 $\mu(s; \theta_{\text{now}})$ 会直接影响价值网络 q 。策略不同，得到的价值网络 q 就不同。

虽然价值网络 $q(s, a; w)$ 通常是对动作价值函数 $Q_\pi(s, a)$ 的近似，但是我们最终的目标是让 $q(s, a; w)$ 趋近于最优动作价值函数 $Q_*(s, a)$ 。回忆一下，如果 π 是最优策略 π^* ，那么 $Q_\pi(s, a)$ 就等于 $Q_*(s, a)$ 。训练 DPG 的目的是让 $\mu(s; \theta)$ 趋近于最优策略 π^* ，那么理想情况下， $q(s, a; w)$ 最终趋近于 $Q_*(s, a)$ 。

问题 10.2

DPG 的训练中有行为策略 $\mu(s; \theta_{\text{old}}) + \epsilon$ 和目标策略 $\mu(s; \theta_{\text{now}})$ 。价值网络 $q(s, a; w)$ 近似动作价值函数 $Q_\pi(s, a)$ 。请问此处的 π 指的是行为策略还是目标策略？



答案是目标策略 $\mu(s; \theta_{\text{now}})$ ，因为目标策略对价值网络的影响很大。在理想情况下，行为策略对价值网络没有影响。我们用 TD 算法训练价值网络，TD 算法的目的在于鼓励价值网络的预测趋近于 TD 目标。理想情况下，

$$q(s_j, a_j; w) = \underbrace{r_j + \gamma \cdot Q(s_{j+1}, \mu(s_{j+1}; \theta_{\text{now}}); w_{\text{now}})}_{\text{TD 目标}}, \quad \forall (s_j, a_j, r_j, s_{j+1}).$$

在收集经验的过程中，行为策略决定了如何基于 s_j 生成 a_j ，然而这不重要。上面的公式只希望等式左边去拟合等式右边，而不在乎 a_j 是如何生成的。

10.3.2 从价值学习的角度看待 DPG

假如我们知道最优动作价值函数 $Q_*(s, a; \mathbf{w})$, 我们可以这样做决策: 给定当前状态 s_t , 选择最大化 Q 值的动作

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q_*(s_t, a).$$

DQN 记作 $Q(s, a; \mathbf{w})$, 它是 $Q_*(s, a; \mathbf{w})$ 的函数近似。训练 DQN 的目的是让 $Q(s, a; \mathbf{w})$ 趋近 $Q_*(s, a; \mathbf{w})$, $\forall s \in \mathcal{S}, a \in \mathcal{A}$ 。在训练好 DQN 之后, 可以这样做决策:

$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a; \mathbf{w}).$$

如果动作空间 \mathcal{A} 是离散集合, 那么上述最大化很容易实现。可是如果 \mathcal{A} 是连续集合, 则很难对 Q 求最大化。

可以把 DPG 看做对最优动作价值函数 $Q_*(s, a)$ 的另一种近似方式, 用于连续控制问题。我们希望学到策略网络 $\mu(s; \theta)$ 和价值网络 $q(s, a; \mathbf{w})$, 使得

$$q\left(s, \mu(s; \theta); \mathbf{w}\right) \approx \max_{a \in \mathcal{A}} Q_*(s, a), \quad \forall s \in \mathcal{S}.$$

我们可以把 μ 和 q 看做是 Q_* 的近似分解, 而这种分解的目的在于方便做决策:

$$\begin{aligned} a_t &= \mu(s_t; \theta) \\ &\approx \operatorname{argmax}_{a \in \mathcal{A}} Q_*(s_t, a). \end{aligned}$$

10.3.3 DPG 的高估问题

在第 6.2 节中, 我们讨过 DQN 的高估问题: 如果用 Q 学习算法训练 DQN, 则 DQN 会高估真实最优价值函数 Q_* 。把 DQN 记作 $Q(s, a; \mathbf{w})$ 。如果用 Q 学习算法训练 DQN, 那么 TD 目标是

$$\hat{y}_j = r_j + \gamma \cdot \max_{a \in \mathcal{A}} Q(s_{j+1}, a; \mathbf{w}).$$

第 6.2 节得出结论: 如果 $Q(s, a; \mathbf{w})$ 是最优动作价值函数 $Q_*(s, a)$ 的无偏估计, 那么 \hat{y}_j 是对 $Q_*(s_j, a_j)$ 的高估。用 \hat{y}_j 作为目标去更新 DQN, 会导致 $Q(s_j, a_j; \mathbf{w})$ 高估 $Q_*(s_j, a_j)$ 。第 6.2 节的另一个结论是自举会导致高估的传播, 造成高估越来越严重。

DPG 也存在高估问题, 用上一节的算法训练出的价值网络 $q(s, a; \mathbf{w})$ 会高估真实动作价值 $Q_\pi(s, a)$ 。造成 DPG 高估的原因与 DQN 类似: 第一, TD 目标是对真实动作价值的高估; 第二, 自举导致高估的传播。下面具体分析两个原因; 如果读者不感兴趣, 只需要记住上述结论即可, 可以跳过下面的内容。

最大化造成高估: 训练策略网络的时候, 我们希望策略网络计算出的动作 $\hat{a} = \mu(s; \theta)$ 能得到价值网络尽量高的评价, 也就是让 $q(s, \hat{a}; \mathbf{w})$ 尽量大。我们通过求解下面的优化模型来学习策略网络:

$$\theta^* = \operatorname{argmax}_{\theta} \mathbb{E}_S \left[q(S, \hat{A}; \mathbf{w}) \right], \quad \text{s.t. } \hat{A} = \mu(S; \theta).$$

这个公式的意思是 $\mu(s; \theta^*)$ 是最优的确定策略网络。上面的公式与下面的公式意义相同

(虽然不严格等价):

$$\boldsymbol{\mu}(s; \boldsymbol{\theta}^*) = \underset{\mathbf{a} \in \mathcal{A}}{\operatorname{argmax}} q(s, \mathbf{a}; \mathbf{w}), \quad \forall s \in \mathcal{S}.$$

这个公式的意思也是 $\boldsymbol{\mu}(s; \boldsymbol{\theta}^*)$ 是最优的确定策略网络。训练价值网络 q 时用的 TD 目标是

$$\begin{aligned}\hat{y}_j &= r_j + \gamma \cdot q(s_{j+1}, \boldsymbol{\mu}(s_{j+1}; \boldsymbol{\theta}); \mathbf{w}) \\ &\approx r_j + \gamma \cdot \max_{\mathbf{a}_{j+1}} q(s_{j+1}, \mathbf{a}_{j+1}; \mathbf{w}).\end{aligned}$$

根据第 6.2 节中分析, 上面公式中的 \max 会导致 \hat{y}_j 高估真实动作价值 $Q_\pi(s_j, a_j; \mathbf{w})$ 。在训练 q 时, 我们把 \hat{y}_j 作为目标, 鼓励价值网络 $q(s_j, a_j; \mathbf{w})$ 接近 \hat{y}_j , 这会导致 $q(s_j, a_j; \mathbf{w})$ 高估真实动作价值。

自举造成偏差传播: 我们在第 6.2 节讨论过自举 (Bootstrapping) 造成偏差的传播。

TD 目标

$$\hat{y}_j = r_j + \gamma \cdot q(s_{j+1}, \boldsymbol{\mu}(s_{j+1}; \boldsymbol{\theta}); \mathbf{w})$$

是用价值网络算出来的, 而它又被用于更新价值网络 q 本身, 这属于自举。假如价值网络 $q(s_{j+1}, \mathbf{a}_{j+1}; \boldsymbol{\theta})$ 高估了真实动作价值 $Q_\pi(s_{j+1}, \mathbf{a}_{j+1})$, 那么 TD 目标 \hat{y}_j 则是对 $Q_\pi(s_j, \mathbf{a}_j)$ 的高估, 这会导致 $q(s_j, a_j; \mathbf{w})$ 高估 $Q_\pi(s_j, \mathbf{a}_j)$ 。自举让高估从 $(s_{j+1}, \mathbf{a}_{j+1})$ 传播到 (s_j, \mathbf{a}_j) 。

10.4 双延时确定策略梯度 (TD3)

由于存在高估等问题，DPG 实际运行的效果并不好。本节介绍的 Twin Delayed Deep Deterministic Policy Gradient (TD3) 可以大幅提升算法的表现，把策略网络和价值网络训练得更好。注意，本节只是改进训练用的算法，并不改变神经网络的结构。

10.4.1 高估问题的解决方案

解决方案——目标网络：为了解决自举和最大化造成的高估，我们需要使用目标网络 (Target Networks) 计算 TD 目标 \hat{y}_j 。训练中需要两个目标网络：

$$q(s, \mathbf{a}; \mathbf{w}^-) \text{ 和 } \mu(s; \boldsymbol{\theta}^-).$$

它们与价值网络、策略网络的结构完全相同，但是参数不同。TD 目标是用目标网络算的：

$$\hat{y}_j = r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}; \mathbf{w}^-), \quad \text{其中 } \hat{\mathbf{a}}_{j+1} = \mu(s_{j+1}; \boldsymbol{\theta}^-).$$

把 \hat{y}_j 作为目标，更新 \mathbf{w} ，鼓励 $q(s_j, a_j; \mathbf{w})$ 接近 \hat{y}_j 。四个神经网络之间的关系如图 10.7 所示。这种方法可以在一定程度上缓解高估，但是实验表明高估仍然很严重。

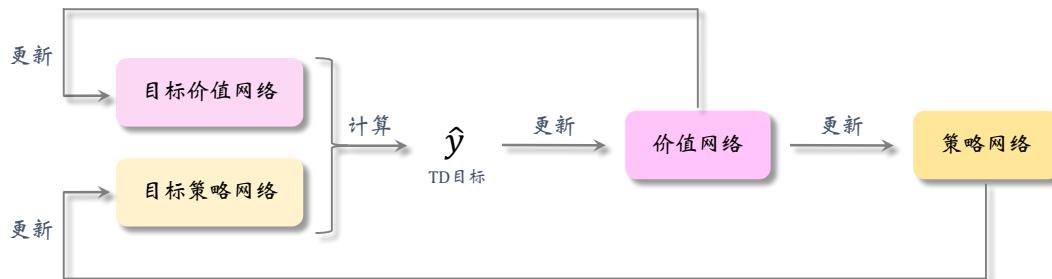


图 10.7：四个神经网络之间的关系。

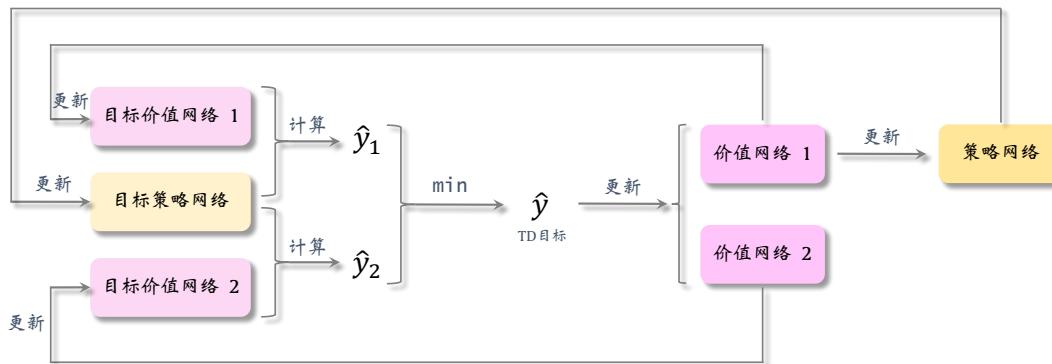


图 10.8：截断双 Q 学习算法中六个神经网络之间的关系。

更好的解决方案——截断双 Q 学习 (Clipped Double Q-Learning)：这种方法使用两个价值网络和一个策略网络：

$$q(s, \mathbf{a}; \mathbf{w}_1), \quad q(s, \mathbf{a}; \mathbf{w}_2), \quad \mu(s; \boldsymbol{\theta}).$$

10.4 双延时确定策略梯度 (TD3)

三个神经网络各对应一个目标网络：

$$q(s, \mathbf{a}; \mathbf{w}_1^-), \quad q(s, \mathbf{a}; \mathbf{w}_2^-), \quad \mu(s; \boldsymbol{\theta}^-).$$

用目标策略网络计算动作：

$$\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}^-),$$

然后用两个目标价值网络计算：

$$\begin{aligned}\hat{y}_{j,1} &= r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}^-; \mathbf{w}_1^-), \\ \hat{y}_{j,2} &= r_j + \gamma \cdot q(s_{j+1}, \hat{\mathbf{a}}_{j+1}^-; \mathbf{w}_2^-).\end{aligned}$$

取两者较小者为 TD 目标：

$$\hat{y}_j = \min \left\{ \hat{y}_{j,1}, \hat{y}_{j,2} \right\}.$$

截断双 Q 学习中的六个神经网络的关系如图 10.8 所示。

10.4.2 其他改进方法

可以在截断双 Q 学习算法的基础上做两处小的改进，进一步提升算法的表现。两种改进分别是往动作中加噪声、减小更新策略网络和目标网络的频率。

往动作中加噪声： 上一小节中截断双 Q 学习用目标策略网络计算动作： $\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}^-)$ 。把这一步改成：

$$\hat{\mathbf{a}}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}^-) + \xi.$$

公式中的 ξ 是个随机向量，表示噪声，它的每一个元素独立随机从截断正态分布 (Clipped Normal Distribution) 中抽取。把截断正态分布记作 $\mathcal{CN}(0, \sigma^2, -c, c)$ ，意思是均值为零，标准差为 σ 的正态分布，但是变量落在区间 $[-c, c]$ 之外的概率为零。正态分布与截断正态分布的对比如图 10.9 所示。使用截断正态分布，而非正态分布，是为了防止噪声 ξ 过大。使用截断，保证噪声大小不会超过 $-c$ 和 c 。

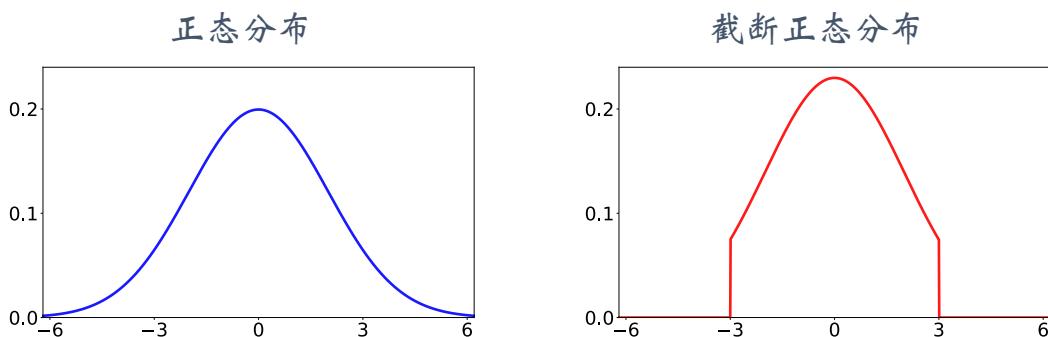


图 10.9：正态分布 $\mathcal{N}(0, 1^2)$ 和截断正态分布 $\mathcal{CN}(0, 1^2, -3, 3)$ 。

减小更新策略网络和目标网络的频率： Actor-Critic 用价值网络来指导策略网络的更新。如果价值网络 q 本身不可靠，那么用价值网络 q 给动作打的分数是不准确的，无助于改进策略网络 μ 。在价值网络 q 还很差的时候就急于更新 μ ，非但不能改进 μ ，反而

会由于 μ 的变化导致 q 的训练不稳定。

实验表明，应当让策略网络 μ 以及三个目标网络的更新慢于价值网络 q 。传统的 Actor-Critic 的每一轮训练都对策略网络、价值网络、以及目标网络做一次更新。更好的方法是每一轮更新一次价值网络，但是每隔 k 轮更新一次策略网络和三个目标网络。 k 是超参数，需要调。

10.4.3 训练流程

本节介绍了三种技巧，改进 DPG 的训练。第一，用截断双 Q 学习，缓解价值网络的高估。第二，往目标策略网络中加噪声，起到平滑作用。第三，降低策略网络和三个目标网络的更新频率。使用这三种技巧的算法被称作双延时确定策略梯度 (Twin Delayed Deep Deterministic Policy Gradient)，缩写是 TD3。

TD3 与 DPG 都属于异策略 (Off-policy)，可以用任意的行为策略收集经验，事后做经验回放训练策略网络和价值网络。收集经验的方式与原始的训练算法相同，用 $a_t = \mu(s_t; \theta) + \epsilon$ 与环境交互，把观测到的四元组 (s_t, a_t, r_t, s_{t+1}) 存入经验回放数组。

初始的时候，策略网络和价值网络的参数都是随机的。这样初始化目标网络的参数：

$$\mathbf{w}_1^- \leftarrow \mathbf{w}_1, \quad \mathbf{w}_2^- \leftarrow \mathbf{w}_2, \quad \boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}.$$

训练策略网络和价值网络的时候，每次从数组中随机抽取一个四元组，记作 (s_j, a_j, r_j, s_{j+1}) 。用下标 now 表示神经网络当前的参数，用下标 new 表示更新后的参数。然后执行下面的步骤，更新价值网络、策略网络、目标网络。

1. 让目标策略网络做预测： $\hat{a}_{j+1}^- = \mu(s_{j+1}; \boldsymbol{\theta}_{\text{now}}^-) + \xi$ 。其中向量 ξ 的每个元素都独立从截断正态分布 $\mathcal{CN}(0, \sigma^2, -c, c)$ 中抽取。

2. 让两个目标价值网络做预测：

$$\widehat{q}_{1,j+1} = q(s_{j+1}, \hat{a}_{j+1}^-; \mathbf{w}_{1,\text{now}}^-) \quad \text{和} \quad \widehat{q}_{2,j+1} = q(s_{j+1}, \hat{a}_{j+1}^-; \mathbf{w}_{2,\text{now}}^-).$$

3. 计算 TD 目标：

$$\widehat{y}_j = r_j + \gamma \cdot \min \left\{ \widehat{q}_{1,j+1}, \widehat{q}_{2,j+1} \right\}.$$

4. 让两个价值网络做预测：

$$\widehat{q}_{1,j} = q(s_j, a_j; \mathbf{w}_{1,\text{now}}) \quad \text{和} \quad \widehat{q}_{2,j} = q(s_j, a_j; \mathbf{w}_{2,\text{now}}).$$

5. 计算 TD 误差：

$$\delta_{1,j} = \widehat{q}_{1,j} - \widehat{y}_j \quad \text{和} \quad \delta_{2,j} = \widehat{q}_{2,j} - \widehat{y}_j.$$

6. 更新价值网络：

$$\begin{aligned} \mathbf{w}_{1,\text{new}} &\leftarrow \mathbf{w}_{1,\text{now}} - \alpha \cdot \delta_{1,j} \cdot \nabla_{\mathbf{w}} q(s_j, a_j; \mathbf{w}_{1,\text{now}}), \\ \mathbf{w}_{2,\text{new}} &\leftarrow \mathbf{w}_{2,\text{now}} - \alpha \cdot \delta_{2,j} \cdot \nabla_{\mathbf{w}} q(s_j, a_j; \mathbf{w}_{2,\text{now}}). \end{aligned}$$

7. 每隔 k 轮更新一次策略网络和三个目标网络：

- 让策略网络做预测： $\hat{a}_j = \mu(s_j; \theta)$ 。然后更新策略网络：

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{now}} + \beta \cdot \nabla_{\boldsymbol{\theta}} \mu(s_j; \boldsymbol{\theta}_{\text{now}}) \cdot \nabla_{\mathbf{a}} q(s_j, \hat{a}_j; \mathbf{w}_{1,\text{now}}).$$

- 更新目标网络的参数:

$$\begin{aligned}\boldsymbol{\theta}_{\text{new}}^- &\leftarrow \tau \boldsymbol{\theta}_{\text{new}} + (1 - \tau) \boldsymbol{\theta}_{\text{now}}^-, \\ \boldsymbol{w}_{1,\text{new}}^- &\leftarrow \tau \boldsymbol{w}_{1,\text{new}} + (1 - \tau) \boldsymbol{w}_{1,\text{now}}^-, \\ \boldsymbol{w}_{2,\text{new}}^- &\leftarrow \tau \boldsymbol{w}_{2,\text{new}} + (1 - \tau) \boldsymbol{w}_{2,\text{now}}^-.\end{aligned}$$

10.5 随机高斯策略

上一节用确定策略网络解决连续控制问题。本节用不同的方法做连续控制，本节的策略网络是随机的，它是随机正态分布（也叫高斯分布）。

10.5.1 基本思路

我们先研究最简单的情形：自由度等于 1，也就是说动作 a 是实数，动作空间 $\mathcal{A} \subset \mathbb{R}$ 。把动作的均值记作 $\mu(s)$ ，标准差记作 $\sigma(s)$ ，它们都是状态 s 的函数。用正态分布的概率密度函数作为策略函数：

$$\pi(a | s) = \frac{1}{\sqrt{6.28 \cdot \sigma(s)}} \cdot \exp\left(-\frac{[a - \mu(s)]^2}{2 \cdot \sigma^2(s)}\right). \quad (10.2)$$

假如我们知道函数 $\mu(s)$ 和 $\sigma(s)$ 的解析表达式，可以这样做控制：

1. 观测到当前状态 s ，预测均值 $\hat{\mu} = \mu(s)$ 和标准差 $\hat{\sigma} = \sigma(s)$ 。
2. 从正态分布中做随机抽样： $a \sim \mathcal{N}(\hat{\mu}, \hat{\sigma}^2)$ ；智能体执行动作 a 。

然而我们并不知道 $\mu(s)$ 和 $\sigma(s)$ 是怎么样的函数。一个很自然的想法是用神经网络来近似这两个函数。把神经网络记作 $\mu(s; \theta)$ 和 $\sigma(s; \theta)$ ，其中 θ 表示神经网络中的可训练参数。但实践中最好不要直接近似标准差 σ ，而是近似方差对数 $\ln \sigma^2$ ²。定义两个神经网络：

$$\mu(s; \theta) \text{ 和 } \rho(s; \theta),$$

分别用于预测均值和方差对数。可以按照图10.10来搭建神经网络。神经网络的输入是状态 s ，通常是向量、矩阵、或者张量。神经网络有两个输出头，分别记作 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 。可以这样用神经网络做控制：

1. 观测到当前状态 s ，计算均值 $\hat{\mu} = \mu(s; \theta)$ ，方差对数 $\hat{\rho} = \rho(s; \theta)$ ，以及方差 $\hat{\sigma}^2 = \exp(\hat{\rho})$ 。
2. 从正态分布中做随机抽样： $a \sim \mathcal{N}(\hat{\mu}, \hat{\sigma}^2)$ ；智能体执行动作 a 。

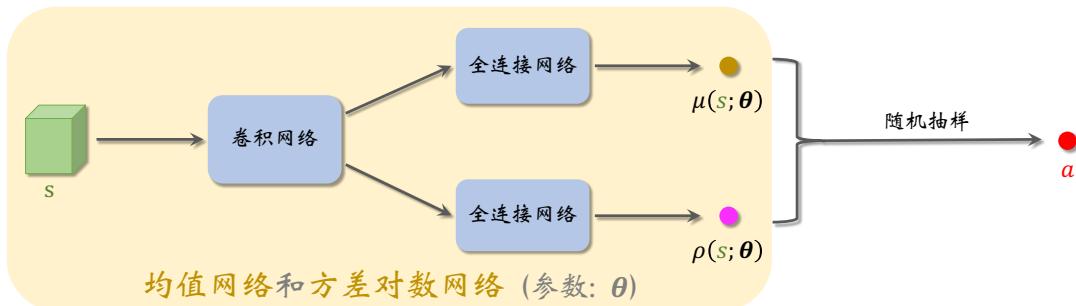


图 10.10：高斯策略网络有两个头，一个输出均值 $\hat{\mu}$ ，另一个输出方差对数 $\hat{\rho}$ 。

用神经网络近似均值和标准差之后，公式 (10.2) 中的策略函数 $\pi(a|s)$ 变成了下面的

²标准差 σ 必须非负，如果把 σ 作为优化变量，那么优化模型有约束条件，给求解造成困难。方差对数 ρ 的取值范围是所有实数，因此不需要约束条件。

策略网络：

$$\pi(a|s; \theta) = \frac{1}{\sqrt{6.28 \cdot \exp[\rho(s; \theta)]}} \cdot \exp\left(-\frac{[a - \mu(s; \theta)]^2}{2 \cdot \exp[\rho(s; \theta)]}\right).$$

实际做控制的时候，我们只需要神经网络 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ ，用不到真正的策略网络 $\pi(a|s; \theta)$ 。

10.5.2 随机高斯策略网络

上一小节假设控制问题的自由度是 $d = 1$ ，也就是说动作 a 是标量。实际问题中的自由度 d 往往大于 1，那么动作 a 是 d 维向量。对于这样的问题，我们修改一下神经网络结构，让两个输出 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 都 d 维向量；见图10.11。

用标量 a_i 表示动作向量 a 的第 i 个元素。用函数 $\mu_i(s; \theta)$ 和 $\rho_i(s; \theta)$ 分别表示 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 的第 i 个元素。我们用下面这个特殊的多元正态分布的概率密度函数作为策略网络：

$$\pi(a|s; \theta) = \prod_{i=1}^d \frac{1}{\sqrt{6.28 \cdot \exp[\rho_i(s; \theta)]}} \cdot \exp\left(-\frac{[a_i - \mu_i(s; \theta)]^2}{2 \cdot \exp[\rho_i(s; \theta)]}\right).$$

做控制的时候只需要均值网络 $\mu(s; \theta)$ 和方差对数网络 $\rho(s; \theta)$ ，不需要策略网络 $\pi(a|s; \theta)$ 。做训练的时候也不需要 $\pi(a|s; \theta)$ ，而是要用辅助网络 $f(s, a; \theta)$ 。总而言之，策略网络 π 只是帮助你理解本节的方法而已，实际算法中不会出现 π 。

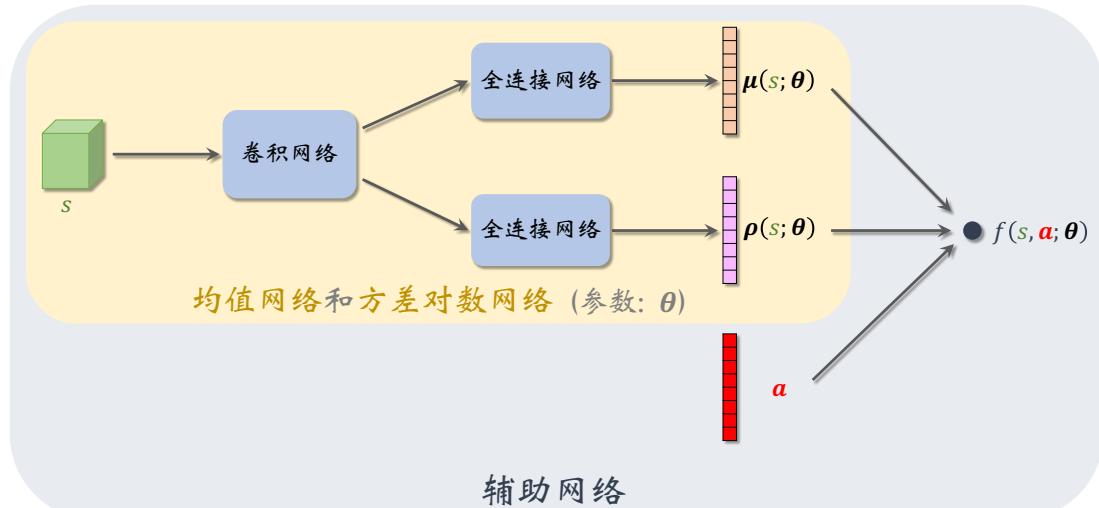


图 10.11：辅助网络的结构示意图。辅助神经网络的输入是状态 s 与动作 a ，输出是实数 $f(s, a; \theta)$ 。

图10.11描述了辅助网络 $f(s, a; \theta)$ 与 μ 、 ρ 、 a 的关系。辅助网络具体是这样定义的：

$$f(s, a; \theta) = -\frac{1}{2} \sum_{i=1}^d \left(\rho_i(s; \theta) + \frac{[a_i - \mu_i(s; \theta)]^2}{\exp[\rho_i(s; \theta)]} \right).$$

它的可训练参数 θ 都是从 $\mu(s; \theta)$ 和 $\rho(s; \theta)$ 中来的。不难发现，辅助网络与策略网络有

这样的关系：

$$f(s, \mathbf{a}; \boldsymbol{\theta}) = \ln \pi(\mathbf{a}|s; \boldsymbol{\theta}) + \text{Constant.} \quad (10.3)$$

10.5.3 策略梯度

回忆一下之前学过的内容。在 t 时刻的折扣回报记作随机变量

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

动作价值函数 $Q_\pi(s_t, \mathbf{a}_t)$ 是对折扣回报 U_t 的条件期望。前面章节推导过策略梯度的蒙特卡洛近似：

$$\mathbf{g} = Q_\pi(s, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta}} \ln \pi(\mathbf{a}|s; \boldsymbol{\theta}).$$

由公式 (10.3) 可得：

$$\mathbf{g} = Q_\pi(s, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta}). \quad (10.4)$$

有了策略梯度，就可以学习参数 $\boldsymbol{\theta}$ 。训练的过程大致如下：

1. 搭建均值网络 $\mu(s; \boldsymbol{\theta})$ 、方差对数网络 $\rho(s; \boldsymbol{\theta})$ 、辅助网络 $f(s, \mathbf{a}; \boldsymbol{\theta})$ 。
2. 让智能体与环境交互，记录每一步的状态、动作、奖励，并对参数 $\boldsymbol{\theta}$ 做更新：
 - (a). 观测到当前状态 s ，计算均值、方差对数、方差：

$$\hat{\mu} = \mu(s; \boldsymbol{\theta}), \quad \hat{\rho} = \rho(s; \boldsymbol{\theta}), \quad \hat{\sigma}^2 = \exp(\hat{\rho}).$$

此处的指数函数 $\exp(\cdot)$ 应用到向量的每一个元素上。

- (b). 设 $\hat{\mu}_i$ 和 $\hat{\sigma}_i$ 分别是 d 维向量 $\hat{\mu}$ 和 $\hat{\sigma}$ 的第 i 个元素。从正态分布中做抽样：

$$a_i \sim \mathcal{N}(\hat{\mu}_i, \hat{\sigma}_i^2), \quad \forall i = 1, \dots, d.$$

把得到的动作记作 $\mathbf{a} = [a_1, \dots, a_d]$ 。

- (c). 近似计算动作价值： $\hat{q} \approx Q_\pi(s, \mathbf{a})$ 。
- (d). 用反向传播计算出辅助网络关于参数 $\boldsymbol{\theta}$ 的梯度： $\nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta})$ 。
- (e). 用策略梯度上升更新参数：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta \cdot \hat{q} \cdot \nabla_{\boldsymbol{\theta}} f(s, \mathbf{a}; \boldsymbol{\theta}).$$

此处的 β 是学习率。

但是算法中有一个没解决的问题：我们并不知道动作价值 $Q_\pi(s, \mathbf{a})$ 。有两种办法近似 $Q_\pi(s, \mathbf{a})$ ：REINFORCE 用实际观测的折扣回报代替 $Q_\pi(s, \mathbf{a})$ ，Actor-Critic 用价值网络近似 Q_π 。后面两小节具体讲解这两种算法。

10.5.4 用 REINFORCE 学习参数

REINFORCE 用实际观测的折扣回报 $u_t = \sum_{k=t}^n \gamma^{k-t} \cdot r_k$ 代替动作价值 $Q_\pi(s_t, \mathbf{a}_t)$ 。道理是这样的。动作价值是回报的期望：

$$Q_\pi(s_t, \mathbf{a}_t) = \mathbb{E}[U_t | S_t = s_t, A_t = \mathbf{a}_t].$$

随机变量 U_t 的一个实际观测值 u_t 是期望的蒙特卡洛近似。这样一来，公式 (10.4) 中的

策略梯度就能近似成

$$\mathbf{g} \approx u_t \cdot \nabla_{\theta} f(s, \mathbf{a}; \theta).$$

在搭建好均值网络 $\mu(s; \theta)$ 、方差对数网络 $\rho(s; \theta)$ 、辅助网络 $f(s, \mathbf{a}; \theta)$ 之后，我们用 REINFORCE 更新参数 θ 。设当前参数为 θ_{now} 。REINFORCE 重复以下步骤，直到收敛：

1. 用 $\mu(s; \theta_{\text{now}})$ 和 $\rho(s; \theta_{\text{now}})$ 控制智能体与环境交互，完成一局游戏，得到一条轨迹：

$$s_1, \mathbf{a}_1, r_1, s_2, \mathbf{a}_2, r_2, \dots, s_n, \mathbf{a}_n, r_n.$$

2. 计算所有的回报：

$$u_t = \sum_{k=t}^T \gamma^{k-t} \cdot r_k, \quad \forall t = 1, \dots, n.$$

3. 对辅助网络做反向传播，得到所有的梯度：

$$\nabla_{\theta} f(s_t, \mathbf{a}_t; \theta_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 用策略梯度上升更新参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot u_t \cdot \nabla_{\theta} f(s_t, \mathbf{a}_t; \theta_{\text{now}})$$

上述算法标准的 REINFORCE，效果不如使用基线的 REINFORCE。读者可以参考第 8.2 节的内容，把状态价值作为基线，改进上面描述的算法。REINFORCE 算法属于同策略 (On-policy)，不能使用经验回放。

10.5.5 用 Actor-Critic 学习参数

Actor-Critic 需要搭建一个价值网络 $q(s, \mathbf{a}; \mathbf{w})$ ，用于近似动作价值函数 $Q_{\pi}(s, \mathbf{a})$ 。价值网络的结构如图 10.12 所示。此外，还需要一个目标价值网络 $q(s, \mathbf{a}; \mathbf{w}^-)$ ，网络结构相同，但是参数不同。

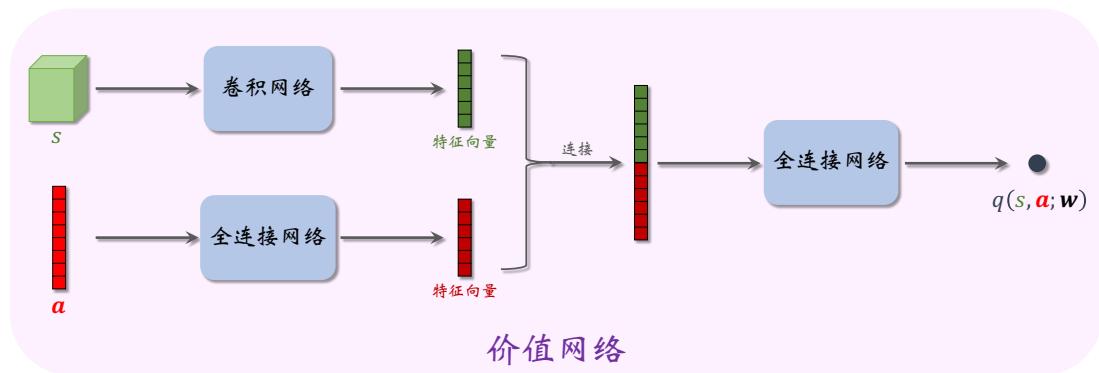


图 10.12：价值网络 $q(s, \mathbf{a}; \mathbf{w})$ 的结构。输入是状态 s 和动作 a ，输出是实数。

在搭建好均值网络 μ 、方差对数网络 ρ 、辅助网络 f 、价值网络 q 之后，我们用 SARSA 算法更新价值网络参数 \mathbf{w} ，用近似策略梯度更新控制器参数 θ 。设当前参数为 \mathbf{w}_{now} 和 θ_{now} 。重复以下步骤更新价值网络参数、控制器参数，直到收敛：

1. 实际观测到当前状态 s_t ，用控制器算出均值 $\mu(s_t; \theta_{\text{now}})$ 和方差对数 $\rho(s_t; \theta_{\text{now}})$ ，然

后随机抽样得到动作 \mathbf{a}_t 。智能体执行动作 \mathbf{a}_t ，观测到奖励 r_t 与新的状态 s_{t+1} 。

2. 计算均值 $\mu(s_{t+1}; \theta_{\text{now}})$ 和方差对数 $\rho(s_{t+1}; \theta_{\text{now}})$ ，然后随机抽样得到动作 $\tilde{\mathbf{a}}_{t+1}$ 。这个动作只是假想动作，智能体不予执行。
3. 用价值网络计算出：

$$\hat{q}_t = q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}).$$

4. 用目标网络计算出：

$$\hat{q}_{t+1} = q(s_{t+1}, \tilde{\mathbf{a}}_{t+1}; \mathbf{w}_{\text{now}}^-).$$

5. 计算 TD 目标和 TD 误差：

$$\hat{y}_t = r_t + \gamma \cdot \hat{q}_{t+1}, \quad \delta_t = \hat{q}_t - \hat{y}_t.$$

6. 更新价值网络的参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}).$$

7. 更新策略网络参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \hat{q}_t \cdot \nabla_{\theta} f(s_t, \mathbf{a}_t; \theta_{\text{now}})$$

8. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

算法中的 α 、 β 、 τ 都是超参数，需要手动调整。上述算法标准的 Actor-Critic，效果不如 Advantage Actor-Critic (A2C)。读者可以参考第 8.3 节的内容，用 A2C 改进上面描述的算法。

∽ 第十章 相关文献 ∽

确定策略梯度 (Deterministic Policy Gradient, DPG) 方法由 David Silver 等人在 2014 年提出 [99]。随后同一批作者把相似的想法与深度学习结合起来，提出深度确定策略梯度 (Deep Deterministic Policy Gradient, 缩写 DDPG)，文章在 2016 年发表 [67]。这两篇论文使得 DPG 方法流行起来。但值得注意的是，相似的想法在更早的论文中有提出：[46, 85]。

2018 年的论文 [42] 提出三种对 DPG 的改进方法，并将改进的算法命名为 TD3。2017 年的论文 [45] 提出了 Soft Actor-Critic (SAC)，也可以解决连续控制问题。

Degriz 等人在 2012 年发表的论文 [35] 使用正态分布的概率密度函数作为策略函数，并且用线性函数近似均值和方差对数。类似的连续控制方法最早由 Williams 在 1987 和 1992 年提出 [126-127]。

第十一章 对状态的不完全观测

11.1 不完全观测问题

之前章节中的 DQN $Q(s, a; \mathbf{w})$, 策略网络 $\pi(a|s; \boldsymbol{\theta})$ 、 $\mu(s; \boldsymbol{\theta})$, 价值网络 $q(s, a; \mathbf{w})$ 、 $v(s; \mathbf{w})$ 都需要把当前状态 s 作为输入。之前我们一直假设可以完全观测到状态 s ; 在围棋、象棋、五子棋等简单的游戏中, 棋盘上当前的格局就是完整的状态, 符合完全观测的假设。但是在很多实际应用中, 完全观测假设往往不符合实际。比如在星际争霸、英雄联盟等电子游戏中, 屏幕上当前的画面并不能完整反映出游戏的状态, 因为观测只是地图的一小部分; 甚至最近的 100 帧也无法反映出游戏真实的状态。

把 t 时刻的状态记作 s_t , 把观测记作 o_t 。观测 o_t 可以是当前游戏屏幕上的画面, 也可以是最近 100 帧画面。我们无法用 $\pi(a_t|s_t; \boldsymbol{\theta})$ 做决策, 因为我们不知道 s_t 。最简单的解决办法就是用当前观测 o_t 代替状态 s_t , 用 $\pi(a_t|o_t; \boldsymbol{\theta})$ 做决策。同理, 对于 DQN 和价值网络, 也用 o_t 代替 s_t 。虽然这种简单的方法可行, 但是效果恐怕不好。



图 11.1: 在迷宫问题中, 智能体可能知道迷宫的整体格局, 也可能只知道自己附近的格局。

图 11.1 的例子是让智能体走迷宫。图 11.1(a) 中智能体可以完整观测到迷宫 s ; 这种问题最容易解决。图 11.1(b) 中智能体只能观测到自身附近一小块区域 o_t , 这属于不完全观测问题, 这种问题较难解决。如果仅仅靠当前观测 o_t 做决策, 智能体做出的决策是非常盲目的, 很难走出迷宫。一种更合理的办法是让智能体记住过去的观测, 这样就能对状态的观测越来越完整, 做出越来越理性的决策; 如图 11.1(c) 所示。

对于不完全观测的强化学习问题, 应当记忆过去的观测, 用所有已知的信息做决策。这正是人类解决不完全观测问题的方式。对于星际争霸、扑克牌、麻将等不完全观测的游戏, 人类玩家也需要记忆; 人类玩家的决策不止依赖于当前时刻的观测 o_t , 而是依赖于过去所有的观测 o_1, \dots, o_t 。把从初始到 t 时刻为止的所有观测记作:

$$\mathbf{o}_{1:t} = [o_1, o_2, \dots, o_t]$$

可以用 $\mathbf{o}_{1:t}$ 代替状态 s , 作为策略网络的输入, 那么策略网络就记作:

$$\pi(a_t | \mathbf{o}_{1:t}; \boldsymbol{\theta}).$$

该如何实现这样一个策略网络呢？请注意， $\mathbf{o}_{1:t}$ 的大小是变化的。如果 o_1, \dots, o_t 都是 $d \times 1$ 的向量，那么 $\mathbf{o}_{1:t}$ 是 $d \times t$ 的矩阵或 $dt \times 1$ 的向量，它的大小随 t 增长。卷积层和全连接层都要求输入大小固定，因此不能简单地用卷积层和全连接层实现策略网络。一种可行的办法是将卷积层、全连接层与循环层结合，这样就能处理不固定长度的输入。

11.2 循环神经网络 (RNN)

循环神经网络 (Recurrent Neural Network), 缩写 RNN, 是一类神经网络的总称, 由循环层 (Recurrent Layers) 和其他种类的层组成。循环层的作用是把一个序列 (比如时间序列、文本、语音) 映射到一个特征向量。设向量 $\mathbf{x}_1, \dots, \mathbf{x}_n$ 是一个序列。对于所有的 $t = 1, \dots, n$, 循环层把 $(\mathbf{x}_1, \dots, \mathbf{x}_t)$ 映射到特征向量 \mathbf{h}_t 。依次把 $\mathbf{x}_1, \dots, \mathbf{x}_n$ 输入循环层, 会得到:

$$\begin{aligned} (\mathbf{x}_1) &\Rightarrow \mathbf{h}_1, \\ (\mathbf{x}_1, \mathbf{x}_2) &\Rightarrow \mathbf{h}_2, \\ (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) &\Rightarrow \mathbf{h}_3, \\ &\vdots \\ (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_{n-1}) &\Rightarrow \mathbf{h}_{n-1}, \\ (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \dots, \mathbf{x}_{n-1}, \mathbf{x}_n) &\Rightarrow \mathbf{h}_n. \end{aligned}$$

RNN 的好处在于不论输入序列的长度 t 是多少, 从序列中提取出的特征向量 \mathbf{h}_t 的大小是固定的。请特别注意, \mathbf{h}_t 并非只依赖于 \mathbf{x}_t 这一个向量, 而是依赖于 $[\mathbf{x}_1, \dots, \mathbf{x}_t]$; 理想情况下, \mathbf{h}_t 记住了 $[\mathbf{x}_1, \dots, \mathbf{x}_t]$ 中的主要信息。比如 \mathbf{h}_3 是对 $[\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3]$ 的概要, 而非是对 \mathbf{x}_3 这一个向量的概要。

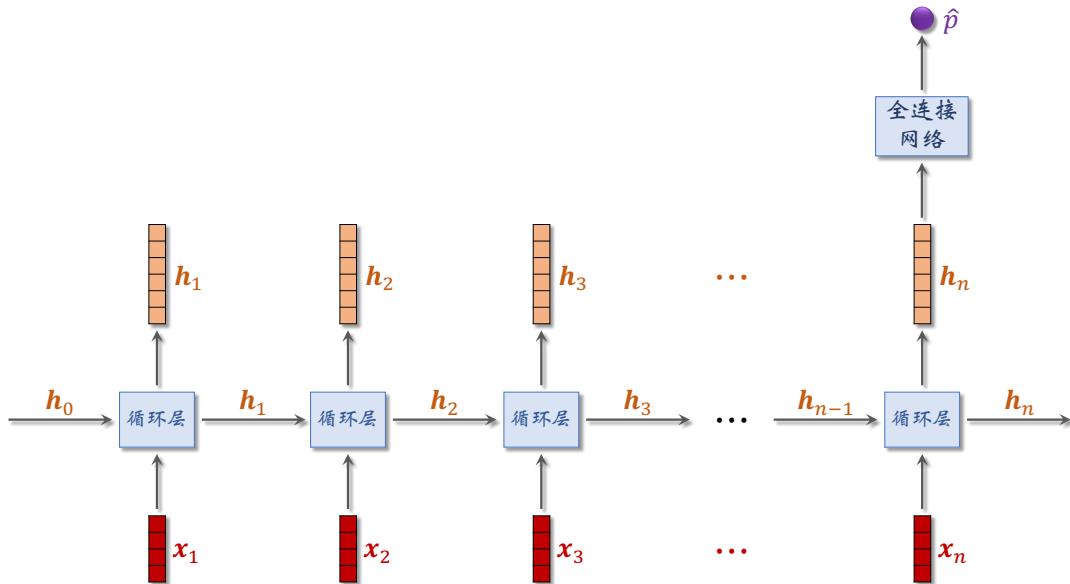


图 11.2: 输入是序列 $\mathbf{x}_1, \dots, \mathbf{x}_t$ 。向量 \mathbf{h}_t 是从所有 t 个输入中提取的特征, 可以把它看做输入序列的一个概要。把 \mathbf{h}_t 输入全连接层 (带 Sigmoid 激活函数), 得到分类结果 \hat{p} 。

举个例子, 用户给商品写的评论由 n 个字组成 (不同的评论有不同的 n) , 我们想要判断评论是正面的还是负面的, 这是个二元分类问题。用词嵌入 (Word Embedding) 把每个字映射到一个向量, 得到 $\mathbf{x}_1, \dots, \mathbf{x}_n$, 把它们依次输入循环层。循环层依次输出 $\mathbf{h}_1, \dots, \mathbf{h}_n$ 。我们只需要用 \mathbf{h}_n , 因为它是从全部输入 $\mathbf{x}_1, \dots, \mathbf{x}_n$ 中提取的特征; 可以忽略掉 $\mathbf{h}_1, \dots, \mathbf{h}_{n-1}$ 。最后, 二元分类器把 \mathbf{h}_n 作为输入, 输出一个介于 0 到 1 之间的数 \hat{p} ,

0 代表负面，1 代表正面。图 11.2 描述了神经网络的结构。

循环层的种类有很多，常见的包括简单循环层、LSTM、GRU。本书只介绍简单循环层。LSTM、GRU 是对简单循环层的改进，结构更复杂，效果更好；但是它们的原理与简单循环层基本相同。读者只需要理解简单循环层就足够了。用 TensorFlow、PyTorch、Keras 编程实现的话，几种循环层的使用方法完全相同（唯一区别是函数名）。

简单循环层的输入记作 $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^{d_{\text{in}}}$ ，输出记作 $\mathbf{h}_1, \dots, \mathbf{h}_n \in \mathbb{R}^{d_{\text{out}}}$ 。循环层的参数是矩阵 $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ 和向量 $\mathbf{b} \in \mathbb{R}^{d_{\text{in}}}$ 。循环层的输出是这样计算出来的：从 $t = 1, \dots, n$ ，依次计算

$$\mathbf{h}_t = \tanh(\mathbf{W}[\mathbf{h}_{t-1}; \mathbf{x}_t] + \mathbf{b}).$$

图 11.3 解释上面的公式。注意，不论输入序列长度 n 是多少，简单循环层的参数只有唯一的 \mathbf{W} 和 \mathbf{b} 。公式中的 \tanh 是双曲正切函数，见图 11.4。 \tanh 是标量函数；如果输入是向量，那么 \tanh 应用到向量的每一个元素上。对于 $d \times 1$ 的向量 \mathbf{z} ，有

$$\tanh(\mathbf{z}) = [\tanh(z_1), \tanh(z_2), \dots, \tanh(z_d)]^T.$$

$$\mathbf{h}_t = \tanh \left[\begin{array}{c} \mathbf{W} \\ \hline \end{array} \right] \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}$$

图 11.3: 简单循环层。

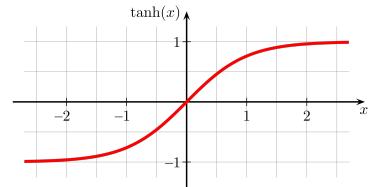


图 11.4: 双曲正切函数。

11.3 RNN 作为策略网络

在不完全观测的设定下，我们希望策略网络能利用所有已经收集的观测 $\mathbf{o}_{1:t} = [o_1, \dots, o_t]$ 做决策。定义策略网络为

$$\mathbf{f}_t = \pi(a_t | \mathbf{o}_{1:t}; \boldsymbol{\theta})$$

结构如图 11.5 所示。在第 t 时刻，观测到 o_t ，用卷积网络提取特征，得到向量 \mathbf{x}_t 。循环层把 \mathbf{x}_t 作为输入，然后输出 \mathbf{h}_t 。 \mathbf{h}_t 是从 $\mathbf{x}_1, \dots, \mathbf{x}_t$ 中提取出的特征，是对所有观测 $\mathbf{o}_{1:t} = [o_1, \dots, o_t]$ 的一个概要。全连接网络（输出层激活函数是 Softmax）把 \mathbf{h}_t 作为输入，然后输出向量 \mathbf{f}_t ，作为 t 时刻决策的依据。 \mathbf{f}_t 的维度是动作空间的大小 $|\mathcal{A}|$ ，它的每个元素对应一个动作，表示选择该动作的概率。

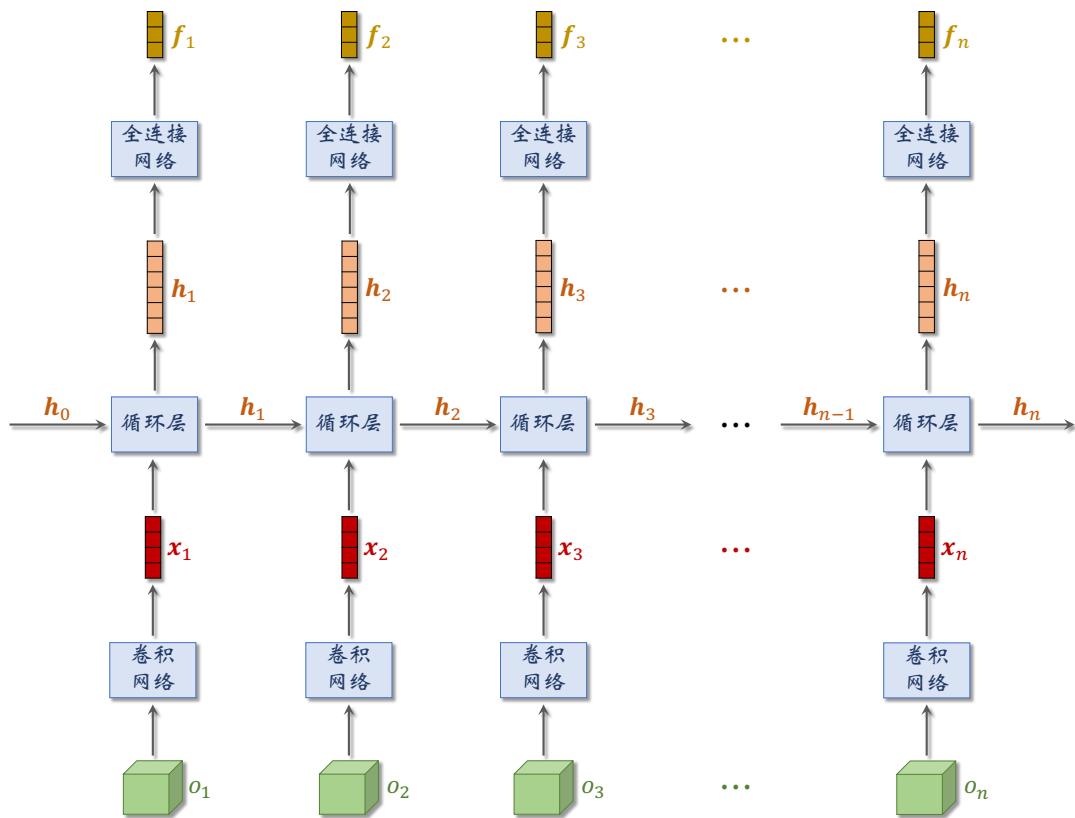


图 11.5：基于 RNN 的策略网络。图中所有的全连接网络都有相同的参数；所有的循环层都有相同的参数；所有的卷积层都有相同的参数。

对于不完全观测问题，我们可以类似地搭建 DQN 和价值网络。DQN 可以定义为：

$$Q(\mathbf{o}_{1:t}, a_t; \mathbf{w}).$$

价值网络可以定义为：

$$q(\mathbf{o}_{1:t}, a_t; \mathbf{w}) \quad \text{或} \quad v(\mathbf{o}_{1:t}; \mathbf{w}).$$

这些神经网络与图 11.5 中策略网络的区别仅在于全连接网络的结构而已；它们使用的卷积网络、循环层与图 11.5 相同。

∽ 第十一章 相关文献 ∽

RNN 是一类很重要的神经网络。学术界认为最早的 RNN 是 Hopfield network [53]，尽管它跟我们今天用的 RNN 很不一样。现在最常用的 RNN 包括 LSTM [52] 和 GRU [29]。注意力机制 (Attention) 由 2015 年的论文 [6] 提出，将注意力机制与 RNN 结合，可以大幅提升 RNN 在机器翻译任务上的表现。注意力机制显然可以用于本章介绍的 RNN 策略网络，但是这样会大幅增加计算量。

2015 年的论文 [47] 首先将 RNN 应用于深度强化学习，把 RNN 与 DQN 相结合，把得到的方法叫做 DRQN。在此之后，RNN 成为解决不完全观测问题的一种标准技巧，比如论文 [74, 39, 86]。

第十二章 模仿学习

模仿学习 (Imitation Learning) 不是强化学习，而是强化学习的一种替代品。模仿学习与强化学习有相同的目的：两者的目的都是学习策略网络，从而控制智能体。模仿学习与强化学习有不同的原理：模仿学习向人类专家学习，目标是让策略网络做出的决策与人类专家相同；而强化学习利用环境反馈的奖励改进策略，目标是让累计奖励（即回报）最大化。

本章介绍三种常见的模仿学习方法：行为克隆 (Behavior Cloning)、逆向强化学习 (Inverse Reinforcement Learning)、生成判别模仿学习 (GAIL)。行为克隆不需要让智能体与环境交互，因此学习的“成本”很低；而逆向强化学习、生成判别模仿学习则需要让智能体与环境交互。

12.1 行为克隆

行为克隆 (Behavior Cloning) 是最简单的模仿学习。行为克隆的目的是模仿人的动作，学出一个随机策略网络 $\pi(a|s; \theta)$ 或者确定策略网络 $\mu(s; \theta)$ 。虽然行为克隆的目的与强化学习中的策略学习类似，但是行为克隆的本质是监督学习（分类或者回归），而不是强化学习。行为克隆通过模仿人类专家的动作来学习策略，而强化学习则是从奖励中学习策略。

模仿学习需要一个事先准备好的数据集，由（状态，动作）这样的二元组构成，记作：

$$\mathcal{X} = \{(s_1, a_1), \dots, (s_n, a_n)\}.$$

其中 s_j 是一个状态，而对应的 a_j 是人类专家基于状态 s_j 做出的动作。可以把 s_j 和 a_j 分别视作监督学习中的输入和标签。

12.1.1 连续控制问题

连续控制的意思是动作空间 \mathcal{A} 是连续集合，比如 $\mathcal{A} = [0, 360] \times [0, 180]$ 。我们搭建类似图 12.1 的确定策略网络，记作 $\mu(s; \theta)$ 。输入是状态 s ，输出是动作向量 a ，它的维度 d 是控制问题的自由度。

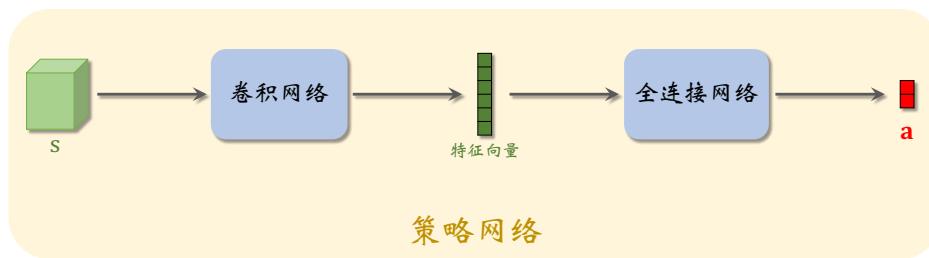


图 12.1：确定策略网络 $\mu(s; \theta)$ 的结构。输入是状态 s ，输出是动作 a 。

行为克隆用回归的方法训练确定策略网络。训练数据集 \mathcal{X} 中的二元组 (s, \mathbf{a}) 的意思是基于状态 s , 人做出动作 \mathbf{a} 。行为克隆鼓励策略网络的决策 $\mu(s; \theta)$ 接近人做出的动作 \mathbf{a} 。定义损失函数

$$L(s, \mathbf{a}; \theta) \triangleq \frac{1}{2} [\mu(s; \theta) - \mathbf{a}]^2.$$

损失函数越小, 说明策略网络的决策越接近人的动作。用梯度更新 θ :

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} L(s, \mathbf{a}; \theta),$$

这样可以让 $\mu(s; \theta)$ 更接近 \mathbf{a} 。

训练流程: 给定数据集 $\mathcal{X} = \{(s_j, \mathbf{a}_j)\}_{j=1}^n$ 。重复下面的随机梯度下降, 直到算法收敛:

1. 从序号 $\{1, \dots, n\}$ 中做均匀随机抽样, 把抽到的序号记作 j 。
2. 设当前策略网络参数为 θ_{now} 。把 s_j 、 \mathbf{a}_j 作为输入, 做反向传播计算梯度, 然后用梯度更新 θ :

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \nabla_{\theta} L(s_j, \mathbf{a}_j; \theta_{\text{now}}).$$

12.1.2 离散控制问题

离散控制的意思是动作空间 \mathcal{A} 是离散集合, 例如 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$ 。我们搭建类似图 12.2 的策略网络, 记作 $\pi(a|s; \theta)$ 。输入是状态 s , 输出记作向量 \mathbf{f} 。 \mathbf{f} 的维度是 $|\mathcal{A}|$, 它的每个元素对应一个动作, 表示选择该动作的概率值。比如给定状态 s , 策略网络输出:

$$\begin{aligned} f_1 &= \pi(\text{左} | s; \theta) = 0.2, \\ f_2 &= \pi(\text{右} | s; \theta) = 0.1, \\ f_3 &= \pi(\text{上} | s; \theta) = 0.7. \end{aligned}$$

也就是说向量 $\mathbf{f} = [0.2, 0.1, 0.7]^T$ 。

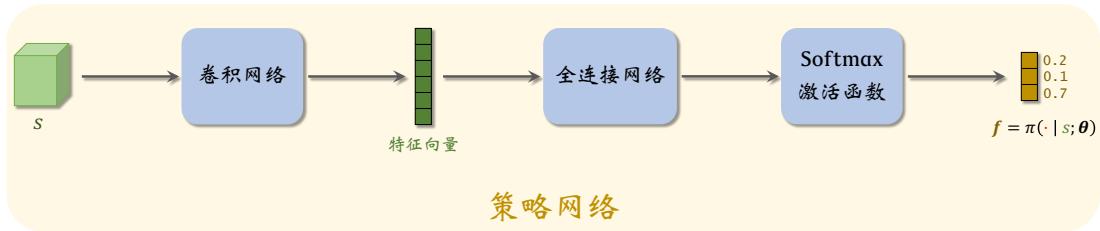


图 12.2: 策略网络 $\pi(a|s; \theta)$ 的神经网络结构。

行为克隆把策略网络 $\pi(a|s; \theta)$ 看做一个多类别分类器, 用监督学习的方法训练这个分类器。把训练数据集 \mathcal{X} 中的动作 a 看做类别标签, 用于训练分类器。需要对类别标签 a 做 One-Hot 编码, 得到 $|\mathcal{A}|$ 维的向量, 记作粗体字母 \bar{a} 。例如 $\mathcal{A} = \{\text{左}, \text{右}, \text{上}\}$, 那么

12.1 行为克隆

对动作的 One-Hot 编码就是：

$$\begin{aligned} a = \text{左} &\implies \bar{a} = [1; 0; 0], \\ a = \text{右} &\implies \bar{a} = [0; 1; 0], \\ a = \text{上} &\implies \bar{a} = [0; 0; 1]. \end{aligned}$$

向量 \bar{a} 与 f 都可以看做是离散的概率分布，可以用交叉熵 (Cross Entropy) 衡量两个分布的区别。交叉熵的定义是：

$$H(\bar{a}, f) \triangleq -\sum_{i=1}^{|A|} \bar{a}_i \cdot \ln f_i.$$

向量 \bar{a} 与 f 越接近，它们的交叉熵越小。用交叉熵作为损失函数：

$$H[\bar{a}, \pi(\cdot | s; \theta)],$$

用梯度更新参数 θ ：

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} H[\bar{a}, \pi(\cdot | s; \theta)].$$

这样可以使交叉熵减小，也就是说策略网络做出的决策 f 更接近人的动作 \bar{a} 。

训练流程： 给定数据集 $\mathcal{X} = \{(s_j, a_j)\}_{j=1}^n$ ，对所有的 a_j 做 One-Hot 编码，变成向量 \bar{a}_j 。重复下面的随机梯度下降，直到算法收敛：

1. 从序号 $\{1, \dots, n\}$ 中做均匀随机抽样，把抽到的序号记作 j 。
2. 设当前策略网络的参数是 θ_{now} 。把 s_j 、 \bar{a}_j 作为输入，做反向传播计算梯度，然后用梯度更新 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \nabla_{\theta} H[\bar{a}_j, \pi(\cdot | s_j; \theta_{\text{now}})].$$

12.1.3 行为克隆与强化学习的对比

行为克隆不是强化学习。强化学习让智能体与环境交互，用环境反馈的奖励指导策略网络的改进，目的是最大化回报的期望。而行为克隆不需要与环境交互，而是利用事先准备好的数据集，用人类的动作指导策略网络的改进，目的是让策略网络的决策更像人类的决策。行为克隆的本质是监督学习（分类或者回归），而不是强化学习，因为行为克隆不需要与环境交互。

行为克隆训练出的策略网络通常效果不佳。人类不会探索奇怪的状态和动作，因此数据集上的状态和动作缺乏多样性。在数据集上做完行为克隆之后，智能体面对真实的环境，可能会见到陌生的状态，智能体的决策可能会很糟糕。行为克隆存在“错误累加”的缺陷。假如当前智能体的决策 a_t 不够好。那么下一时刻的状态 s_{t+1} 可能会比较罕见，于是智能体的决策 a_{t+1} 会很差；这又导致状态 s_{t+2} 非常奇怪，使得决策 a_{t+2} 更糟糕。行为克隆训练出的策略常会进入这种恶性循环。

强化学习效果通常优于行为克隆。如果用强化学习，那么智能体探索过各种各样的状态，尝试过各种各样的动作，知道面对各种状态时应该做什么决策。智能体通过探索，各种状态都见过，比行为克隆有更多的“人生经验”，因此表现会更好。强化学习在围棋、

电子游戏上的表现可以远超顶级人类玩家，而行为克隆却很难超越人类高手。

强化学习的一个缺点在于需要与环境交互，需要探索，而且会改变环境。举个例子，假如把强化学习应用到手术机器人，从随机初始化开始训练策略网络，至少要致死、致残几万个病人才能训练好策略网络。假如把强化学习应用到无人车，从随机初始化开始训练策略网络，至少要撞毁几万辆无人车才能训练好策略网络。假如把强化学习应用到广告投放，那么从初始化到训练好策略网络期间需要做探索，投放的广告会很随机，会严重降低广告收入。如果在真实物理世界应用强化学习，要考虑初始化和探索带来的成本。

行为克隆的优势在于离线训练，可以避免与真实环境的交互，不会对环境产生影响。假如用行为克隆训练手术机器人，只需要把人类医生的观测和动作记录下来，离线训练手术机器人，而不需要真的在病人身上做实验。尽管行为克隆效果不如强化学习，但是行为克隆的成本低。可以先用行为克隆初始化策略网络，而不是随机初始化，然后再做强化学习，这样可以减小对物理世界的有害影响。

12.2 逆向强化学习

逆向强化学习 (Inverse Reinforcement Learning, 缩写 IRL) 非常有名，但是在今天已经不常用了。下一节介绍的 GAIL 更简单，效果更好。本节只简单介绍 IRL 的主要思想，而不深入讲解其数学原理。

IRL 的基本设定：第一，IRL 假设智能体可以与环境交互¹，环境会根据智能体的动作更新状态，但是不会给出奖励。智能体与环境交互的轨迹是这样的：

$$s_1, a_1, \quad s_2, a_2, \quad s_3, a_3, \quad \dots, \quad s_n, a_n.$$

这种设定非常符合物理世界的实际情况。比如人类驾驶汽车，与物理环境交互，根据观测做出决策，得到上面公式中轨迹，轨迹中没有奖励。是不是汽车驾驶问题中没有奖励呢？其实是有奖励的。避免碰撞、遵守交通规则、尽快到达目的地，这些操作背后都有隐含的奖励，只是环境不会直接把奖励告诉我们而已。把奖励看做 (s_t, a_t) 的函数，记作 $R^*(s_t, a_t)$ 。

第二，IRL 假设我们可以把人类专家的策略 $\pi^*(a|s)$ 作为一个黑箱调用。黑箱的意思是我们不知道策略的解析表达式，但是可以使用黑箱策略控制智能体与环境交互，生成轨迹。IRL 假设人类学习策略 π^* 的方式与强化学习相同，都是最大化回报（即累计奖励）的期望，即

$$\pi^* = \max_{\pi} \mathbb{E}_{S_t, A_t, \dots, S_n, A_n} \left[\sum_{k=t}^n \gamma^{k-t} \cdot R^*(S_k, A_k) \right]. \quad (12.1)$$

因为 π^* 与奖励函数 $R^*(s, a)$ 密切相关，所以可以从 π^* 反推出 $R^*(s, a)$ 。

IRL 的基本思想：IRL 的目的是学到一个策略网络 $\pi(a|s; \theta)$ ，模仿人类专家的黑箱策略 $\pi^*(a|s)$ 。如图 12.3 所示，IRL 首先从 $\pi^*(a|s)$ 中学习其隐含的奖励函数 R^* ，然后利用奖励函数做强化学习，得到策略网络的参数 θ 。我们用神经网络 $R(s, a; \rho)$ 来近似奖励函数 R^* 。神经网络 R 的输入是 s 和 a ，输出是实数；我们需要学习它的参数 ρ 。

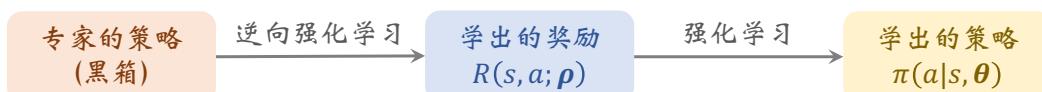


图 12.3

从黑箱策略反推奖励：假设人类专家的黑箱策略 $\pi^*(a|s)$ 满足公式 (12.1)，即 π^* 是应对奖励函数 R^* 的最优策略。对于不同的奖励函数 R^* ，则会有不同的 $\pi^*(a|s)$ 。是否能由 π^* 的决策反推出 R^* 呢？举个例子，图 12.4 是走格子的游戏，动作空间是 $\mathcal{A} = \{\text{上}, \text{下}, \text{左}, \text{右}\}$ 。两个表格表示两局游戏的状态，蓝色的箭头表示 π^* 做出的决策。请读者仔细观察，尝试推断游戏的奖励函数 R^* 。

既然蓝色箭头是最优策略做出的决策，那么沿着蓝色箭头走，可以最大化回报。我

¹注意，上一节的行为克隆无需智能体与环境交互。

们不难做出以下推断：

- 到达绿色格子有正奖励 r_+ ，原因是智能体尽量通过绿色格子。到达绿色格子的奖励只能被收集一次，否则智能体会反复回到绿色格子。
- 到达红色格子有负奖励 $-r_-$ ，因为智能体尽量避开红色格子。由于左图中智能体穿越两个红色格子去收集绿色奖励，说明 $r_+ \gtrsim 2r_-$ 。由于右图中智能体没有穿越四个红格子去收集绿色奖励，而是穿越一个红格子，说明 $r_+ \lesssim 3r_-$ 。
- 到达终点有正奖励 r_* ，因为智能体会尽力走到终点。由于右图中的智能体穿过红色格子，说明 $r_* > r_-$ 。
- 智能体尽量走最短路，说明每走一步，有一个负奖励 $-r_\rightarrow$ 。但是 r_\rightarrow 比较小，否则智能体不会绕路去收集绿色奖励。

注意，从智能体的轨迹中，只能大致推断出奖励函数，但是不可能推断出奖励 r_+ 、 $-r_-$ 、 r_* 、 r_\rightarrow 具体的大小。把四个奖励的数值同时乘以 10，根据新的奖励训练策略，最终学出的最优策略跟原来相同；这说明最优策略对应的奖励函数是不唯一的。

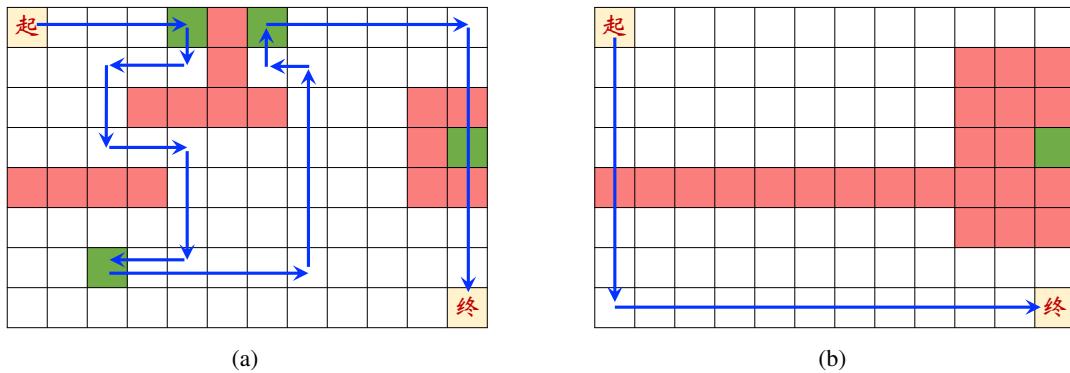


图 12.4：左右两张图表示走格子游戏的两个状态，图中蓝色箭头表示智能体的轨迹。

用奖励函数训练策略网络：假设我们已经学到了奖励函数 $R(s, a; \rho)$ ，那么就可以用它来训练一个策略网络。用策略网络 $\pi(a|s; \theta_{\text{now}})$ 控制智能体与环境交互，每次得到这样一条轨迹：

$$s_1, a_1, s_2, a_2, s_3, a_3, \dots, s_n, a_n,$$

轨迹中没有奖励。比如用策略网络控制无人车驾驶，得到的就是这样一条没有奖励的轨迹。好在我们已经从人类专家身上学到了奖励函数 $R(s, a; \rho)$ ，可以用 R 算出奖励：

$$\hat{r}_t = R(s_t, a_t; \rho), \quad \forall t = 1, \dots, n.$$

可以用任意策略学习方法更新策略网络参数 θ ，比如用 REINFORCE：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n \gamma^{t-1} \cdot \hat{u}_t \cdot \nabla_{\theta} \ln \pi(a|s; \theta_{\text{now}}).$$

公式中的 $\hat{u}_t \triangleq \sum_{k=t}^n \gamma^{k-t} \cdot \hat{r}_k$ 是近似回报。

更新奖励函数：具体该如何学习奖励函数 $R(s, a; \rho)$ 呢？因为我们用 $R(s, a; \rho)$ 来训练策略网络 $\pi(a|s; \theta)$ ，所以 $\pi(a|s; \theta)$ 依赖于 ρ 。IRL 的目标是让 $\pi(a|s; \theta)$ 尽量接近人类

12.2 逆向强化学习

专家的策略 $\pi^*(a|s)$ 。因此要寻找参数 ρ 使得学到的 $\pi(a|s; \theta)$ 最接近 $\pi^*(a|s)$ 。学习 ρ 的方法有很多种，本书不具体介绍了，有兴趣的读者可以阅读相关的文献。

12.3 生成判别模仿学习 (GAIL)

生成判别模仿学习 (Generative Adversarial Imitation Learning, 缩写 GAIL) 需要让智能体与环境交互，但是无法从环境获得奖励²。GAIL 还需要收集人类专家的决策记录（即很多条轨迹）。GAIL 的目标是学习一个策略网络，使得判别器无法区分一条轨迹是策略网络的决策还是人类专家的决策。

12.3.1 生成判别网络 (GAN)

GAIL 的设计基于生成判别网络 (Generative Adversarial Network, 缩写 GAN)。本小节简单介绍 GAN 的基础知识。生成器 (Generator) 和判别器 (Discriminator) 各是一个神经网络。生成器负责生成假的样本，而判别器负责判定一个样本是真是假。举个例子，在人脸数据集上训练生成器和判别器，那么生成器的目标是生成假的人脸图片，可以骗过判别器；而判别器的目标是判断一张图片是真实的还是生成的。理想情况下，当训练结束的时候，判别器的分类准确率是 50%，意味着生成器的输出已经以假乱真。

生成器记作 $a = G(s; \theta)$ ，其中 θ 是参数。它的输入是向量 s ，向量的每一个元素从均匀分布 $U(-1, 1)$ 或标准正态分布 $\mathcal{N}(0, 1)$ 中抽取。生成器的输出是数据（比如图片） x 。生成器通常是一个深度神经网络，其中可能包含卷积层 (Convolution)、反卷积层 (Transposed Convolution)、上采样层 (Upsampling)、全连接层 (Dense) 等。生成器的具体实现取决于具体的问题。

判别器记作 $\hat{p} = D(x; \phi)$ ，其中 ϕ 是参数。它的输入是图片 x ；输出 \hat{p} 是介于 0 到 1 之间的概率值，0 表示“假的”，1 表示“真的”。判别器的功能是二分类器，实现方法很简单。判别器主要由卷积层、池化层 (Pooling)、全连接层等组成。

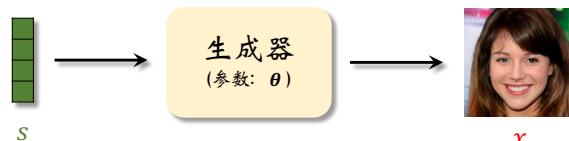


图 12.5：生成器 $a = G(s; \theta)$ 。



图 12.6：判别器 $\hat{p} = D(x; \phi)$ 。

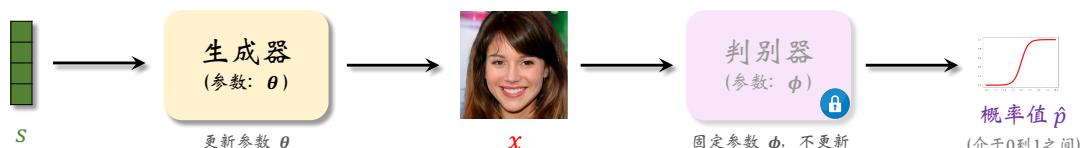


图 12.7：训练生成器 $G(s; \theta)$ 。

训练生成器：将生成器与判别器相连，如图 12.7 所示。固定住判别器的参数，只更

²GAIL 和 IRL 都需要让智能体与环境交互，而行为克隆不需要。

12.3 生成判别模仿学习 (GAIL)

新生成器的参数 θ , 使得生成的图片 $x = G(s; \theta)$ 在判别器的眼里更像真的。对于任意一个随机生成的向量 s , 应该改变 θ , 使得判别器的输出 $\hat{p} = D(x; \phi)$ 尽量接近 1。可以用交叉熵作为损失函数:

$$E(s; \theta) = \ln \left[1 - \underbrace{D(x; \phi)}_{\text{越大越好}} \right]; \quad \text{s.t. } x = G(s; \theta).$$

判别器的输出 $\hat{p} = D(x; \phi)$ 是介于 0 到 1 之间的数。 \hat{p} 越接近 1, 则损失函数 $E(s; \theta) = \ln(1 - \hat{p})$ 越小。训练生成器参数 θ 的时候, 我们希望 \hat{p} 尽量接近 1, 所以应当更新 θ 使得 $E(s; \theta)$ 减小。做一次梯度下降更新 θ :

$$\theta \leftarrow \theta - \beta \cdot \nabla_{\theta} E(s; \theta).$$

此处的 β 是学习率, 需要用户手动调。

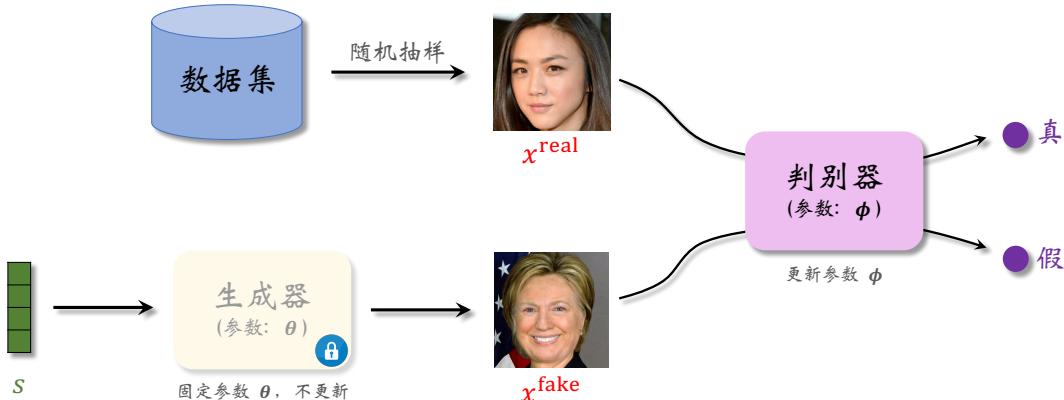


图 12.8: 训练判别器 $D(x; \phi)$ 。

训练判别器: 判别器的本质是个二分类器, 它的输出值 $\hat{p} = D(x; \phi)$ 表示对真伪的预测; \hat{p} 接近 1 表示“真”, \hat{p} 接近 0 表示“假”。判别器的训练如图 12.8 所示。从真实数据集中抽取一个样本, 记作 x^{real} 。再随机生成一个向量 s , 用生成器生成 $x^{\text{fake}} = G(s; \theta)$ 。训练判别器的目标是改进参数 ϕ , 让 $D(x^{\text{real}}; \phi)$ 更接近 1 (真), 让 $D(x^{\text{fake}}; \phi)$ 更接近 0 (假)。也就是说让判别器的分类结果更准确, 更好区分真实图片和生成的假图片。可以用交叉熵作为损失函数:

$$F(x^{\text{real}}, x^{\text{fake}}; \phi) = \ln \left[1 - \underbrace{D(x^{\text{real}}; \phi)}_{\text{越大越好}} \right] + \ln \underbrace{D(x^{\text{fake}}; \phi)}_{\text{越小越好}}.$$

判别器的判断越准确, 则损失函数 $F(x^{\text{real}}, x^{\text{fake}}; \phi)$ 越小。为什么呢?

- 判别器越相信 x^{real} 为真, 则 $D(x^{\text{real}}; \phi)$ 越大, 那么公式中 $\ln[1 - D(x^{\text{real}}; \phi)]$ 越小。
- 判别器越相信 x^{fake} 为假, 则 $D(x^{\text{fake}}; \phi)$ 越小, 那么公式中 $\ln D(x^{\text{fake}}; \phi)$ 越小。

为了减小损失函数 F , 可以做一次梯度下降更新判别器参数 ϕ :

$$\phi \leftarrow \phi - \eta \cdot \nabla_{\phi} F(x^{\text{real}}, x^{\text{fake}}; \phi).$$

此处的 η 是学习率, 需要用户手动调。

批量随机梯度 (Mini-Batch SGD): 上述训练生成器和判别器的方式其实是随机梯度下降 (SGD)，每次只用一个样本。实践中，不妨每次用一个批量 (Batch) 的样本，比如用 $b = 16$ 个，那么会计算出 b 个梯度。用 b 个梯度的平均去更新生成器和判别器。

训练流程: 实践中，要同时训练生成器和判别器，让两者同时进步。³ 每一轮要更新一次生成器，更新一次判别器。设当前生成器、判别器的参数分别为 θ_{now} 和 ϕ_{now} 。

1. (从均匀分布或正态分布中) 随机抽样 b 个向量: s_1, \dots, s_b 。
2. 用生成器生成假样本: $x_j^{\text{fake}} = G(s_j; \theta_{\text{now}})$, $\forall j = 1, \dots, b$ 。
3. 从训练数据集中随机抽样 b 个真样本: $x_1^{\text{real}}, \dots, x_b^{\text{real}}$ 。
4. 更新生成器 $G(s; \theta)$ 的参数:

(a). 计算平均梯度:

$$\mathbf{g}_\theta = \frac{1}{b} \sum_{j=1}^b \nabla_\theta E(s_j; \theta_{\text{now}}).$$

(b). 做梯度下降更新生成器参数: $\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \mathbf{g}_\theta$ 。

5. 更新判别器 $D(x; \phi)$ 的参数:

(a). 计算平均梯度:

$$\mathbf{g}_\phi = \frac{1}{b} \sum_{j=1}^b \nabla_\phi F(x_j^{\text{real}}, x_j^{\text{fake}}; \phi_{\text{now}}).$$

(b). 做梯度下降更新判别器参数: $\phi_{\text{new}} \leftarrow \phi_{\text{now}} - \eta \cdot \mathbf{g}_\phi$ 。

12.3.2 GAIL 的生成器和判别器

训练数据: GAIL 的训练数据是被模仿的对象（比如人类专家）操作智能体得到的轨迹，记作

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_m, a_m].$$

数据集中有 k 条轨迹，把数据集记作:

$$\mathcal{X} = \{\tau^{(1)}, \tau^{(2)}, \dots, \tau^{(k)}\}.$$

生成器: 上一小节中 GAN 的生成器记作 $x = G(s; \theta)$ ，它的输入 s 是个随机抽取的向量，输出 x 是一个数据点（比如一张图片）。本小节中 GAIL 的生成器是策略网络 $\pi(a|s; \theta)$ ，如图 12.9 所示。策略网络的输入是状态 s ，输出是一个向量：

$$\mathbf{f} = \pi(\cdot | s; \theta).$$

输出向量 \mathbf{f} 的维度是动作空间的大小 \mathcal{A} ，它的每个元素对应一个动作，表示执行该动作的概率。给定初始状态 s_1 ，并让智能体与环境交互，可以得到一条轨迹：

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_n, a_n].$$

其中动作是根据策略网络抽样得到的: $a_t \sim \pi(\cdot | s_t; \theta)$, $\forall t = 1, \dots, n$; 下一时刻的状态

³不能让判别器比生成器进步快太多，否则训练会失败。假如判别器的准确率是 100%，那么无论生成器的输出 x 是什么，总被判别为“假”，那么生成器就不知道什么样的 x 更像真的，因而无从改进。

12.3 生成判别模仿学习 (GAIL)

是环境根据状态转移函数计算出来的: $s_{t+1} \sim p(\cdot | s_t, a_t), \forall t = 1, \dots, n$ 。



图 12.9: 策略网络 $\pi(a|s;\theta)$ 的神经网络结构。输入是状态 s , 输出是动作空间 \mathcal{A} 中每个动作的概率值。

判别器: GAIL 的判别器记作 $D(s, a; \phi)$, 它的结构如图 12.10 所示。判别器的输入是状态 s , 输出是一个向量:

$$\hat{p} = D(s, \cdot | \phi).$$

输出向量 \hat{p} 的维度是动作空间的大小 \mathcal{A} , 它的每个元素对应一个动作 a , 把一个元素记作:

$$\hat{p}_a = D(s, a; \phi) \in (0, 1), \quad \forall a \in \mathcal{A}.$$

\hat{p}_a 接近 1 表示 (s, a) 为“真”, 即动作 a 是人类专家做的。 \hat{p}_a 接近 0 表示 (s, a) 为“假”, 即策略网络生成的。



图 12.10: 判别器 $D(s, a; \phi)$ 的神经网络结构。输入是状态 s 。输出向量的维度等于 $|\mathcal{A}|$, 每个元素对应一个动作, 每个元素值都介于 0 到 1 之间。

12.3.3 GAIL 的训练

训练的目的是让生成器 (即策略网络) 生成的轨迹与数据集中的轨迹 (即被模仿对象的轨迹) 一样好。在训练结束的时候, 判别器无法区分生成的轨迹与数据集里的轨迹。

训练生成器: 设 θ_{now} 是当前策略网络的参数。用策略网络 $\pi(a|s;\theta_{\text{now}})$ 控制智能体与环境交互, 得到一条轨迹:

$$\tau = [s_1, a_1, s_2, a_2, \dots, s_n, a_n].$$

判别器可以评价 (s_t, a_t) 有多真实; $D(s_t, a_t; \phi)$ 越大, 说明 (s_t, a_t) 在判别器的眼里越真实。把

$$u_t = \ln D(s_t, a_t; \phi)$$

作为第 t 步的回报； u_t 越大，则说明 (s_t, a_t) 越真实。我们有这样一条轨迹：

$$s_1, a_1, u_1, \quad s_2, a_2, u_2, \quad \dots, \quad s_n, a_n, u_n.$$

于是可以用 TRPO 来更新策略网络。设当前策略网络的参数为 θ_{now} 。定义目标函数：

$$\tilde{L}(\theta | \theta_{\text{now}}) \triangleq \frac{1}{n} \sum_{t=1}^n \frac{\pi(a_t | s_t; \theta)}{\pi(a_t | s_t; \theta_{\text{now}})} \cdot u_t.$$

求解下面的带约束的最大化问题，得到新的参数：

$$\theta_{\text{new}} = \underset{\theta}{\operatorname{argmax}} \tilde{L}(\theta | \theta_{\text{now}}); \quad \text{s.t. } \operatorname{dist}(\theta_{\text{now}}, \theta) \leq \Delta. \quad (12.2)$$

此处的 dist 衡量 θ_{now} 与 θ 的区别， Δ 是一个需要调的超参数。TRPO 的详细解释见第 9.1 节。

训练判别器：训练判别器的目的是让它能区分真的轨迹与生成的轨迹。从训练数据集中均匀抽样一条轨迹，记作

$$\tau^{\text{real}} = [s_1^{\text{real}}, a_1^{\text{real}}, \dots, s_m^{\text{real}}, a_m^{\text{real}}].$$

用策略网络控制智能体与环境交互，得到一条轨迹，记作

$$\tau^{\text{fake}} = [s_1^{\text{fake}}, a_1^{\text{fake}}, \dots, s_n^{\text{fake}}, a_n^{\text{fake}}].$$

公式中的 m 、 n 分别是两条轨迹的长度。

训练判别器的时候，要鼓励判别器做出准确的判断。我们希望判别器知道 $(s_t^{\text{real}}, a_t^{\text{real}})$ 是真的，所以应该鼓励 $D(s_t^{\text{real}}, a_t^{\text{real}}; \phi)$ 尽量大。我们希望判别器知道 $(s_t^{\text{fake}}, a_t^{\text{fake}})$ 是假的，所以应该鼓励 $D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi)$ 尽量小。定义损失函数

$$F(\tau^{\text{real}}, \tau^{\text{fake}}; \phi) = \underbrace{\frac{1}{m} \sum_{t=1}^m \ln [1 - D(s_t^{\text{real}}, a_t^{\text{real}}; \phi)]}_{D \text{ 的输出越大, 这一项越小}} + \underbrace{\frac{1}{n} \sum_{t=1}^n \ln D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi)}_{D \text{ 的输出越小, 这一项越小}}$$

我们希望损失函数尽量小，也就是说判别器能区分开真假轨迹。可以做梯度下降来更新参数 ϕ ：

$$\phi \leftarrow \phi - \eta \cdot \nabla_{\phi} F(\tau^{\text{real}}, \tau^{\text{fake}}; \phi). \quad (12.3)$$

这样可以让损失函数减小，让判别器更能区分开真假轨迹。

训练流程：每一轮训练更新一个生成器，更新一次判别器。训练重复以下步骤，直到收敛。设当前生成器和判别器的参数分别为 θ_{now} 和 ϕ_{now} 。

1. 从训练数据集中均匀抽样一条轨迹，记作

$$\tau^{\text{real}} = [s_1^{\text{real}}, a_1^{\text{real}}, \dots, s_m^{\text{real}}, a_m^{\text{real}}].$$

2. 用策略网络 $\pi(a | s; \theta_{\text{now}})$ 控制智能体与环境交互，得到一条轨迹，记作

$$\tau^{\text{fake}} = [s_1^{\text{fake}}, a_1^{\text{fake}}, \dots, s_n^{\text{fake}}, a_n^{\text{fake}}].$$

3. 用判别器评价策略网络的决策是否真实：

$$u_t = \ln D(s_t^{\text{fake}}, a_t^{\text{fake}}; \phi_{\text{now}}), \quad \forall t = 1, \dots, n.$$

4. 把 τ^{fake} 和 u_1, \dots, u_n 作为输入，用公式 (12.2) 更新策略网络参数，得到 θ_{new} 。

5. 把 τ^{real} 和 τ^{fake} 作为输入，用公式 (12.3) 更新判别器参数，得到 ϕ_{new} 。

∽ 第十二章 相关文献 ∽

行为克隆 (Behavior Cloning) 这个概念很早就出现在人工智能领域，比如 1995 年的论文 [7]、1997 年的论文 [20]。论文 [87, 107] 研究了行为克隆的理论误差，指出行为克隆会让错误累加。行为克隆也叫做 Learning from Demonstration (LfD) [5]。LfD 这个名字最早由 1997 年的论文提出 [90]。

逆向强化学习 (Inverse Reinforcement Learning) 这个问题首先由 Ng 和 Russell 2000 年的论文提出 [81]。这个问题原本是指“从最优策略中推断出奖励函数”。Abbeel 和 Ng 2004 年的论文 [1] 提出从人类专家的策略中反向学习出奖励函数，然后用奖励函数训练策略函数；这种方法被称作学徒学习 (Apprenticeship Learning)。本书第 12.2 节的内容主要基于学徒学习的思想。逆向强化学习的方法有很多种，比如 [16, 38, 64, 106, 136]。

生成判别模仿学习 (Generative Adversarial Imitation Learning) 由 Ho 和 Ermon 在 2016 提出 [50]。它主要基于生成判别网络 (Generative Adversarial Network，缩写 GAN)。GAN 由 Goodfellow 等人在 2014 年提出 [43]。

第四部分

多智能体强化学习

第十三章 并行计算

机器学习的实践中普遍使用并行计算，利用大量的计算资源（比如很多块 GPU）缩短训练所需的时间，用几个小时就能完成原本需要很多天才能完成的训练。深度强化学习自然也不例外；可以用很多处理器同时收集经验、计算梯度，让原本需要很长时间的训练在较短的时间内完成。第 13.1 以并行梯度下降为例讲解并行计算基础知识。第 13.2 介绍异步并行梯度下降算法。第 13.3 介绍两种异步强化学习算法。

13.1 并行计算基础

本节以并行梯度下降 (Parallel Gradient Descent) 为例讲解并行计算的基础知识，用 MapReduce 架构实现并行梯度下降，并且分析并行计算中的时间开销。

13.1.1 并行梯度下降

本节用最小二乘回归 (Least Squares Regression) 为例讲解并行梯度下降的基本原理。把训练数据记作 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ 。最小二乘回归定义为：

$$\min_{\mathbf{w}} \left\{ L(\mathbf{w}) \triangleq \frac{1}{2n} \sum_{j=1}^n (\mathbf{x}_j^T \mathbf{w} - y_j)^2 \right\}.$$

这个优化问题的目标是寻找向量 $\mathbf{w}^* \in \mathbb{R}^d$ ，使得对于所有的 j ， $\mathbf{x}_j^T \mathbf{w}^*$ 都很接近 y_j 。我们可以用梯度下降算法求解这个优化问题。梯度下降重复这个步骤，直到收敛：

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}).$$

公式中的 η 是学习率。如果 η 的取值比较合理，那么梯度下降可以保证 \mathbf{w} 收敛到最优解 \mathbf{w}^* 。目标函数 $L(\mathbf{w})$ 的梯度可以写作：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{n} \sum_{j=1}^n \mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w}), \quad \text{其中 } \mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w}) \triangleq (\mathbf{x}_j^T \mathbf{w} - y_j) \mathbf{x}_j \in \mathbb{R}^d.$$

由于 \mathbf{x}_j 和 \mathbf{w} 都是 d 维向量，因此计算一个 $\mathbf{g}(\mathbf{x}_j, y_j; \mathbf{w})$ 的时间复杂度是 $\mathcal{O}(d)$ 。计算梯度 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 需要计算 \mathbf{g} 函数 n 次，所以计算 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 的时间复杂度是 $\mathcal{O}(nd)$ 。如果用 m 块处理器做并行计算，那么理想情况下每块处理器的计算量是 $\mathcal{O}(\frac{nd}{m})$ 。

下面举一个简单的例子讲解并行梯度下降。假设我们有两块处理器。把梯度 $\nabla_{\mathbf{w}} L(\mathbf{w})$ 展开，得到：

$$\begin{aligned} & \nabla_{\mathbf{w}} L(\mathbf{w}) \\ &= \frac{1}{n} \left[\underbrace{\mathbf{g}(\mathbf{x}_1, y_1; \mathbf{w}) + \dots + \mathbf{g}(\mathbf{x}_{\frac{n}{2}}, y_{\frac{n}{2}}; \mathbf{w})}_{\text{用一号处理器计算, 把结果记作 } \tilde{\mathbf{g}}^1} + \underbrace{\mathbf{g}(\mathbf{x}_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}; \mathbf{w}) + \dots + \mathbf{g}(\mathbf{x}_n, y_n; \mathbf{w})}_{\text{用二号处理器计算, 把结果记作 } \tilde{\mathbf{g}}^2} \right]. \end{aligned}$$

两块处理器各承担一半的计算量，分别输出 d 维向量 $\tilde{\mathbf{g}}^1$ 和 $\tilde{\mathbf{g}}^2$ 。将两块处理器的结果汇总，得到梯度：

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{1}{n} (\tilde{\mathbf{g}}^1 + \tilde{\mathbf{g}}^2).$$

并行梯度下降中的“计算”非常简单；而并行计算的复杂之处在于通信。在一轮梯度下降开始之前，需要把最新的模型参数 w 发送给两块处理器，否则处理器无法计算梯度。在两块处理器完成计算之后，需要做通信，把结果 \tilde{g}^1 和 \tilde{g}^2 汇总到一块处理器上。下一小节以 MapReduce 架构为例，讲解并行梯度下降的实现。

13.1.2 MapReduce

并行计算需要在计算机集群上完成。一个集群有很多处理器和内存条，它们被划分到多个节点 (Compute Node) 上。一个节点上可以有多个处理器，处理器可以共享内存。节点之间不能共享内存，即一个节点不能访问另一个节点的内存。如果两个节点相连接，它们可以通过计算机网络通信（比如 TCP/IP 协议）。

为了协调节点的计算和通信，需要有相应的软件系统。MapReduce 是由 Google 开发的一种软件系统，用于大规模的数据分析和机器学习。MapReduce 原本是软件系统的名字，但是后来人们把类似的系统架构都称作 MapReduce。除了 Google 自己的 MapReduce，比较有名的系统还有 Hadoop¹ 和 Spark²。MapReduce 属于 Server-Client 架构，有一个节点作为中央服务器，其余节点作为 Worker，受服务器控制。服务器用于协调整个系统，而计算主要由 Worker 节点并行完成。

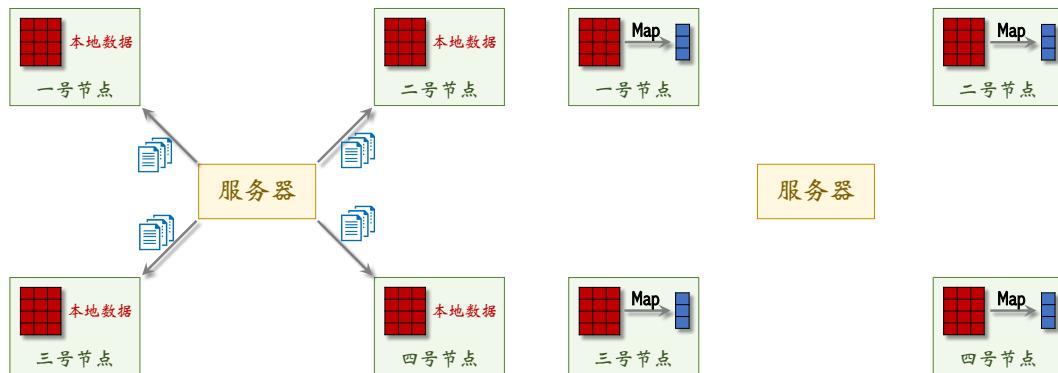


图 13.1: MapReduce 中的广播 (Broadcast) 操作。



图 13.2: MapReduce 中的映射 (Map) 操作。

服务器可以与 Worker 节点做通信传输数据（但是 Worker 节点之间不能相互通信）。一种通信方式是广播 (Broadcast)，即服务器将同一条信息同时发送给所有 Worker 节点；如图 13.1 所示。比如做并行梯度下降的时候，服务器需要把更新过的参数 $w \in \mathbb{R}^d$ 广播到所有 Worker 节点。MapReduce 架构不允许服务器将一条信息只发送给一号节点，而将一条不同的信息只发送给二号节点。服务器只能把相同信息广播到所有节点。

每个节点都可以做计算。映射 (Map) 操作让所有 Worker 节点同时并行做计算；如图 13.2 所示。如果我们要编程实现一个算法，需要自己定义一个函数，它可以让每个 Worker 节点把它的本地数据映射到一些输出值。比如做并行梯度下降的时候，定义函数 g 把三

¹<https://hadoop.apache.org/>

²<https://spark.apache.org/>

元组 $(\mathbf{x}_j, y_j, \mathbf{w})$ 映射到向量

$$z_j = (\mathbf{x}_j^T \mathbf{w} - y_j) \mathbf{x}_j.$$

映射操作要求所有节点都要同时执行同一个函数，比如 $\mathbf{g}(\mathbf{x}_j, y_j, \mathbf{w})$ 。节点不能各自执行不同的函数。

Worker 节点可以向服务器发送信息，最常用的通信操作是 **规约 (Reduce)**。这种操作可以把 Worker 节点上的数据做归并，并且传输到服务器上。如图 13.3 所示，系统对 Worker 节点输出的蓝色向量做规约。如果执行 `sum` 规约函数，那么结果是四个蓝色向量的加和。如果执行 `mean` 规约函数，那么结果是四个蓝色向量的均值。如果执行 `count` 规约函数，那么结果是整数 4，即蓝色向量的数量。

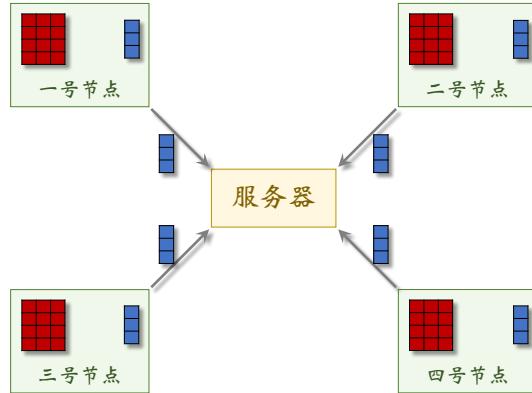


图 13.3: MapReduce 中的规约 (Reduce) 操作。

13.1.3 用 MapReduce 实现并行梯度下降

数据并发 (Data Parallelism): 为了使用 MapReduce 实现并行梯度下降，我们需要把数据集 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 划分到 m 个 Worker 节点上，每个节点上存一部分数据；见图 13.4。这种划分方式叫做数据并发。与数据并发相对的是模型并发 (Model Parallelism)，即将模型参数 \mathbf{w} 划分到 m 个 Worker 节点上；每个节点有全部数据，但是只有一部分模型参数。本书只介绍数据并发，不讨论模型并发。

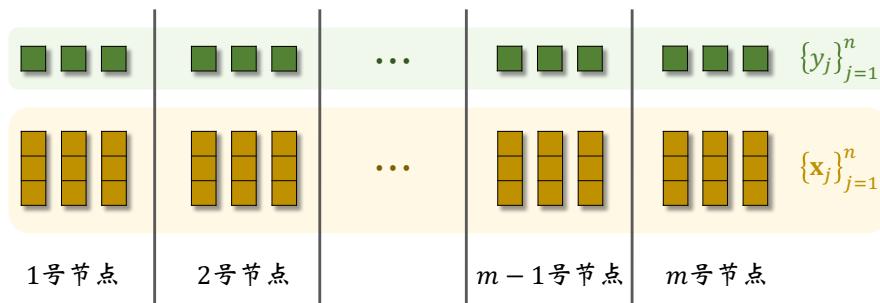


图 13.4: 将数据集 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ 划分到 m 个 Worker 节点上。

并行梯度下降的流程: 用数据并发，设集合 $\mathcal{I}_1, \dots, \mathcal{I}_m$ 是集合 $\{1, 2, \dots, n\}$ 的划分；集合 \mathcal{I}_k 包含第 k 个 Worker 节点上所有样本的序号。并行梯度下降需要重复——广播、映射、规约、更新参数——这四个步骤，直到算法收敛；见示意图 13.5。

1. **广播 (Broadcast):** 服务器将当前的模型参数 \mathbf{w}_{now} 广播到 m 个 Worker 节点。这样一来，所有节点都知道 \mathbf{w}_{now} 。
2. **映射 (Map):** 这一步让 m 个 Worker 节点做并行计算，用本地数据计算梯度。需要

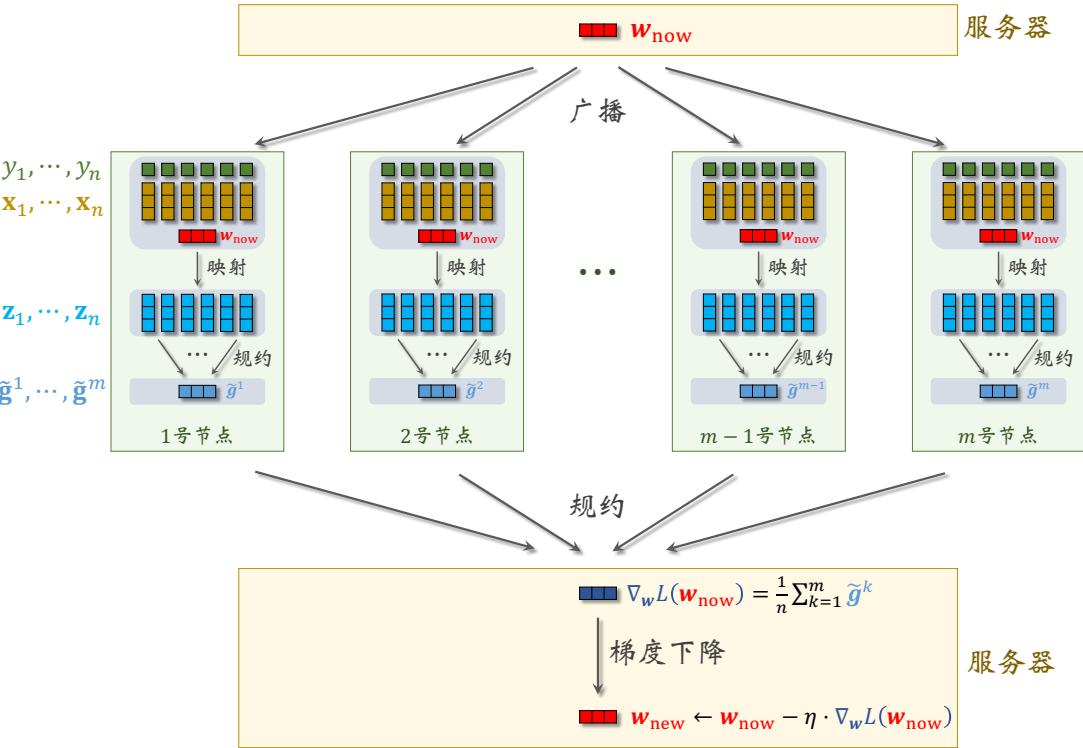


图 13.5: 并行梯度下降的流程。

在编程的时候定义这样一个映射函数：

$$g(\mathbf{x}, y, \mathbf{w}) = (\mathbf{x}^T \mathbf{w} - y) \mathbf{x}.$$

第 k 号 Worker 节点做如下映射：

$$g : (\mathbf{x}_j, y_j, \mathbf{w}_{\text{now}}) \mapsto z_j = (\mathbf{x}_j^T \mathbf{w}_{\text{now}} - y_j) \mathbf{x}_j, \quad \forall j \in \mathcal{I}_k.$$

这样一来，第 k 号 Worker 节点得到向量的集合 $\{z_j\}_{j \in \mathcal{I}_k}$ 。

3. 规约 (Reduce): 在做完映射之后，向量 $z_1, \dots, z_n \in \mathbb{R}^d$ 分布式存储在 m 个 Worker 节点上，每个节点有一个子集。不难看出，目标函数 $L(\mathbf{w}) = \frac{1}{2n} \sum_{j=1}^n (\mathbf{x}_j^T \mathbf{w} - y_j)^2$ 在 \mathbf{w}_{now} 处的梯度等于：

$$\nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}) = \frac{1}{n} \sum_{j=1}^n z_j.$$

因此，我们应该使用 `sum` 规约函数。每个 Worker 节点首先会规约自己本地的 $\{z_j\}_{j \in \mathcal{I}_k}$ ，得到

$$\tilde{\mathbf{g}}^k \triangleq \sum_{j \in \mathcal{I}_k} z_j, \quad \forall k = 1, \dots, m.$$

然后将 $\tilde{\mathbf{g}}_k \in \mathbb{R}^d$ 发送给服务器，服务器对 $\tilde{\mathbf{g}}^1, \dots, \tilde{\mathbf{g}}^m$ 求和，再除以 n ，得到梯度：

$$\nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}) \leftarrow \frac{1}{n} \sum_{k=1}^m \tilde{\mathbf{g}}^k.$$

先在本地做规约，再做通信，只需要传输 md 个浮点数；如果不先在本地归约，直接把所有的 $\{z_j\}_{j=1}^n$ 都发送给服务器，那么需要传输 nd 个浮点数，通信代价大得

多。

- 更新参数：最后，服务器在本地做梯度下降，更新模型参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}).$$

这样就完成了一轮梯度下降，对参数做了一次更新。

13.1.4 并行计算的代价

通常用算法实际运行所需的时间来衡量并行计算的表现。时间有两种定义，请读者注意区分。

- 钟表时间 (Wall-clock Time)**，也叫 Elapsed Real Time，意思是程序实际运行的时间。可以这样理解钟表时间：在程序开始运行的时候，记录下墙上钟表的时刻；在程序结束的时候，再记录钟表的时刻；两者之差就是钟表时间。
- 处理器时间 (CPU Time 或 GPU Time)** 是所有处理器运行时间的总和。比如使用 4 块 CPU 做并行计算，程序运行的钟表时间是 1 分钟，期间 CPU 没有空闲，那么系统的 CPU 时间等于 4 分钟。

处理器数量越多，每块处理器承担的计算量就越小，那么程序运行速度就会越快。所以并行计算可以让钟表时间更短。用多个处理器做并行计算，然而总计算量没有减少，因此并行计算不会让处理器时间更短。

通常用**加速比 (Speedup Ratio)** 衡量并行计算带来的速度提升。加速比是这样计算的：

$$\text{加速比} = \frac{\text{使用一个节点所需的钟表时间}}{\text{使用 } m \text{ 个节点所需的钟表时间}}.$$

通常来说，节点数量越多，算力越强，加速比就越大。在实验报告中，通常需要把加速比绘制成一条曲线。把节点数量设置为不同的值，比如 $m = 1, 2, 4, 8, 16, 32$ ，得到相应的加速比。把 m 作为横轴，把加速比作为纵轴，绘制出加速比曲线；见图 13.6。

在最理想的情况下，使用 m 个节点，每个节点承担 $\frac{1}{m}$ 的计算量，那么钟表时间会减小到原来的 $\frac{1}{m}$ ，即加速比等于 m 。图 13.6 中的蓝色直线是理想情况下的加速比。但实际的加速比往往是图中的红色曲线，即加速比小于 m 。其原因在于计算所需时间只占总的钟表时间的一部分。通信等操作也要花费时间，导致加速比达不到 m 。下面分析并行计算中常见的时间开销。

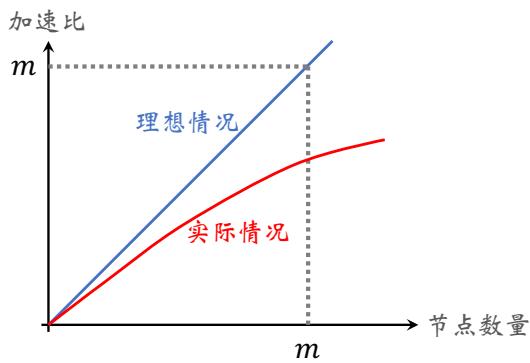


图 13.6: 加速比曲线。

通信量 (Communication Complexity) 的意思是又有多少个比特或者浮点数在服务器与 Worker 节点之间传输。在并行梯度下降的例子中，每一轮梯度下降需要做两次通信：服务器将模型参数 $\mathbf{w} \in \mathbb{R}^d$ 广播给 m 个 Worker 节点，Worker 节点将计算出的梯度 $\tilde{\mathbf{g}}^1, \dots, \tilde{\mathbf{g}}^m$ 发送给服务器。因此每一轮梯度下降的通信量都是 $\mathcal{O}(md)$ 。很显然，通信量越大，通信

花的时间越长。

延迟 (Latency) 是由计算机网络的硬件和软件系统决定的。做通信的时候，需要把大的矩阵、向量拆分成小数据包，通过计算机网络逐个传输数据包。即使数据包再小，从发送到接收之间也需要花费一定时间，这个时间就是延迟。通常来说，延迟与通信次数成正比，而跟通信量关系不大。

通信时间 主要由通信量和延迟造成。我们无法准确预估通信时间（指的是钟表时间），除非实际做实验测量。但我们不妨用下面的公式粗略估计通信时间：

$$\text{通信时间} \approx \frac{\text{通信量}}{\text{带宽}} + \text{延迟}.$$

在并行计算中，通信时间是不容忽视的，通信时间甚至有可能超过计算时间。降低通信量和通信次数是设计并行算法的关键。只有当通信时间远低于计算时间，才能取得较高的加速比。

13.2 同步与异步

本节讨论同步算法、异步算法的区别，重点介绍异步并行梯度下降。用在机器学习中，异步算法的表现通常优于同步算法。

13.2.1 同步算法

上一节介绍的并行梯度下降算法属于**同步算法 (Synchronous Algorithm)**。如图 13.7 所示，在所有 Worker 节点都完成映射 (Map) 的计算之后，系统才能执行规约 (Reduce) 通信。这意味着即使有些节点先完成计算，也必须等待最慢节点；在等待期间，节点处于空闲状态。图 13.7 中黑色的竖线表示同步屏障，即所有节点都完成计算之后才能开始通信，当通信完成之后才能开始下一轮计算。

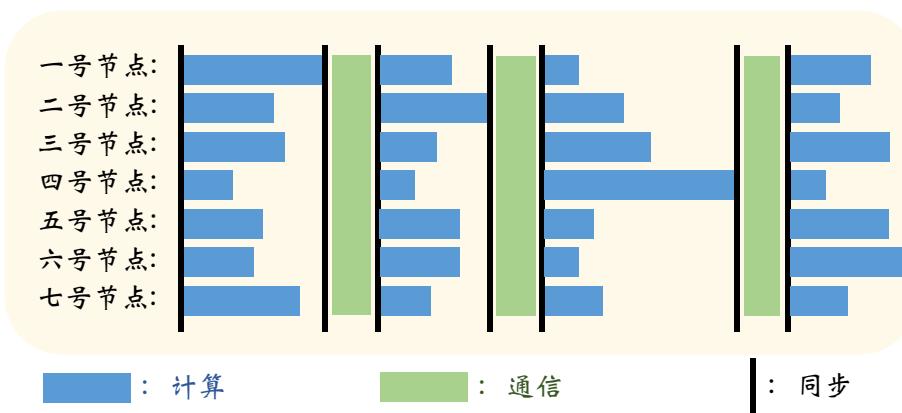


图 13.7: 同步梯度下降中的计算、通信、同步。图中横向表示时间。

同步的代价： 实际软硬件系统中存在负载不平衡、软硬件不稳定、I/O 速度不稳定等因素。因此 Worker 节点会有先后、快慢之分，不会恰好在同一时刻完成任务。同步要求每一轮都必须等待所有节点完成计算，这势必导致“短板效应”，即任务所需时间取决于最慢的节点。同步会造成很多节点处于空闲状态，无法有效利用集群的算力。

Straggler Effect 意思是一个节点的速度远慢于其余节点，导致整个系统长时间处于空闲状态，等待最慢的节点。Straggler 也叫 Outlier，字面意思是“掉队者”。产生 Straggler 的原因有很多种，比如在某个节点的硬件或软件出错之后，节点死掉或者重启，导致计算时间多几倍。如果把 MapReduce 这样的需要同步的系统部署到廉价、可靠性低的硬件上，Straggler Effect 可能会很严重。

13.2.2 异步算法

如果把图 13.7 中的同步屏障去掉，得到的算法就叫做**异步算法 (Asynchronous Algorithm)**，如图 13.8 所示。在异步算法中，一个 Worker 节点无需等待其余节点完成计算或通信。当一个 Worker 节点完成计算，它立刻跟 Server 通信，然后开始下一轮的计算。异步算法避免了等待，节点几乎没有空闲的时间，因此系统的利用率很高。

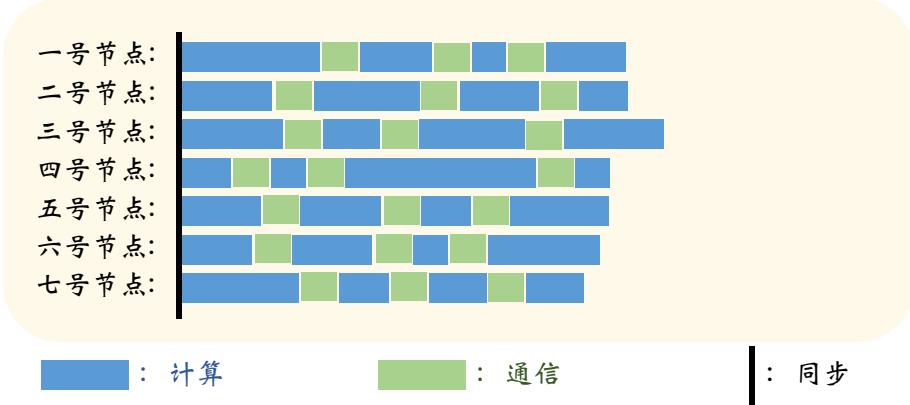


图 13.8: 异步算法中的计算、通信、同步。图中横向表示时间。

下面介绍异步梯度下降算法。我们仍然采用数据并发的方式，即把数据集 $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ 划分到 m 个 Worker 节点上。如图 13.9 所示，服务器可以单独与某个 Worker 节点通信：Worker 节点把计算出梯度发送给服务器，服务器把最新的参数发送给这个 Worker 节点。如果想要编程实现异步算法，可以用 Message Passing Interface (MPI) 这样底层的库，也可以借助 Ray³ 这样的框架。用户需要做的工作是编程实现 Worker 端、服务器端的计算。



图 13.9: 异步梯度下降。

Worker 端的计算：每个 Worker 节点独立做计算，独立与服务器通信；Worker 节点之间不通信，不等待。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的模型参数。把接收到的参数记作 w_{now} 。
2. 利用本地的数据 $\{(\mathbf{x}_j, y_j)\}_{j \in \mathcal{I}_k}$ 和参数 w_{now} 计算本地的梯度：

$$\tilde{\mathbf{g}}^k = \frac{1}{|\mathcal{I}_k|} \sum_{j \in \mathcal{I}_k} (\mathbf{x}_j^T w_{\text{now}} - y_j) \mathbf{x}_j.$$

³<https://ray.io/>

3. 把计算出的梯度 $\tilde{\mathbf{g}}^k$ 发送给服务器。

服务器端的计算：服务器上储存一份模型参数，并且用 Worker 发来的梯度更新参数。每当收到一个 Worker（比如第 k 号 Worker）发送来的梯度（记作 $\tilde{\mathbf{g}}^k$ ），服务器就立刻做梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \tilde{\mathbf{g}}^k.$$

服务器还需要监听 Worker 发送的请求。如果有 Worker 索要参数，就把当前的参数 \mathbf{w}_{new} 发送给这个 Worker。

13.2.3 同步与异步梯度下降的对比

上一节介绍的同步并行梯度下降完全等价于标准的梯度下降，只是把计算分配到了多个 Worker 节点上而已。然而异步梯度下降算法与标准的梯度下降是不等价的。同步与异步梯度下降不只是编程实现有区别，更是在算法上有本质区别。

1. 不难证明，**同步并行梯度下降**更新参数的方式为：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{now}}),$$

即标准的梯度下降。在同一时刻，所有 Worker 节点上的参数是相同的，都是 \mathbf{w}_{now} 。所有 Worker 节点都基于相同的 \mathbf{w}_{now} 计算梯度。

2. 对于**异步并行梯度下降**，在同一时刻，不同 Worker 节点上的参数 \mathbf{w} 通常是不同的。比如两个 Worker 分别在 t_1 和 t_2 时刻向服务器索要参数。在两个时刻之间，服务器可能已经对参数做了多次更新，导致在 t_1 和 t_2 时刻取回的参数不同。两个 Worker 节点会基于不同的参数计算梯度。

在理论上，异步梯度下降的收敛速度慢于同步算法，即需要更多的计算量才能达到相同的精度。但是实践中异步梯度下降远比同步算法快（指的是钟表时间），这是因为异步算法无需等待，Worker 节点几乎不会空闲，利用率很高。

13.3 并行强化学习

并行强化学习的目的在于用更少的钟表时间完成训练。第 13.3.1、13.3.2 小节分别用异步并行算法训练 DQN、Actor-Critic。本节介绍的异步算法与上一节的异步算法很类似，都是由 Worker 节点计算梯度，由服务器更新模型参数。

13.3.1 异步并行双 Q 学习

DQN 和双 Q 学习： DQN 是一个神经网络，记作 $Q(s, a; \mathbf{w})$ ，其中 s 是状态， a 是动作， \mathbf{w} 表示神经网络参数（包含多个向量、矩阵、张量）。通常用双 Q 学习等算法训练 DQN。双 Q 学习需要目标网络 $Q(s, a; \mathbf{w}^-)$ ，它的结构与 DQN 相同，但是参数不同。双 Q 学习属于异策略，即由任意策略控制智能体收集经验，事后做经验回放更新 DQN 参数。第 6 章介绍的高级技巧可以很容易地与双 Q 学习结合，此处就不详细解释了。

系统架构： 如图 13.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器可以随时给某个 Worker 发送一条信息，一个 Worker 也可以随时给服务器发送信息，但是 Worker 之间不能通信。服务器和 Worker 都存储 DQN 的参数。服务器上的参数是最新的，服务器用 Worker 发来的梯度对参数做更新。Worker 节点有自己的目标网络，而服务器上不存储目标网络。每个 Worker 节点有自己的环境，比如运行一个超级玛丽游戏，用 DQN 控制智能体与环境交互，收集经验，把 (s, a, r, s') 这样的四元组存储到本地的经验回放数组。在收集经验的同时，Worker 节点做经验回放，计算梯度，把梯度发送给服务器。

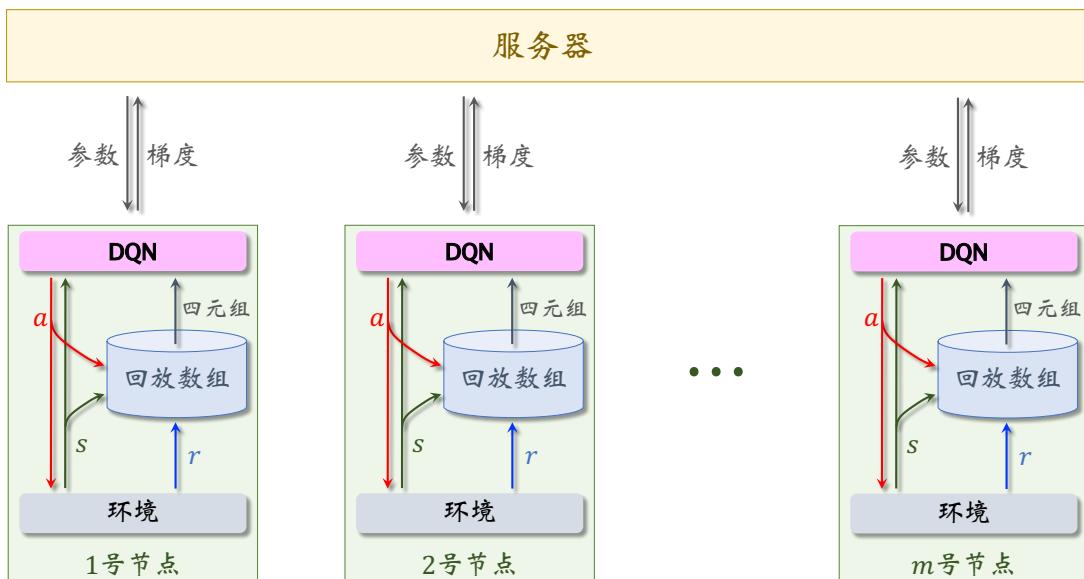


图 13.10：用异步并行算法训练 DQN。图中没有画出目标网络。

Worker 端的计算： 每个 Worker 节点本地有独立的环境，独立的经验回放数组，还有一个 DQN 和一个目标网络。（图 13.10 中没有画出目标网络。）设某个 Worker 节点当

前参数为 \mathbf{w}_{now} 。它用 ϵ -greedy 策略控制智能体与本地环境交互，收集经验。 ϵ -greedy 的定义是：

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

把收集到的经验 (s_t, a_t, r_t, s_{t+1}) 存入本地的经验回放数组。

与此同时，所有的 Worker 节点都要参与异步梯度下降。Worker 节点在本地做计算，还要与服务器通信。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的 DQN 参数。把接收到的参数记作 \mathbf{w}_{new} 。
2. 更新本地的目标网络：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

3. 在本地做经验回放，计算本地梯度：

- (a). 从本地的经验回放数组中随机抽取 b 个四元组，记作

$$(s_1, a_1, r_1, s'_1), (s_2, a_2, r_2, s'_2), \dots, (s_b, a_b, r_b, s'_b).$$

b 是批量的大小，由用户自己设定，比如 $b = 16$ 。

- (b). 用双 Q 学习计算 TD 目标。对于所有的 $j = 1, \dots, b$ ，分别计算

$$\hat{y}_j = r_j + \gamma \cdot Q(s'_j, a'_j; \mathbf{w}_{\text{new}}^-), \quad \text{其中 } a'_j = \operatorname{argmax}_a Q(s'_j, a; \mathbf{w}_{\text{new}}^-).$$

- (c). 定义目标函数：

$$L(\mathbf{w}) \triangleq \frac{1}{2b} \sum_{j=1}^b \left[Q(s_j, a_j; \mathbf{w}) - \hat{y}_j \right]^2.$$

- (d). 计算梯度：

$$\tilde{\mathbf{g}}^k = \nabla_{\mathbf{w}} L(\mathbf{w}_{\text{new}}).$$

4. 把计算出的梯度 $\tilde{\mathbf{g}}^k$ 发送给服务器。

服务器端的计算：服务器上储存有一份模型参数，记作 \mathbf{w}_{now} 。每当一个 Worker 节点发来请求，服务器就把 \mathbf{w}_{now} 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 $\tilde{\mathbf{g}}^k$ ，服务器就立刻做梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \tilde{\mathbf{g}}^k.$$

13.3.2 A3C: 异步并行 A2C

A2C 有一个策略网络 $\pi(a|s; \theta)$ 和一个价值网络 $v(s; \mathbf{w})$ 。通常用策略梯度更新策略网络，用 TD 算法更新价值网络。为了让 TD 算法更稳定，需要一个目标网络 $v(s; \mathbf{w}^-)$ ，它的结构与价值网络相同，但是参数不同。A2C 属于同策略，不能使用经验回放。A2C 的实现详见第 8.3 节。异步并行 A2C 被称作 **Asynchronous Advantage Actor-Critic (A3C)**。

系统架构：如图 13.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器维护策略网络和价值网络最新的参数，并用 Worker 节点发来的梯度更新参数。每个 Worker 节点有一份参数的拷贝，并每隔一段时间向服务器索要最新的参数。每个 Worker 节点有

一个目标网络，而服务器上不储存目标网络。每个 Worker 节点有独立的环境，用本地的策略网络控制智能体与环境交互，用状态、动作、奖励计算梯度。

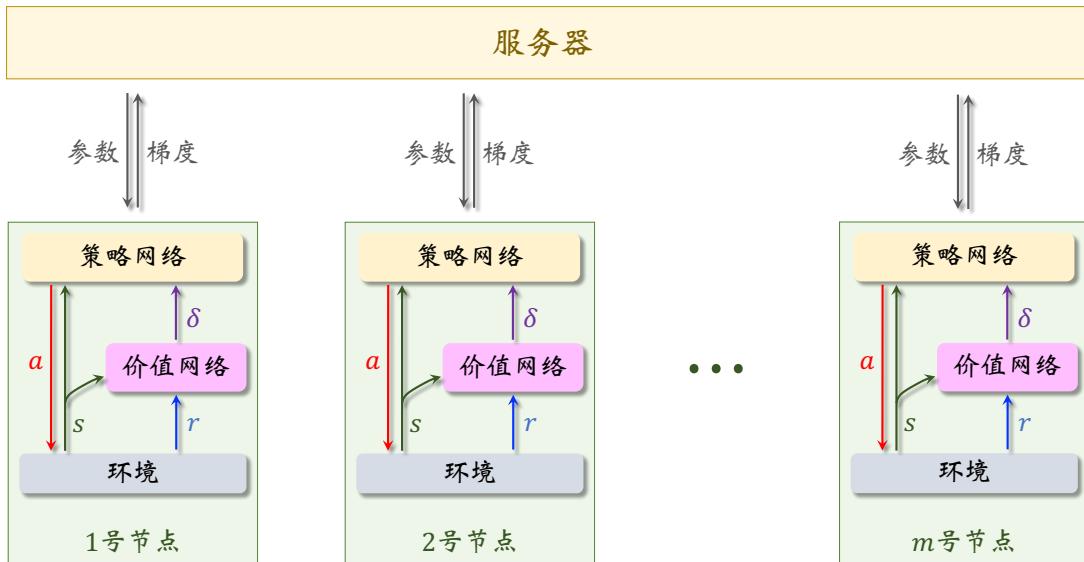


图 13.11: A3C，即异步并行 A2C。图中没有画出目标网络。

Worker 端的计算: 每个 Worker 节点有独立的环境，独立做计算，随时可以与服务器通信。每个 Worker 节点本地有一个策略网络 $\pi(a|s; \theta)$ 、一个价值网络 $v(s; w)$ 、一个目标网络 $v(s; w^-)$ 。设第 k 个 Worker 节点当前参数为 θ_{now} 、 w_{now} 、 w_{now}^- 。第 k 个 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的参数。把接收到的参数记作 θ_{new} 、 w_{new} 。
2. 更新本地的目标网络：
$$w_{\text{new}}^- \leftarrow \tau \cdot w_{\text{new}} + (1 - \tau) \cdot w_{\text{now}}^-.$$
3. 重复下面的步骤 b 次 (b 是用户设置的超参数)，或是从头到尾完成一回合游戏。让智能体与环境交互，计算策略梯度，并累积策略梯度。全零初始化 $\tilde{g}_\theta^k \leftarrow \mathbf{0}$ 、 $\tilde{g}_w^k \leftarrow \mathbf{0}$ ，用它们累积梯度。
 - (a). 基于当前状态 s_t ，根据策略网络做决策 $a_t \sim \pi(\cdot | s_t, \theta)$ ，让智能体执行动作 a_t 。随后观测到奖励 r_t 和新状态 s_{t+1} 。
 - (b). 计算 TD 目标 \hat{y}_t 和 TD 误差 δ_t : ⁴

$$\begin{aligned} \hat{y}_t &= r_t + \gamma \cdot v(s_{t+1}; w_{\text{new}}^-), \\ \delta_t &= v(s_t; w_{\text{new}}) - \hat{y}_t. \end{aligned}$$

(c). 累积梯度：

$$\begin{aligned} \tilde{g}_w^k &\leftarrow \tilde{g}_w^k + \delta_t \cdot \nabla_w v(s_t; w_{\text{new}}), \\ \tilde{g}_\theta^k &\leftarrow \tilde{g}_\theta^k + \delta_t \cdot \nabla_\theta \ln \pi(a_t | s_t; \theta_{\text{new}}). \end{aligned}$$

⁴此处可以用多步 TD 目标等技巧；详见第 5.3 节。

4. 把累积的梯度 $\tilde{\mathbf{g}}_{\theta}^k$ 和 $\tilde{\mathbf{g}}_w^k$ 发送给服务器。

服务器端的计算： 服务器上储存有一份模型参数，记作 θ_{now} 和 w_{now} 。每当一个 Worker 节点发来请求，服务器就把 θ_{now} 和 w_{now} 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 $\tilde{\mathbf{g}}_{\theta}^k$ 和 $\tilde{\mathbf{g}}_w^k$ ，服务器就立刻做梯度下降更新参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \tilde{\mathbf{g}}_w^k,$$

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \tilde{\mathbf{g}}_{\theta}^k.$$

∽ 第十三章 相关文献 ∽

MapReduce 原本是指 Google 内部使用的软件系统，现在泛指这类系统架构。Google 的 MapReduce 系统不对外开源，但是外界可以通过 2008 年的论文 [34] 了解系统的设计。外界有多个开源项目力图实现 MapReduce 系统，其中最有名的是 Hadoop。后来基于 Hadoop 等项目开发的 Spark [132] 比 Hadoop MapReduce 的速度更快。本章介绍的异步并行算法主要基于 Parameter Server [66] 的思想。Ray [78] 是一个开源的软件系统，包含 Parameter Server 的功能。用 Ray 很容易实现异步并行算法，而且 Ray 对强化学习有很好的支持。

本章介绍的并行强化学习算法主要基于 2015 年的论文 [80] 和 2016 年的论文 [75]。两篇论文都是异步算法，主要区别在于 2015 年的论文 [80] 使用经验回放，而 2016 年的论文 [75] 不用经验回放。对于 Atari 游戏这类问题，获取经验非常容易，于是使用经验回放与否其实无关紧要。

第十四章 多智能体系统

之前章节的设定都是单智能体系统 (Single-Agent System, 缩写 SAS)。本章和后面三章介绍多智能体系统 (Multi-Agent System, 缩写 MAS) 和多智能体强化学习 (Multi-Agent Reinforcement Learning, 缩写 MARL)。本章讲解多智能体系统的基本概念，帮助大家理解 MAS 与 SAS 的区别。第 14.1 节讲解 MAS 的四种常见设定。第 14.2 节定义 MAS 的专业术语，将之前所学的观测、动作、奖励、策略、价值等概念推广到 MAS。第 14.3 节介绍几种常用的实验环境，用于对比 MARL 方法的优劣。

14.1 多智能体系统的设定

多智能体系统与单智能体系统的区别： 多智能体系统 (Multi-Agent System, 缩写 MAS) 中包含 m 个智能体，智能体共享环境，智能体之间会相互影响。智能体之间是如何相互影响的呢？一个智能体的动作会改变环境状态，从而影响其余所有智能体。举个例子，股市中的每个自动交易程序就可以看做一个智能体。尽管智能体（自动交易程序）之间不会交流，它们依然会相互影响：一个交易程序的决策会影响股价，从而对其他自动交易程序有利或有害。

注意，MAS 与上一章的并行强化学习是不同的概念。上一章用 m 个节点并行计算，每个节点有独立的环境，每个环境中有一个智能体。虽然 m 个节点上一共有 m 个智能体，但是智能体之间完全独立，不会相互影响。而本章 MAS 只有一个环境，环境中有 m 个相互影响的智能体。并行强化学习的设定是 m 个单智能体系统 (Single-Agent System, 缩写 SAS) 的并集，可以视作 MAS 的一种特例。举个例子，环境中有 m 个机器人，这属于 MAS 的设定。假如把每个机器人隔绝在一个密闭的房间中，机器人之间不会通信，那么 MAS 就变成了多个 SAS 的并集。

多智能体强化学习 (Multi-Agent Reinforcement Learning, 缩写 MARL) 是指让多个智能体处于相同的环境中，每个智能体独立与环境交互，利用环境反馈的奖励改进自己的策略，以获得更高的回报（即累计奖励）。在多智能体系统中，一个智能体的策略不能简单依赖于自身的观测、动作，还需要考虑到其他智能体的观测、动作。因此，MARL 比单智能体强化学习 (Single-Agent Reinforcement Learning, 缩写 SARL) 更困难。

多智能体系统有四种常见设定： 合作关系 (Fully Cooperative)、竞争关系 (Fully Competitive)、合作竞争的混合 (Mixed Cooperative & Competitive)、利己主义 (Self-Interested)。图 14.1 举例说明了四种常见设定。接下来具体讲解这些设定。

第一种设定是**完全合作关系**：智能体的利益一致，获得的奖励相同，有共同的目标。比如图 14.1 中，多个工业机器人协同装配汽车。他们的目标是相同的，都希望把汽车装好。假设一共有 m 个智能体，它们在 t 时刻获得的奖励分别是 $R_t^1, R_t^2, \dots, R_t^m$ 。（用上标表示智能体，用下标表示时刻。）在完全合作关系中，它们的奖励是相同的：

$$R_t^1 = R_t^2 = \dots = R_t^m, \quad \forall t.$$

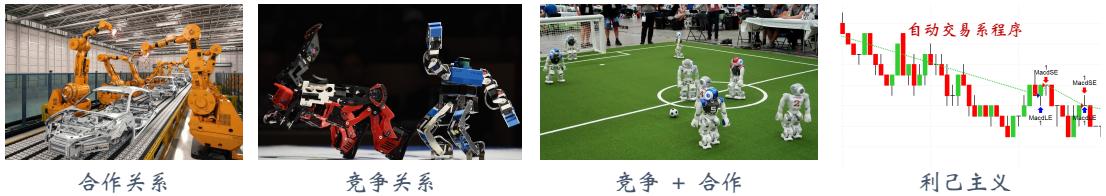


图 14.1: 多智能体强化学习的四种常见设定。四张图片来源于网络。

第二种设定是**完全竞争关系**: 一方的收益是另一方的损失。比如图 14.1 中的两个格斗机器人，它们的利益是冲突的，一方的胜利就是另一方的失败。在完全竞争的设定下，双方的奖励是负相关的：对于所有的 t ，有 $R_t^1 \propto -R_t^2$ 。如果是零和博弈，双方的获得的奖励总和等于 0： $R_t^1 = -R_t^2$ 。

第三种设定是**合作竞争的混合**。智能体分成多个群组；组内的智能体是合作关系，它们的奖励相同；组间是竞争关系，两组的奖励是负相关的。比如图 14.1 中的足球机器人：两组是竞争关系，一方的进球是另一方的损失；而组内是合作关系，队友的利益是一致的。

第四种设定是**利己主义**。系统内有多个智能体；一个智能体的动作会改变环境状态，从而让别的智能体受益或者受损。利己主义的意思是智能体只想最大化自身的累计奖励，而不在乎他人收益或者受损。比如图 14.1 中的股票自动交易程序可以看做是一个智能体；环境（股市）中有多个智能体。这些智能体的目标都是最大化自身的收益，因此可以看做利己主义。智能体之间会相互影响：一个智能体的决策会影响股价，从而影响其他自动交易程序的收益。智能体之间有潜在而又未知的竞争与合作关系：一个智能体的决策可能会帮助其他智能体获利，也可能导致其他智能体受损。设计自动交易程序的时候，不应当把它看做孤立的系统，而应当考虑到其他自动交易程序的行为。

不同设定下学出的策略会有所不同。在**合作**的设定下，每个智能体的决策要考虑到队友的策略，要与队友做到尽量好的配合，而不是个人英雄主义；这个道理在足球、电子竞技中是显然的。在**竞争**的设定下，智能体要考虑到对手的策略，相应调整自身策略；比如在象棋游戏中，如果你很熟悉对手的套路，并相应调整自己的策略，那么你的胜算会更大。在**利己主义**的设定下，一个智能体的决策无需考虑其他智能体的利益，尽管一个智能体的动作可能会在客观上帮助或者妨害其他智能体。

14.2 多智能体系统的基本概念

本书第 3 章定义了单智能体系统的专业术语，比如状态、动作、奖励、策略、价值。在本节中，我们将这些定义推广到多智能体系统。在此后的章节中，我们用 m 表示智能体的数量，用上标 i 表示智能体的序号（ i 从 1 到 m ），依然用下标 t 表示时刻。

14.2.1 专业术语

本章依然用大写字母 S 表示**状态** (State) 随机变量，用小写字母 s 表示状态的观测值。注意，单个智能体未必能观测到完整状态。如果单个智能体的观测只是部分状态，我们就用 o^i 表示第 i 号智能体的不完全观测。

每个智能体都会做出**动作** (Action)。把第 i 号智能体的动作随机变量记作 A^i ，把动作的实际观测值记作 a^i 。如果不加上标 i ，则意味着所有智能体的动作的连接：

$$A = [A^1, A^2, \dots, A^m], \quad a = [a^1, a^2, \dots, a^m].$$

把第 i 号智能体的动作空间 (Action Space) 记作 \mathcal{A}^i ，它包含该智能体所有可能的动作。整个系统的动作空间是 $\mathcal{A} = \mathcal{A}^1 \times \dots \times \mathcal{A}^m$ 。两个智能体的动作空间 \mathcal{A}^i 和 \mathcal{A}^j 可能相同，也可能不同。比如在电子游戏中，有的士兵会远程攻击，而有的士兵只能近距离攻击，不同类型的士兵可以有不同的动作空间。

所有智能体都执行动作之后，环境依据**状态转移函数** (State-Transition Function) 给出下一时刻的状态。状态转移函数是个条件概率密度函数，记作

$$p(s_{t+1} | s_t; a_t) = \mathbb{P}[S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t].$$

它的意思是下一时刻状态 S_{t+1} 取决于当前时刻状态 S_t 、以及所有 m 个智能体的动作 $A_t = [A_t^1, A_t^2, \dots, A_t^m]$ 。

奖励 (Reward) 是环境反馈给智能体的数值。把第 i 号智能体的奖励随机变量记作 R^i ，把奖励的实际观测值记作 r^i 。在合作的设定下， $R^1 = R^2 = \dots = R^m$ ；在竞争的设定下， $R^1 \propto -R^2$ 。第 t 时刻的奖励 R_t^i 由状态 S_t 和所有智能体的动作 $A = [A^1, A^2, \dots, A^m]$ 共同决定。为什么一个智能体获得的奖励会取决于其他智能体的动作呢？举个例子，在足球比赛中，假如对方失误，自己进了个乌龙球；而你什么也没做，就获得了一分的奖励。

折扣回报 (Discounted Return) 也叫折扣累计奖励，它的定义类似于单智能体系统。第 i 号智能体的折扣回报是它自己的奖励的加权和：

$$U_t^i = R_t^i + \gamma \cdot R_{t+1}^i + \gamma^2 \cdot R_{t+2}^i + \gamma^3 \cdot R_{t+3}^i + \dots$$

此处的 $\gamma \in [0, 1]$ 是折扣率 (Discount Factor)。

14.2.2 策略网络

策略网络的意思是用神经网络近似策略函数。可以让每个智能体有自己的策略网络。对于**离散控制问题**，把第 i 号智能体的策略网络记作：

$$\hat{\mathbf{f}} = \pi(\cdot | s; \theta^i).$$

策略网络的输入是状态 s , 输出是向量 $\hat{\mathbf{f}}$ 。向量 $\hat{\mathbf{f}}$ 的维度是动作空间的大小 $|\mathcal{A}^i|$, $\hat{\mathbf{f}}$ 的每个元素表示一个动作的概率。 $\hat{\mathbf{f}}$ 的元素都是正实数, 而且相加等于 1。做决策的时候, 根据 $\hat{\mathbf{f}}$ 做随机抽样, 得到动作 a^i , 第 i 号智能体执行这个动作。

对于连续控制问题, 即动作空间 \mathcal{A}^i 是连续集, 把第 i 号智能体的策略网络记作:

$$\mathbf{a}^i = \mu(s; \boldsymbol{\theta}^i), \quad \forall i = 1, \dots, m.$$

有了这个策略网络, 第 i 号智能体就可以基于当前状态 s , 直接计算出需要执行的动作 \mathbf{a}^i 。

在上面的两种策略网络中, 每个智能体的策略网络有各自的参数: $\boldsymbol{\theta}^1, \boldsymbol{\theta}^2, \dots, \boldsymbol{\theta}^m$ 。在有些情况下, 策略网络的角色是可以互换的, 比如同一型号无人机的功能是相同的, 那么它们的策略网络是相同的: $\boldsymbol{\theta}^1 = \boldsymbol{\theta}^2 = \dots = \boldsymbol{\theta}^m$ 。但是在很多应用中, 策略网络不能互换。比如在足球机器人的应用中, 球员有的是负责进攻的前锋, 有的是负责防守的后卫, 还有一个守门员。它们的策略网络不能互换, 所以参数 $\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m$ 各不相同。

14.2.3 动作价值函数

上面讨论过, 第 i 号智能体在第 t 时刻得到的奖励 R_t^i 依赖于状态 S_t 、以及所有智能体的动作 $A_t = [A_t^1, \dots, A_t^m]$ 。因为(折扣)回报 U_t^i 是未来所有奖励 $R_t^i, R_{t+1}^i, \dots, R_n^i$ 之和, 所以 U_t^i 依赖于未来所有状态

$$S_t, S_{t+1}, S_{t+2}, \dots, S_n$$

与所有智能体未来的动作

$$A_t, A_{t+1}, A_{t+2}, \dots, A_n.$$

在 t 时刻, 回报 U_t^i 是个随机变量, 其随机性的来源是未来所有状态、所有智能体未来的动作。

如果用期望消掉回报 U_t^i 中的随机性, 就能得到价值函数。把 t 时刻的状态 s_t 和所有智能体的动作 $a_t = [a_t^1, \dots, a_t^m]$ 当做观测值, 用期望消掉 $t+1$ 时刻之后未知的状态和动作, 得到的结果就是动作价值函数 (Action-Value Function):

$$Q_\pi^i(s_t, a_t) = \mathbb{E}[U_t^i \mid S_t = s_t, A_t = a_t]. \quad (14.1)$$

此处的期望是关于这些随机变量求的:

- 未来的状态 $S_{t+1}, S_{t+2}, \dots, S_n$ 。
- 未来动作 $A_{t+1}, A_{t+2}, \dots, A_n$; 这里的 $A_k = [A_k^1, \dots, A_k^m]$ 是所有智能体在 k 时刻的动作。

公式 (14.1) 中关于动作 $A_k = [A_k^1, \dots, A_k^m]$ 求期望, $\forall k$, 要用到动作 A_k 的概率质量函数, 即所有 m 个智能体的策略的乘积:

$$\pi(A_k^1 \mid S_k; \boldsymbol{\theta}^1) \times \pi(A_k^2 \mid S_k; \boldsymbol{\theta}^2) \times \dots \times \pi(A_k^m \mid S_k; \boldsymbol{\theta}^m).$$

也就是说, 第 i 号智能体的动作价值 $Q_\pi^i(s_t, a_t)$ 依赖于所有 m 个智能体的策略。

为什么第 i 号智能体的动作价值 $Q_\pi^i(s, a)$ 会依赖于其余智能体的策略呢? 这里给一个直观的解释。在足球游戏中, 假如你有个猪队友 (即策略很差), 那么你未来获得不了

多少奖励，所以你的 Q_π^i 会比较小。假如把猪队友换成靠谱的队友（即策略更好），你的 Q_π^i 会变大。虽然你没有改变自己的策略，但是你的动作价值 Q_π^i 会随着队友的策略变化。

总结一下。如果系统里有 m 个智能体，那么就有 m 个动作价值函数：

$$Q_\pi^1(s, a), \quad Q_\pi^2(s, a), \quad \dots, \quad Q_\pi^m(s, a).$$

第 i 号智能体的动作价值 $Q_\pi^i(s_t, a_t)$ 并非仅仅依赖于自己当前的动作 a_t^i 与策略 $\pi(a_t^i | s_t; \theta^i)$ 。
 $Q_\pi^i(s_t, a_t)$ 依赖于其余智能体当前的动作

$$a_t = [a_t^1, a_t^2, \dots, a_t^m]$$

与所有智能体的策略

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m).$$

14.2.4 状态价值函数

我们在第 3 章中学过单智能体系统的状态价值函数 (State-Value Function)，记作 $V_\pi(S)$ ，并在策略学习的方法中反复用到 $V_\pi(S)$ 。它是对动作价值函数 $Q_\pi(S, A)$ 关于当前动作 A 的期望：

$$V_\pi(s) = \mathbb{E}_A [Q_\pi(s, A)] = \sum_{a \in \mathcal{A}} \pi(A | s; \theta) \cdot Q_\pi(s, a).$$

下面我们将状态价值函数的定义推广到多智能体系统。

第 i 号智能体的动作价值函数是 $Q_\pi^i(S, A)$ 。想要对 $Q_\pi^i(S, A)$ 关于 $A = [A^1, \dots, A^m]$ 求期望，需要用到 A 的概率质量函数，即所有 m 个智能体的策略的乘积：

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$

状态价值函数可以写成：

$$V_\pi^i(s) = \mathbb{E}_A [Q_\pi^i(s, A)] = \sum_{a^1 \in \mathcal{A}^1} \sum_{a^2 \in \mathcal{A}^2} \dots \sum_{a^m \in \mathcal{A}^m} \pi(a | s; \theta^1, \dots, \theta^m) \cdot Q_\pi^i(s, a).$$

很显然，第 i 号智能体的状态价值 $V_\pi^i(s)$ 依赖于所有智能体的策略：

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m).$$

MARL 的困难之处就在于一个智能体的价值 Q_π^i 与 V_π^i 受其他智能体策略的影响。举个例子，在足球运动中，其他所有人的策略都没变化，只有一个前锋改进了自己的策略，让他自己水平更高。那么他的队友的价值会变大，而对手的价值会变小。一个智能体 i 单独改进自己的策略，未必能让自己的价值 Q_π^i 与 V_π^i 变大，因为其他智能体的策略可能已经发生了变化。

14.3 实验环境

如果你设计出一种新的 MARL 方法，你应该将其与已有的标准方法做比较，看新的方法是否有优势。下面介绍几种 MARL 的实验环境，用于评价 MARL 方法的优劣。建议读者跳过本节内容，等到需要做 MARL 的实验的时候再阅读本节。

14.3.1 Multi-Agent Particle World

Multi-Agent Particle World 是一类简单的多智能体控制问题，其中包含很多种环境，如图 14.2 所示。这些环境由 Lowe 等人 [72] 开发，源代码公开在 GitHub 上：<https://github.com/openai/multiagent-particle-envs.git>。下面介绍图 14.2 中的四个环境。

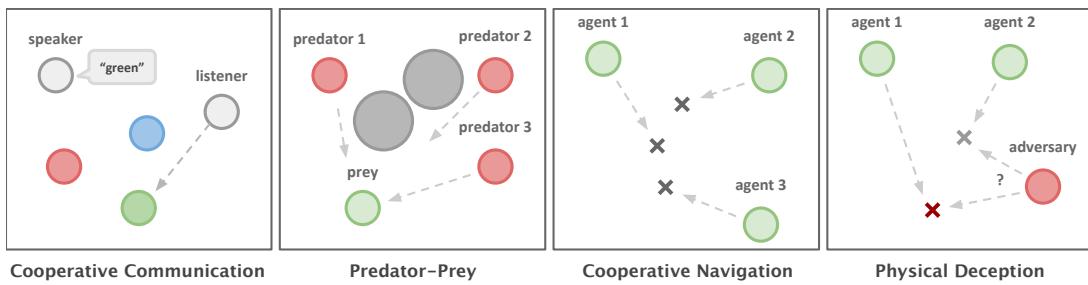


图 14.2：Multi-Agent Particle World 中的四种常用环境。图片来源于 2017 年的论文 [72]。

Cooperative Communication 这个环境中有三个点，每个点有一种颜色，这三个点不会移动。环境中有两个合作关系的智能体，一个叫做“Speaker”，另一个叫做“Listener”，它们是合作关系。任务是给定一种颜色 c ，让 Listener 移动到这种颜色的点上；离该点越近，则奖励越大。

- Speaker 的观测是 c ，即 Speaker 知道任务要求的颜色是什么。
- Speaker 的动作是发送一条信息，比如向量 $[0.1, 0.9, 0]$ 。很显然，训练 Speaker 的目的是让它发送的信息是颜色 c 的编码。
- Listener 的观测是三个点的颜色、三个点的位置（指的是相对位置）、以及 Speaker 发送的信息。比如，这是 Listener 的一个观测：

$$\left(\underbrace{[-1.5, -0.5]}_{\text{红点的位置}}, \underbrace{[-0.9, -0.9]}_{\text{绿点的位置}}, \underbrace{[-0.8, -0.2]}_{\text{蓝点的位置}}, \underbrace{[0, 1, 0]}_{\text{Speaker 发送的信息}} \right).$$

- Listener 的动作空间是这个离散集合：{不动, 上, 下, 左, 右}。

Predator-Prey 这个环境中多个智能体，它们分为两类——多个 Predators (捕食者) 与一个 Prey (猎物)。这个问题属于混合关系，即同时存在合作与竞争关系。Predators 数量多，占有优势；为了平衡双方实力，环境的设置让 Predators 速度慢于 Prey。环境中有关碍物，智能体必须绕路。

- **奖励：**如果一个 Predator 碰到 Prey (猎物)，所有的 Predators 都会收到奖励，而 Prey 受到惩罚。

- **观测:** 每个智能体都能观测到障碍物的位置、其余智能体的位置。此处的“位置”指的是相对位置。
- **动作:** 每个智能体的动作空间都是 {不动, 上, 下, 左, 右}。

Cooperative Navigation 环境中有 m 个合作关系的智能体与 m 个不动的点。

- **奖励:** 每个不动点都带有奖励，离该点最近的智能体会收集到奖励，奖励的大小与距离负相关。也就是说，最好的策略是让 m 个智能体分别覆盖 m 个点。智能体应当远离彼此；如果两个智能体碰撞，则会受到惩罚。
- **观测:** 每个智能体都能观测到其他智能体的位置、以及 m 个点的位置。此处的“位置”指的是相对位置。
- **动作:** 每个智能体的动作空间都是 {不动, 上, 下, 左, 右}。

Physical Deception 这个环境中 $m+1$ 个智能体，其中 m 个是合作关系的玩家，一个是对手。这个问题属于混合关系。

- **奖励:** 环境中有 m 个点，其中一个点 x 带有奖励，离 x 距离最近的玩家获得奖励，奖励的大小与距离负相关。也就是说，应当有一个玩家到达点 x ；但这是不够的。对手也想到达点 x ；对手离 x 越近，对手得到的奖励越大，而对手的奖励是玩家的惩罚。
- **玩家的观测:** 玩家知道所有玩家的位置、所有点的位置、以及哪个点是带奖励的点 x 。此处的“位置”指的是相对位置。
- **对手的观测:** 对手知道所有玩家的位置、所有点的位置，但是不知道哪个点是 x 。此处的“位置”指的是相对位置。
- **动作:** 每个智能体的动作空间都是 {不动, 上, 下, 左, 右}。

虽然只有当覆盖 x 的时候有奖励，但是玩家不能仅仅覆盖点 x ，而不覆盖其余的点。否则对手会推测出 x 是哪一个点。因此，玩家最好的策略是覆盖所有 m 个点，从而迷惑对手。

14.3.2 StarCraft Multi-Agent Challenge (SMAC)

星际争霸 2 (StarCraft II) 是由暴雪在 2010 年推出的一款即时战略游戏。游戏中有很多兵种（即很多类型的智能体），每个兵种有自己的生命值、护甲、移动速度、攻击范围、杀伤力等属性。一个士兵在生命值耗尽的时候死去，从游戏中消失。星际争霸游戏中可以有多个玩家，每个玩家控制一支军队；一支军队中有若干兵种，每个兵种有若干士兵。

StarCraft Multi-Agent Challenge (SMAC) 是基于星际争霸 2 开发的库，是对星际争霸游戏的简化。在 SMAC 中，玩家控制一支军队，与游戏 AI 控制的军队对战。消灭掉对方所有的士兵，就算胜利；如果己方全部士兵都死亡，就算失败。SMAC 由 Samvelyan 等人 [89] 开发，源代码公开在 GitHub 上：<https://github.com/oxwhirl/smac.git>。SMAC 库中有很多对战的环境。图 14.3 展示了两种环境。

SMAC 中每个士兵是一个智能体，有自己的观测（多个向量），能做出离散动作。如图 14.4 所示，每个士兵有自己的视野，能观测到一个圆内的所有队友和对手。每个士兵



(a) 双方各有 3 只 Stalkers 和 5 只 Zealots。 (b) 一方有 7 只 Zealots，另一方有 32 只 Banelings。

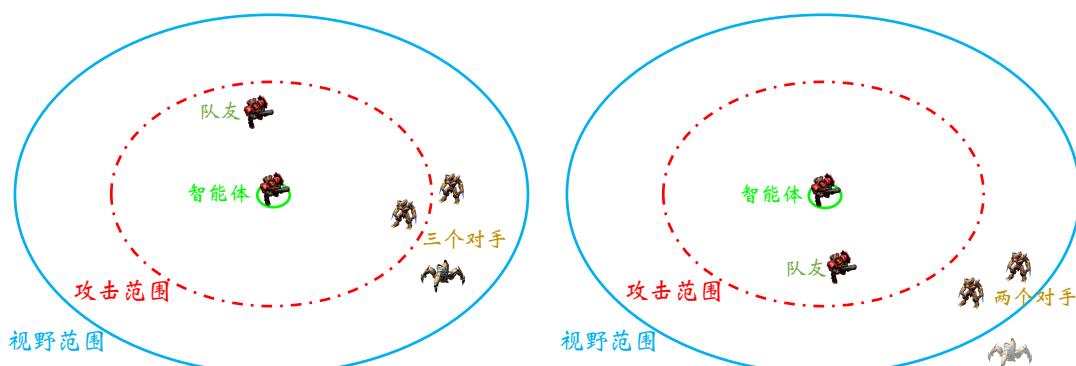
图 14.3: SMAC 库中的两种环境。

有自己的攻击范围，但仅限于攻击范围之内。每个智能体的**观测**表示为多个向量。

- 每个向量对应视野范围内的一个士兵，可以是队友或对手。
- 每个向量包含以下信息：距离、相对位置、生命值 (Health)、护甲 (Shield)、士兵类型 (Unit Type)、上一个动作（仅知道队友的上一个动作）。

每个智能体的**动作空间**是离散的，每个智能体每次可以什么也不做，或者执行下面动作中的一种：

- 向东、西、南、北四个方向中的一个移动。
- 攻击对手（或治疗队友），仅限与攻击范围之内，需要指定被攻击（或被治疗）目标的 ID。



(a) 智能体观测到 4 个士兵，表示成 4 个向量。 (b) 智能体观测到 3 个士兵，表示成 3 个向量。智能体观测不到视野范围之外的士兵。

图 14.4: 玩家控制两个士兵，对手控制三个士兵。玩家的两个士兵相当于两个智能体，它们有各自的观测和动作。

一个团队的士兵是合作关系，奖励是给予团队的，而不是给具体某个士兵。SMAC 有两种类型的奖励可供选择。一种是稀疏的奖励：最终游戏胜利获得奖励 +1，失败获得奖励 -1。另一种是稠密的奖励：杀死对方一个士兵有正奖励，己方士兵被杀有负奖励；外加游戏结束时胜利、失败的奖励。

14.3.3 Hanabi Challenge

花火 (Hanabi) 是一种合作型的卡牌游戏，玩家不能观看自己的牌，只能看其他玩家的。游戏的玩法是要将不同花色的数字牌按顺序排列。每回合中玩家只能获得有限的信息，需要做推理，从而做出决策。花火的规则较为复杂，此处不详细解释。有兴趣的读者可以在互联网上检索“花火卡牌游戏”，了解游戏规则。Hanabi Challenge 由 Bard 等人 [9] 在 2020 年开发，源代码公开在 GitHub 上：<https://github.com/deepmind/hanabi-learning-environment.git>。该程序提供花火游戏的环境，可供 MARL 学术研究。

从强化学习的角度来看，花火属于合作类型的 MARL。一局游戏结束时有奖励，奖励是给予团队的，而非玩家个人。每个玩家相当于一个智能体，他无法观测到全局状态，只能在不完全观测的情况下做出决策。玩家可以看到队友的牌，但是不能看自己的牌；玩家要靠队友提供的情报来推测自己的牌。玩家每一回合可以做出三种动作中的一种：提供情报、弃置一张牌、打出一张牌。提供情报的次数是很有限制的，玩家必须学会传递最有用的情报。玩家获得的奖励由出牌的好坏决定。综上所述，玩家需要学会两种能力：第一，将最有用的情报传递给队友；第二，根据队友传递的情报做出决策。

∽ 第十四章 相关文献 ∽

合作关系的 MARL 在自动控制领域被称作 Team Markov Games [51, 121, 131]。合作关系的 MARL 在 AI 领域最早见于论文 [17, 62]。竞争关系的 MARL 最早见于论文 [69]。混合关系的 MARL 最早见于论文 [54, 61, 70]。

很早就有关文将 Q 学习等价值学习方法推广到 MARL。1993 年的论文 [109] 研究了独立 Q 学习 (Independent Q-Learning, 缩写 IQL)，即智能体独立做 Q 学习，不共享信息。2017 年的论文 [40, 108] 将 IQL 用在深度强化学习。比较有名的多智能体价值学习方法有 Value-Decomposition Networks [102]、QMIX [86] 等方法。目前 MARL 更流行 Actor-Critic 方法，比如 [44, 39, 72, 56]。其中最有名的是 2018 年的 COMA [39] 与 2017 年的 MADDPG [72]。

对 MARL 感兴趣的读者可以阅读这些综述和书籍：Weiss 1999 [125]，Stone & Veloso 2000 [101]，Vlassis 2007 [120]，Shoham & Leyton-Brown 2008 [97]，Buşoniu *et al.* 2010 [23]，Zhang *et al.* 2019 [133]。

第十五章 合作关系设定下的多智能体强化学习

本章只考虑最简单的设定——完全合作关系——并在这种设定下研究多智能体强化学习 (MARL)。第 15.1 节定义“完全合作关系”下的策略学习。第 15.2 节介绍“完全合作关系”下的多智能体 A2C 方法，本书称之为 MAC-A2C。第 15.3 节介绍 MARL 的三种常见架构——完全去中心化、完全中心化、中心化训练 + 去中心化决策——并在三种框架下实现 MAC-A2C。

本章与上一章对状态的定义有所区别。在多智能体系统中，一个智能体未必能观测到全局状态 S 。设第 i 号智能体有一个局部观测，记作 O^i ，它是 S 的一部分。不妨假设所有的局部观测的总和构成全局状态：

$$S = [O^1, O^2, \dots, O^m],$$

MARL 的文献大多采用这种假设。本章中采用的符号如图 15.1 所示。

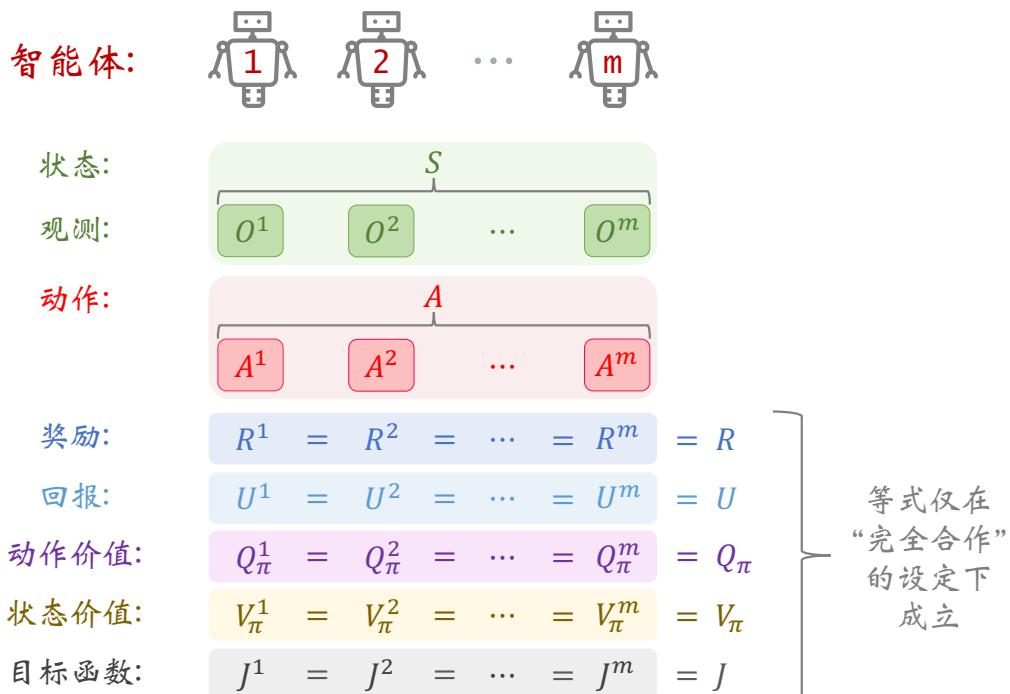


图 15.1: 多智能体强化学习 (MARL) 在“完全合作关系”设定下的符号。

15.1 合作关系设定下的策略学习

MARL 中的完全合作关系 (Fully-Cooperative) 意思是所有智能体的利益是一致的，它们具有相同的奖励：

$$R^1 = R^2 = \dots = R^m \triangleq R.$$

因此，所有的智能体都有相同的回报：

$$U^1 = U^2 = \dots = U^m \triangleq U.$$

因为价值函数是回报的期望，所以所有的智能体都有相同的价值函数。省略上标 i ，把动作价值函数记作 $Q_\pi(S, A)$ ，把状态价值函数记作 $V_\pi(S)$ 。

注意，价值函数 Q_π 和 V_π 依赖于所有智能体的策略：

$$\pi(A^1 | S; \theta^1), \quad \pi(A^2 | S; \theta^2), \quad \dots, \quad \pi(A^m | S; \theta^m).$$

举个例子，在某个竞技电游中，玩家组队打任务；每完成一个任务，团队成员（即智能体）获得相同的奖励。所以大家的 R, U, Q_π, V_π 全都是一样的。回报的期望——即价值函数 Q_π 与 V_π ——显然与所有成员的策略相关：只要有一个猪队友（即策略差）拖后腿，就有可能导致任务失败。通常来说，团队成员有分工合作，所以每个成员的策略是不同的，即 $\theta^i \neq \theta^j$ 。

如果做策略学习（即学习策略网络参数 $\theta^1, \dots, \theta^m$ ），那么所有智能体都有一个共同目标函数：

$$J(\theta^1, \dots, \theta^m) = \mathbb{E}_S[V_\pi(S)].$$

所有智能体的目的是一致的，即改进自己的策略网络参数 θ^i ，使得目标函数 J 增大。那么策略学习可以写作这样的优化问题：

$$\max_{\theta^1, \dots, \theta^m} J(\theta^1, \dots, \theta^m). \quad (15.1)$$

注意，只有“完全合作关系”这种设定下，所有智能体才会有共同的目标函数，其原因在于 $R^1 = \dots = R^m$ 。对于其它设定——“竞争关系”、“混合关系”、“利己主义”——智能体的目标函数是各不相同的（见下一章）。

合作关系设定下的策略学习的原理很简单，即让智能体各自做策略梯度上升，使得目标函数 J 增长。

$$\text{第 1 号智能体执行: } \theta^1 \leftarrow \theta^1 + \alpha^1 \cdot \nabla_{\theta^1} J(\theta^1, \dots, \theta^m),$$

$$\text{第 2 号智能体执行: } \theta^2 \leftarrow \theta^2 + \alpha^2 \cdot \nabla_{\theta^2} J(\theta^1, \dots, \theta^m),$$

$$\vdots \qquad \vdots$$

$$\text{第 } m \text{ 号智能体执行: } \theta^m \leftarrow \theta^m + \alpha^m \cdot \nabla_{\theta^m} J(\theta^1, \dots, \theta^m).$$

公式中的 $\alpha^1, \alpha^2, \dots, \alpha^m$ 是学习率。判断策略学习收敛的标准是目标函数 $J(\theta^1, \dots, \theta^m)$ 不再增长。在实践中，当平均回报不再增长，即可终止算法。由于无法直接计算策略梯度 $\nabla_{\theta^i} J$ ，我们需要对其做近似。下一节用价值网络近似策略梯度，从而推导出一种实际可行的策略梯度方法。

15.2 合作设定下的多智能体 A2C

第 8 章介绍过 Advantage Actor-Critic (A2C) 方法。本节介绍“完全合作关系”设定下的多智能体 A2C 方法 (Multi-Agent Cooperative A2C)，缩写 MAC-A2C。注意，本节介绍的方法仅适用于“完全合作关系”，也就是要求所有智能体有相同的奖励： $R^1 = \dots = R^m$ 。第 15.2.1 小节定义策略网络和价值网络。第 15.2.2 小节描述 MAC-A2C 训练和决策。第 15.2.3 小节讨论 MAC-A2C 实现中的难点。

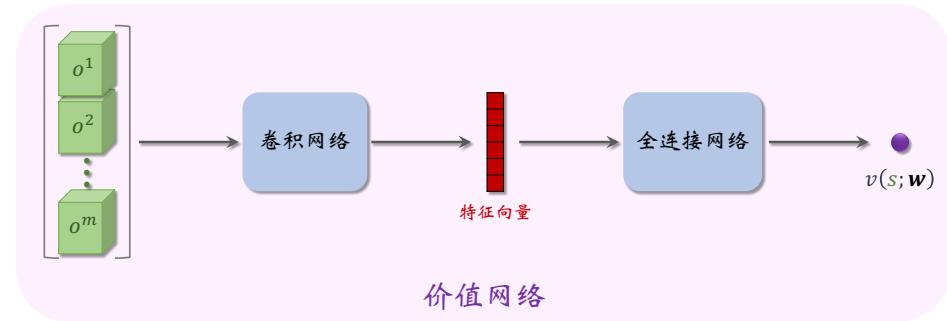


图 15.2: 图中是 MAC-A2C 中的价值网络 $v(s; \mathbf{w})$ 。所有智能体共用这个价值网络。输入是所有智能体的观测： $s = [o^1, \dots, o^m]$ 。输出是价值网络给 s 的评分。

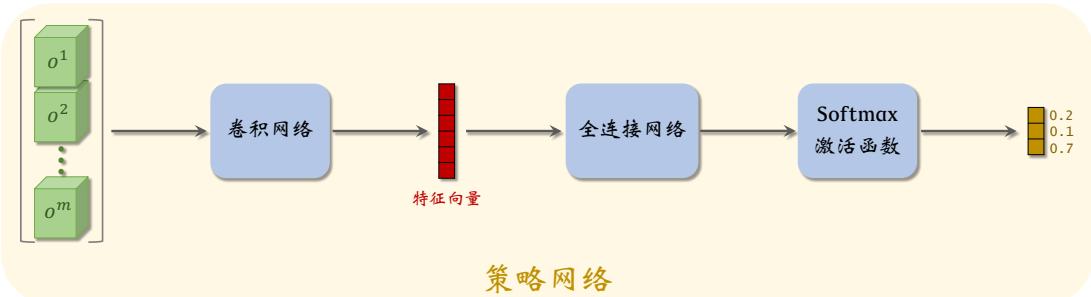


图 15.3: 图中是 MAC-A2C 中第 i 号智能体的策略网络 $\pi(\cdot | s; \theta^i)$ 。所有智能体的策略网络结构都一样，但是参数 $\theta^1, \dots, \theta^m$ 可能不一样。输入是所有智能体的观测： $s = [o^1, \dots, o^m]$ 。输出是在离散动作空间 \mathcal{A}^i 上的概率分布。

15.2.1 策略网络和价值网络

本章只考虑离散控制问题，即动作空间 $\mathcal{A}^1, \dots, \mathcal{A}^m$ 都是离散集合。MAC-A2C 使用两类神经网络：价值网络 v 与策略网络 π ；见图 15.2、图 15.3。

所有智能体共用一个价值网络，记作 $v(s; \mathbf{w})$ ，它是对状态价值函数 $V_\pi(s)$ 的近似。它把所有观测 $s = [o^1, \dots, o^m]$ 作为输入，并输出一个实数，作为对状态 s 的评分。

每个智能体有自己的策略网络。把第 i 号策略网络记作 $\pi(a^i | s; \theta^i)$ 。它的输入是所有智能体的观测 $s = [o^1, \dots, o^m]$ 。它的输出是一个向量，表示动作空间 \mathcal{A}^i 上的概率分布。比如，第 i 号智能体的动作空间是 $\mathcal{A}^i = \{\text{左}, \text{右}, \text{上}\}$ ；策略网络的输出是

$$\pi(\text{左} | s; \theta^i) = 0.2, \quad \pi(\text{右} | s; \theta^i) = 0.1, \quad \pi(\text{上} | s; \theta^i) = 0.7.$$

第 i 号智能体依据该概率分布抽样得到动作 a^i 。

MAC-A2C 属于 Actor-Critic 方法：策略网络 $\pi(A^i | S; \theta^i)$ 相当于第 i 个运动员，负责做决策；价值网络 $v(S; w)$ 相当于评委，对运动员团队的整体表现予以评价，反馈给整个团队一个分数。

训练价值网络： 我们用 TD 算法训练价值网络 $v(s; w)$ 。观测到状态 s_t 、 s_{t+1} 和奖励 r_t ，计算 TD 目标：

$$\hat{y}_t = r_t + \gamma \cdot v(s_{t+1}; w).$$

把 \hat{y}_t 视作常数，更新 w 使得 $v(s_t; w)$ 接近 \hat{y}_t 。定义损失函数：

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

损失函数的梯度等于：

$$\nabla_w L(w) = \delta_t \cdot \nabla_w v(s_t; w),$$

其中 $\delta_t = v(s_t; w) - \hat{y}_t$ 是 TD 误差。做一次梯度下降更新 w ：

$$w \leftarrow w - \alpha \cdot \delta_t \cdot \nabla_w v(s_t; w).$$

这样可以减小损失函数，也就是让 $v(s_t; w)$ 接近 \hat{y}_t 。上述 TD 算法与单智能体 A2C 的 TD 算法完全一样。

训练策略网络： 完全合作关系设定下的动作价值函数记作 $Q_\pi(s, a)$ ，第 i 号智能体的策略网络为 $\pi(A^i | S; \theta^i)$ 。不难证明下面的策略梯度定理（见习题 1）：

定理 15.1. 合作关系 MARL 的策略梯度定理

设基线 b 为不依赖于 $A = [A^1, \dots, A^m]$ 的函数。那么有

$$\nabla_{\theta^i} J(\theta^1, \dots, \theta^m) = \mathbb{E}_{S, A} \left[(Q_\pi(S, A) - b) \cdot \nabla_{\theta^i} \ln \pi(A^i | S; \theta^i) \right].$$

期望中的动作 A 的概率质量函数为

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$



把基线设置为状态价值： $b = V_\pi(s)$ 。定义

$$g^i(s, a; \theta^i) \triangleq (Q_\pi(s, a) - V_\pi(s)) \cdot \nabla_{\theta^i} \ln \pi(a^i | s; \theta^i).$$

定理 15.1 说明 $g^i(s, a^i; \theta^i)$ 是策略梯度的无偏估计，即

$$\nabla_{\theta^i} J(\theta^1, \dots, \theta^m) = \mathbb{E}_{S, A} [g^i(S, A; \theta^i)].$$

因此 $g^i(s, a; \theta^i)$ 可以作为策略梯度的近似。但是我们不知道公式中的 Q_π 、 V_π ，还需要进一步做近似。根据第 8.3 节 A2C 的推导，我们把 $Q_\pi(s_t, a_t)$ 近似成 $r_t + \gamma \cdot v(s_{t+1}; w)$ ，把 $V_\pi(s_t)$ 近似成 $v(s_t; w)$ 。那么近似策略梯度 $g^i(s_t, a_t; \theta^i)$ 可以进一步近似成：

$$\tilde{g}^i(s_t, a_t^i; \theta^i) \triangleq \underbrace{(r_t + \gamma \cdot v(s_{t+1}; w) - v(s_t; w))}_{\text{对 } Q_\pi(s_t, a_t) - V_\pi(s_t) \text{ 的近似}} \cdot \nabla_{\theta^i} \ln \pi(a_t^i | s_t; \theta^i).$$

观测到状态 s_t 、 s_{t+1} 、动作 a_t^i 、奖励 r_t ，这样更新策略网络参数：

$$\boldsymbol{\theta}^i \leftarrow \boldsymbol{\theta}^i + \beta \cdot \tilde{g}^i(s_t, a_t^i; \boldsymbol{\theta}^i).$$

根据 TD 误差 δ_t 的定义，不难看出 $\tilde{g}^i(s_t, a_t^i; \boldsymbol{\theta}^i) = -\delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}^i)$ 。因此，上面更新策略网络参数的公式可以写作：

$$\boldsymbol{\theta}^i \leftarrow \boldsymbol{\theta}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}^i).$$

15.2.2 训练和决策

训练： 实际实现的时候，应当使用目标网络缓解自举造成的偏差。目标网络记作 $v(s; \mathbf{w}^-)$ ，它的结构与 v 相同，但是参数不同。设当前价值网络和目标网络的参数分别是 \mathbf{w}_{now} 和 $\mathbf{w}_{\text{now}}^-$ 。设当前 m 个策略网络的参数分别是 $\boldsymbol{\theta}_{\text{now}}^1, \dots, \boldsymbol{\theta}_{\text{now}}^m$ 。MAC-A2C 重复下面的步骤更新参数：

1. 观测到当前状态 $s_t = [o_t^1, \dots, o_t^m]$ ，让每一个智能体独立做随机抽样：

$$a_t^i \sim \pi(\cdot | s_t; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m,$$

并执行选中的动作。

2. 从环境中观测到奖励 r_t 与下一时刻状态 $s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m]$ 。
3. 让价值网络做预测： $\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}})$ 。
4. 让目标网络做预测： $\hat{v}_{t+1}^- = v(s_{t+1}; \mathbf{w}_{\text{now}}^-)$ 。
5. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^- = r_t + \gamma \cdot \hat{v}_{t+1}^-, \quad \delta_t = \hat{v}_t - \hat{y}_t^-.$$

6. 更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

7. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

8. 更新策略网络参数：

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | s_t; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

MAC-A2C 属于同策略 (On-policy)，不能使用经验回放。

决策： 在完成训练之后，不再需要价值网络 $v(s; \mathbf{w})$ 。每个智能体可以用它自己的策略网络做决策。在时刻 t 观测到全局状态 $s_t = [o_t^1, \dots, o_t^m]$ ，然后做随机抽样得到动作：

$$a_t^i \sim \pi(\cdot | s_t; \boldsymbol{\theta}^i),$$

并执行动作。注意，智能体并不能独立做决策，因为一个智能体的策略网络需要知道其他所有智能体的观测。

15.2.3 实现中的难点

上述 MAC-A2C 的训练和决策貌似简单，然而实现起来却不容易。在 MARL 的常见设定下，第 i 号智能体只知道 o^i ，而观测不到全局状态：

$$s = [o^1, \dots, o^m].$$

这会给决策和训练造成如下的困难：

- 每个智能体有自己的策略网络 $\pi(a^i|s; \theta^i)$ ，可以依靠它做决策。但是它的决策需要全局状态 s 。
- 在训练的过程中，价值网络 $v(s; w)$ 需要知道全局状态 s 才能计算 TD 误差 δ 与梯度 $\nabla_w v(s; w)$ 。
- 在训练的过程中，每个策略网络都需要知道全局状态 s 来计算梯度 $\nabla_{\theta^i} \ln \pi(a^i|s; \theta^i)$ 。

综上所述，如果智能体之间不交换信息，那么智能体既无法做训练，也无法做决策。想要做训练和决策，有两种可行的途径：

- 一种办法是让智能体共享观测。这需要做通信，每个智能体把自己的 o^i 传输给其他智能体。这样每个智能体都有全局的状态 $s = [o^1, \dots, o^m]$ 。
- 另一种办法是对策略网络和价值函数做近似。通常使用 $\pi(a^i|o^i; \theta^i)$ 替代 $\pi(a^i|s; \theta^i)$ 。甚至可以进一步用 $v(o^i; w^i)$ 代替 $v(s; w)$ 。

共享观测的缺点在于通信会让训练和决策的速度变慢。而做近似的缺点在于不完全信息造成训练不收敛、做出错误决策。我们不得不在两种办法之间做出取舍，承受其造成的不良影响。

下一节介绍中心化 (Centralized) 与去中心化 (Decentralized) 的实现方法。中心化让智能体共享信息；优点是训练和决策的效果好，缺点是需要通信，造成延时，影响速度。去中心化需要做近似，避免通信；其优点在于速度快，而缺点则是影响训练和决策的质量。

15.3 三种架构

本节介绍 MAC-A2C 的三种实现方法。第 15.3.1 节介绍 “**中心化训练 + 中心化决策**” (Centralized Training with Centralized Execution)，它是对 MAC-A2C 的忠实实现，训练和决策都需要通信。第 15.3.2 节介绍 “**去中心化训练 + 去中心化决策**” (Decentralized Training with Decentralized Execution)，它对策略网络和价值网络都做近似，以避免训练和决策的通信。第 15.3.3 节介绍 “**中心化训练 + 去中心化决策**” (Centralized Training with Decentralized Execution)，它只近似策略网络以避免决策的通信，它的训练需要通信。

图 15.4 对比了三种架构的策略网络和价值网络。用“完全中心化”作出的决策最好，但是速度最慢，在很多问题中不适用。“中心化训练 + 去中心化决策”虽然在训练中需要通信，但是决策的时候不需要通信，可以做到实时决策。“中心化训练 + 中心化决策”是三种架构中最实用的。

	价值网络	策略网络	训练	决策
中心化训练 + 中心化决策	$v(s; \mathbf{w})$	$\pi(a^i s; \boldsymbol{\theta}^i)$	需要通信	需要通信
去中心化训练 + 去中心化决策	$v(o^i; \mathbf{w}^i)$	$\pi(a^i o^i; \boldsymbol{\theta}^i)$	无需通信	无需通信
中心化训练 + 去中心化决策	$v(s; \mathbf{w})$	$\pi(a^i o^i; \boldsymbol{\theta}^i)$	需要通信	无需通信

图 15.4：三种架构的对比。

15.3.1 中心化训练 + 中心化决策

本节用完全中心化 (Fully Centralized) 的方式实现 MAC-A2C，没有做任何近似。这种实现的缺点在于通信造成延时，使得训练和决策速度变慢。图 15.5 描述了系统的架构。最上面是中央控制器 (Central Controller)，里面部署了价值网络 $v(s; \mathbf{w})$ 与所有 m 个策略网络

$$\pi(a^1 | \boldsymbol{\theta}^1), \quad \pi(a^2 | \boldsymbol{\theta}^2), \quad \dots, \quad \pi(a^m | \boldsymbol{\theta}^m).$$

训练和决策全部由中央控制器完成。智能体负责与环境交互，执行中央控制器的决策 a^i ，并把观测到的 o^i 汇报给中央控制器。如果智能体观测到奖励 r^i ，也发给中央控制器。

中心化训练： 在时刻 t 和 $t+1$ ，中央控制器收集到所有智能体的观测值

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

在“完全合作关系”的设定下，所有智能体有相同的奖励：

$$r_t^1 = r_t^2 = \dots = r_t^m \triangleq r_t.$$

r_t 可以是中央控制器直接从环境中观测到的，也可能是所有智能体本地的奖励 \tilde{r}_t^i 的加和：

$$r_t = \tilde{r}_t^1 + \tilde{r}_t^2 + \dots + \tilde{r}_t^m.$$

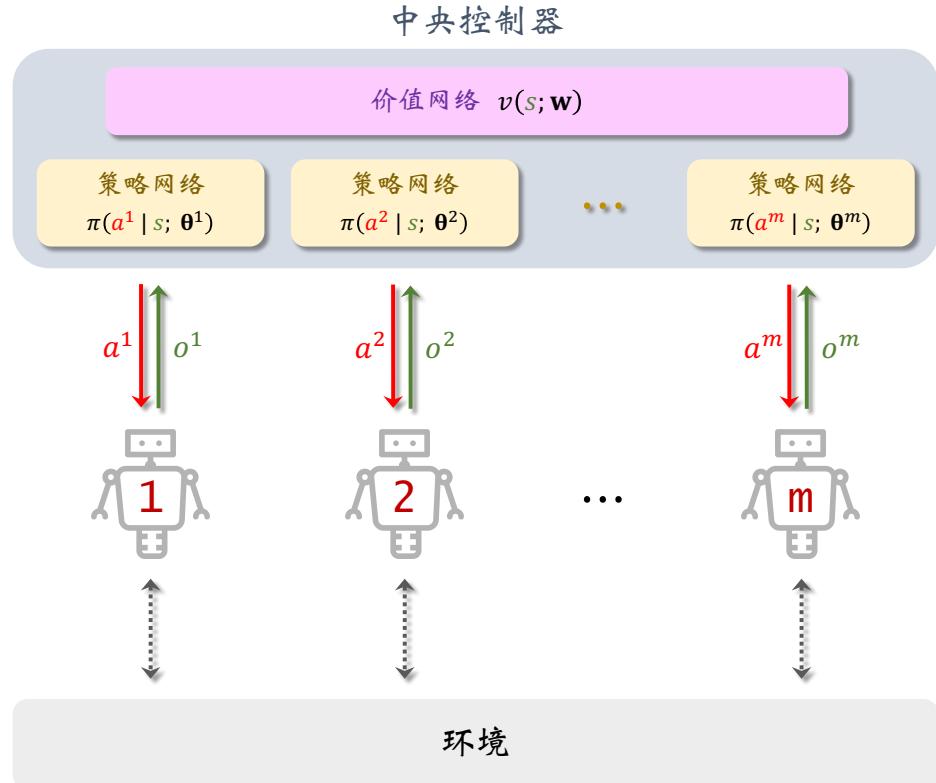


图 15.5: 中心化训练 + 中心化决策的系统架构。

决策是中央控制器上的策略网络做出的，中央控制器因此知道所有的动作：

$$a_t = [a_t^1, \dots, a_t^m].$$

综上所述，中央控制器知道如下信息：

$$s_t, s_{t+1}, a_t, r_t.$$

因此，中央控制器有足够的信息按照第 15.2.2 小节中的算法训练 MAC-A2C，更新价值网络的参数 w 和策略网络的参数 $\theta^1, \dots, \theta^m$ 。

中心化决策：在 t 时刻，中央控制器收集到所有智能体的观测值 $s_t = [o_t^1, \dots, o_t^m]$ ，然后用中央控制器上部署的策略网络做决策：

$$a_t^i \sim \pi(\cdot | s_t; \theta^i), \quad \forall i = 1, \dots, m.$$

中央控制器把决策 a_t^i 传达给第 i 号智能体，该智能体执行 a_t^i 。综上所述，智能体只需要执行中央下达的决策，而不需要自己“思考”。其原因在于策略函数 π 需要全局的状态 s_t 作为输入，而单个智能体不知道全局状态，没有能力单独做决策。

优缺点：中心化训练 + 中心化决策的**优点**在于完全按照 MAC-A2C 的算法实现，没有做任何改动，因此可以确保正确性。基于全局的观测 $s_t = [o_t^1, \dots, o_t^m]$ 做中心化的决策，利用完整的信息，因此作出的决策可以更好。中心化训练和决策的**缺点**在于延迟(Latency)很大，影响训练和决策的速度。在中心化执行的框架下，智能体与中央控制器要做通信。第 i 号智能体要把 o_t^i 传输给中央控制器，而控制器要在收集到所有观测 $[o_t^1, \dots, o_t^m]$ 之

后才会做决策，做出的决策 a_t^i 还得传输给第 i 号智能体。这个过程通常比较慢，使得实时决策不可能做到。机器人、无人车、无人机等应用都需要实时决策，比如在几十毫秒内做出决策；如果出现几百毫秒、甚至几秒的延迟，可能会造成灾难性的后果。

15.3.2 去中心化训练 + 去中心化决策

上一小节的“中心化训练 + 中心化决策”严格按照 MAC-A2C 的算法实现，其缺点在于训练和决策都需要智能体与中央控制器之间通信，造成训练的决策的速度慢。想要避免通信代价，就不得不对策略网络和价值网络做近似。MAC-A2C 中的策略网络

$$\pi(a^1 | s; \theta^1), \quad \pi(a^2 | s; \theta^2), \quad \dots, \quad \pi(a^m | s; \theta^m),$$

和价值网络 $v(s; w)$ 都需要全局的观测 $s = [o^1, \dots, o^m]$ 。“去中心化训练 + 去中心化决策”的基本思想是用局部观测 o^i 代替 s ，把策略网络和价值网络近似成为：

$$\pi(a^i | o^i; \theta^i) \quad \text{和} \quad v(o^i; w^i).$$

在每个智能体上部署一个策略网络和一个价值网络，它们的参数记作 θ^i 和 w^i 。智能体之间不共享参数，即 $\theta^i \neq \theta^j$, $w^i \neq w^j$ 。这样一来，训练就可以在智能体本地完成，无需中央控制器的参与，无需任何通信。见图 15.5 中的系统架构。

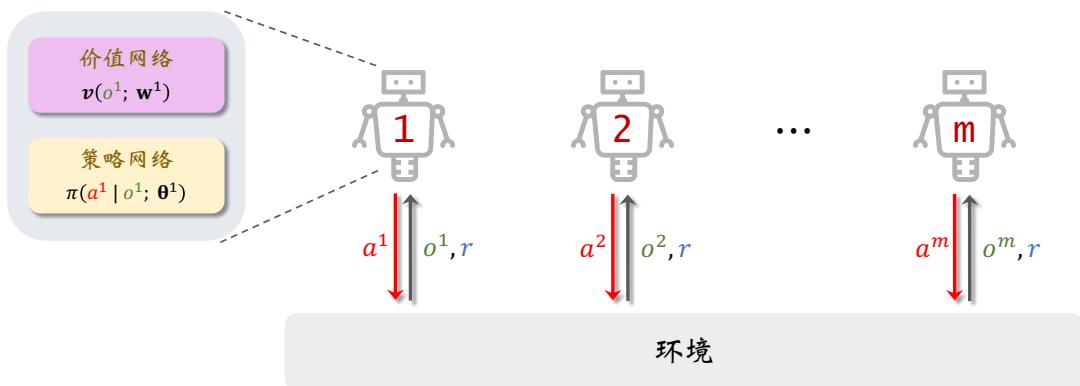


图 15.6: 去中心化训练 + 去中心化决策的系统架构。这种方法也叫做 Independent Actor-Critic。

去中心化训练：假设所有智能体的奖励都是相同的，而且每个智能体都能观测到奖励 r 。每个智能体独立做训练，智能体之间不做通信，不共享观测、动作、参数。这样一来，MAC-A2C 就变成了标准的 A2C，每个智能体独立学习自己的参数 θ^i 与 w^i 。

实际实现的时候，每个智能体还需要一个目标网络，记作 $v(s; w^{i-})$ ，它的结构与 $v(s; w^i)$ 相同，但是参数不同。设第 i 号智能体的策略网络、价值网络、目标网络当前参数分别为 θ_{now}^i 、 w_{now}^i 、 w_{now}^{i-} 。该智能体重复以下步骤更新参数：

1. 在 t 时刻，智能体 i 观测到 o_t^i ，然后做随机抽样 $a_t^i \sim \pi(\cdot | o_t^i; \theta^i)$ ，并执行选中的动作 a_t^i 。
2. 环境反馈给智能体奖励 r_t 与新的观测 o_{t+1}^i 。
3. 让价值网络做预测： $\hat{v}_t^i = v(o_t^i; w_{\text{now}}^i)$ 。
4. 让目标网络做预测： $\hat{v}_{t+1}^i = v(o_{t+1}^i; w_{\text{now}}^{i-})$ 。

5. 计算 TD 目标与 TD 误差:

$$\hat{y}_t^i = r_t + \gamma \cdot \hat{v}_{t+1}^i, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i.$$

6. 更新价值网络参数:

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(o_t^i; \mathbf{w}_{\text{now}}^i).$$

7. 更新目标网络参数:

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}.$$

8. 更新策略网络参数:

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\theta^i} \ln \pi(a_t^i | o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

注 上述算法不是 MAC-A2C，而是单智能体的 A2C。去中心化训练的本质就是单智能体强化学习 (SARL)，而非多智能体强化学习 (MARL)。在 MARL 中，智能体之间会相互影响，而本节中的“去中心化训练”把智能体视为独立个体，忽视它们之间的关联，直接用 SARL 方法独立训练每个智能体。用上述 SARL 的方法解决 MARL 问题，在实践中效果往往不佳。

去中心化决策： 在完成训练之后，智能体 i 不再需要其价值网络 $v(o^i; \mathbf{w}^i)$ 。智能体只需要用其本地部署的策略网络 $\pi(a^i | o^i; \theta^i)$ 做决策即可，决策过程无需通信。去中心化执行的速度很快，可以做到实时决策。

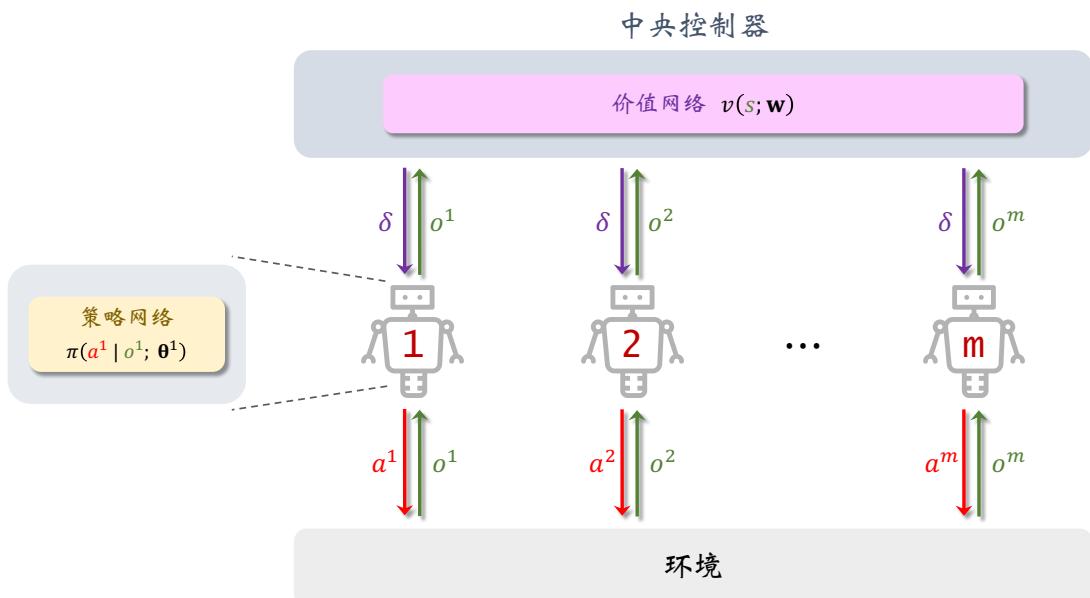


图 15.7: 中心化训练的系统架构。价值网络（以及没画出的目标网络）部署到中央控制器上，策略网络部署到每个智能体上。训练的时候，智能体 i 将观测 \mathbf{o}^i 传输到控制器上，控制器将 TD 误差 δ 传回智能体。

15.3.3 中心化训练 + 去中心化决策

前面两节讨论了完全中心化与完全去中心化，两种实现各有优缺点。当前更流行的 MARL 架构是“中心化训练 + 去中心化决策”。训练的时候使用中央控制器，辅助智能体做训练；见图 15.7。训练结束之后，不再需要中央控制器，每个智能体独立根据本地观测 o^i 做决策；见图 15.8。

本小节与“完全中心化”使用相同的价值网络 $v(s; \mathbf{w})$ 及其目标网络 $v(s; \mathbf{w}^-)$ ；本节与“完全去中心化”使用相同的策略网络：

$$\pi(a^1 | o^1; \boldsymbol{\theta}^1), \dots, \pi(a^m | o^m; \boldsymbol{\theta}^m).$$

第 i 号策略网络的输入是局部观测 o^i ，因此可以将其部署到第 i 号智能体上。价值网络 $v(s; \mathbf{w})$ 的输入是全局状态 $s = [o^1, \dots, o^m]$ ，因此需要将其部署到中央控制器上。

中心化训练：训练的过程需要所有 m 个智能体共同参与，共同改进策略网络参数 $\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m$ 与价值网络参数 \mathbf{w} 。设当前 m 个策略网络的参数为 $\boldsymbol{\theta}_{\text{now}}^1, \dots, \boldsymbol{\theta}_{\text{now}}^m$ 。设当前价值网络和目标网络的参数分别是 \mathbf{w}_{now} 和 $\mathbf{w}_{\text{now}}^-$ 。训练的流程如下：

1. 每个智能体 i 与环境交互，获取当前观测 o_t^i ，独立做随机抽样：

$$a_t^i \sim \pi(\cdot | o_t^i; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m, \quad (15.2)$$

并执行选中的动作。

2. 下一时刻，每个智能体 i 都观测到 o_{t+1}^i 。假设中央控制器可以从环境获取奖励 r_t ，或者向智能体询问奖励 r_t 。
3. 每个智能体 i 向中央控制器传输观测 o_t^i 和 o_{t+1}^i ；中央控制器得到状态

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

4. 中央控制器让价值网络做预测： $\hat{v}_t = v(s_t; \mathbf{w}_{\text{now}})$ 。
5. 中央控制器让目标网络做预测： $\hat{v}_{t+1}^- = v(s_{t+1}; \mathbf{w}_{\text{now}}^-)$ 。
6. 中央控制器计算 TD 目标和 TD 误差：

$$\widehat{y}_t = r_t + \gamma \cdot \widehat{v}_{t+1}^-, \quad \delta_t = \hat{v}_t - \widehat{y}_t,$$

并将 δ_t 广播到所有智能体。

7. 中央控制器更新价值网络参数：

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot \delta_t \cdot \nabla_{\mathbf{w}} v(s_t; \mathbf{w}_{\text{now}}).$$

8. 中央控制器更新目标网络参数：

$$\mathbf{w}_{\text{new}}^- \leftarrow \tau \cdot \mathbf{w}_{\text{new}} + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^-.$$

9. 每个智能体 i 更新策略网络参数：

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | o_t^i; \boldsymbol{\theta}_{\text{now}}^i).$$

注 此处的算法并不等价于第 15.2 节的 MAC-A2C。区别在于此处用 $\pi(a^i | o^i; \boldsymbol{\theta}^i)$ 代替 MAC-A2C 中的 $\pi(a^i | s; \boldsymbol{\theta}^i)$ 。

去中心化决策：在完成训练之后，不再需要价值网络 $v(s; \mathbf{w})$ 。智能体只需要用其

本地部署的策略网络 $\pi(a^i|o^i; \theta^i)$ 做决策，决策过程无需通信。去中心化执行的速度很快，可以做到实时决策。

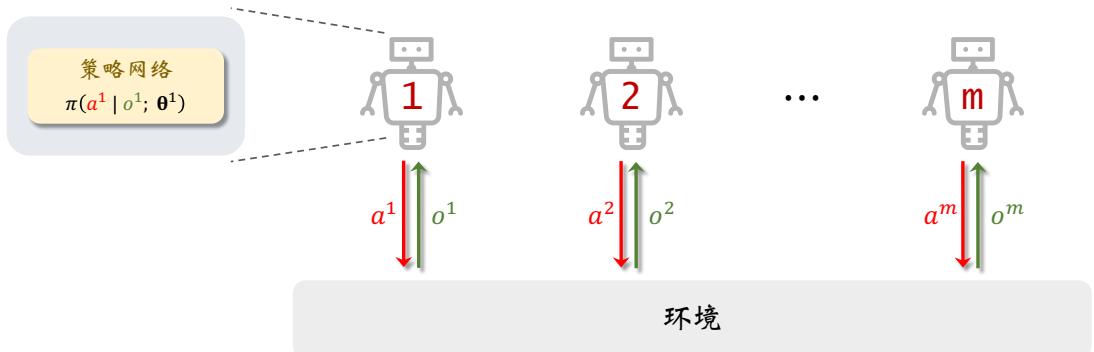


图 15.8: 去中心化决策的系统架构。在完成训练之后，智能体不再做通信，智能体用本地部署的策略网络做决策。

 第十五章 习题 

1. 设动作 $A = [A^1, \dots, A^m]$ 的概率质量函数为

$$\pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) \triangleq \pi(A^1 | S; \boldsymbol{\theta}^1) \times \dots \times \pi(A^m | S; \boldsymbol{\theta}^m).$$

由第 8 章中带基线的策略梯度定理可得：

$$\nabla_{\boldsymbol{\theta}^i} J(\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) = \mathbb{E}_{S,A} \left[\left(Q_\pi(S, A) - b \right) \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) \right].$$

公式中动作 A 的概率质量函数为 $\pi(A | S; \boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m)$, 公式中的 b 是任意不依赖于 A 的函数。请用上面两个公式证明下面的公式：

$$\nabla_{\boldsymbol{\theta}^i} J(\boldsymbol{\theta}^1, \dots, \boldsymbol{\theta}^m) = \mathbb{E}_{S,A} \left[\left(Q_\pi(S, A) - V_\pi(S) \right) \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(A^i | S; \boldsymbol{\theta}^i) \right].$$

∽ 第十五章 相关文献 ∽

完全去中心化的架构早在 1993 年就被提出 [109]，在 2017 年被用在多智能体 DQN 上 [40, 108]。中心化训练 + 去中心化执行 (Centralized Training with Decentralized Execution) 在近年来很流行 [84, 44, 39, 72, 56]。

MAC-A2C 是本书设计出来的简单方法，用于讲解 MARL 的三种架构；MAC-A2C 这个名字并没有出现在任何文献中。MAC-A2C 本质是带基线的 Actor-Critic，其中的基线是状态价值

$$V_\pi(s) \triangleq \mathbb{E}_A [Q_\pi(s, A)],$$

期望是关于动作 $A = [A^1, \dots, A^m]$ 求的。可以把基线换成

$$Q_\pi^{-i}(s, a^{-i}) \triangleq \mathbb{E}_{A^i} [Q_\pi(s, A^i, a^{-i})],$$

公式中 $a^{-i} = [a^1, \dots, a^{i-1}, a^{i+1}, \dots, a^m]$ 。公式中的期望是关于第 i 号智能体的动作 $A^i \sim \pi(\cdot | o^i, \theta^i)$ 求的。用 $Q_\pi^{-i}(s, a^{-i})$ 作为基线，代替 $V_\pi(s)$ ，得到的方法叫做 **C**Ounterfactual **M**ulti-**A**gent，缩写 **COMA** [39]。此外，COMA 还在策略网络中使用 RNN；其原理见第 11 章的解释。COMA 的表现略好于 MAC-A2C，但是 COMA 的实现很复杂，不建议读者自己实现。

第十六章 非合作关系设定下的多智能体强化学习

上一章研究了多智能体强化学习 (MARL) 中最简单的设定——完全合作关系，在这种设定下，所有的智能体有相同的奖励、回报、价值、目标函数。本章研究非合作关系，那么不同智能体各自有不同的奖励、回报、价值、目标函数。本章中采用的符号如图 16.1 所示。

第 16.1 节定义非合作关系设定下的策略学习、策略梯度方法、以及收敛判别。第 16.2 节推导非合作关系下的 A2C 方法，本书称之为 Multi-Agent Noncooperative A2C，缩写 MAN-A2C，可以用于离散控制问题。第 16.3 节用三种架构实现 MAN-A2C：完全去中心化、完全中心化、中心化训练 + 去中心化决策。第 16.4 介绍多智能体确定策略梯度方法，缩写 MADDPG，可以用于连续控制问题。

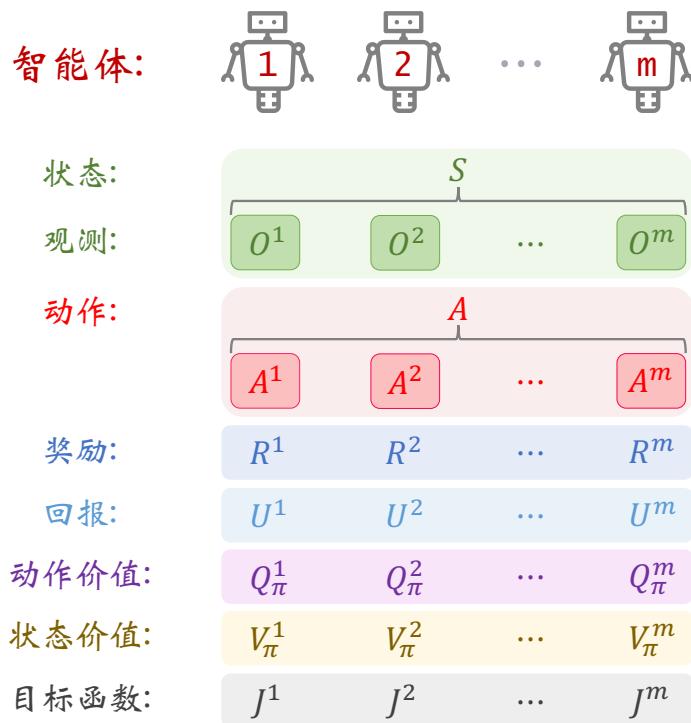


图 16.1: 多智能体强化学习 (MARL) 的符号。

16.1 非合作关系设定下的策略学习

上一章研究合作关系的 MARL，即所有智能体的奖励都相等： $R^1 = \dots = R^m$ 。在这种设定下，所有智能体有相同的状态价值函数 $V_\pi(s)$ 和目标函数

$$J(\theta^1, \dots, \theta^m) = \mathbb{E}_S [V_\pi(s)].$$

目标函数可以衡量策略网络参数 $\theta^1, \dots, \theta^m$ 的好坏。策略学习的目的是改进 $\theta^1, \dots, \theta^m$ 使得 J 变大。合作关系的设定下，策略学习的收敛标准很明确：如果找不到更好的 $\theta^1, \dots, \theta^m$ 使得 J 变大，那么当前的 $\theta^1, \dots, \theta^m$ 就是最优解。

非合作关系设定下的目标函数：如果是非合作关系，那么不存在这样的关系： $R^1 = \dots = R^m$ 。两个智能体的奖励不相等 ($R^i \neq R^j$)，那么它们的回报也不相等 ($U^i \neq U^j$)，回报的期望（价值函数）也不相等。把状态价值记作：

$$V^1(s), V^2(s), \dots, V^m(s).$$

第 i 个智能体的目标函数是状态价值的期望：

$$J^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S [V_\pi^i(s)].$$

J^i 的意义是回报 U^i 的期望，所以能反映出第 i 个智能体的表现好坏。

注 目标函数 J^1, J^2, \dots, J^m 是各不相同的，也就是说智能体没有共同的目标（除非是完全合作关系）。举个例子，在 Predator-Prey（捕食者—猎物）的游戏中，捕食者的目标函数 J^1 与猎物的目标函数 J^2 负相关： $J^1 = -J^2$ 。

注 第 i 个智能体的目标函数 J^i 依赖于所有智能体的策略网络参数 $\theta^1, \dots, \theta^m$ 。为什么一个智能体的目标函数依赖于其他智能体的策略呢？举个例子，捕食者改进自己的策略 θ^1 ，而猎物没有改变策略 θ^2 。虽然猎物的策略 θ^2 没有变化，但是它的目标函数 J^2 会减小。

非合作关系设定下的策略学习：在多智能体的策略学习中，第 i 个智能体的目标是改进自己的策略参数 θ^i ，使得 J^i 尽量大。多智能体的策略学习可以描述为这样的问题：

$$\text{第 1 个智能体求解 : } \max_{\theta^1} J^1(\theta^1, \dots, \theta^m),$$

$$\text{第 2 个智能体求解 : } \max_{\theta^2} J^2(\theta^1, \dots, \theta^m),$$

$$\vdots \quad \vdots$$

$$\text{第 } m \text{ 个智能体求解 : } \max_{\theta^m} J^m(\theta^1, \dots, \theta^m).$$

注意，目标函数 J^1, J^2, \dots, J^m 是各不相同的，也就是说智能体没有共同的目标（除非是完全合作关系）。策略学习的基本思想是让每个智能体各自做策略梯度上升：

$$\text{第 1 号智能体执行 : } \theta^1 \leftarrow \theta^1 + \alpha^1 \cdot \nabla_{\theta^1} J^1(\theta^1, \dots, \theta^m),$$

$$\text{第 2 号智能体执行 : } \theta^2 \leftarrow \theta^2 + \alpha^2 \cdot \nabla_{\theta^2} J^2(\theta^1, \dots, \theta^m),$$

$$\vdots \quad \vdots$$

$$\text{第 } m \text{ 号智能体执行 : } \theta^m \leftarrow \theta^m + \alpha^m \cdot \nabla_{\theta^m} J^m(\theta^1, \dots, \theta^m).$$

公式中的 $\alpha^1, \alpha^2, \dots, \alpha^m$ 是学习率。由于无法直接计算策略梯度 $\nabla_{\theta^i} J^i$ ，我们需要对其做

近似。各种策略学习方法的区别就在于如何对策略梯度做近似。

收敛的判别：在合作关系设定下，所有智能体有相同的目标函数 ($J^1 = \dots = J^m$)，那么判断收敛的标准就是目标函数值不再增长。也就是说改变任何智能体的策略都无法让团队的回报增长。

在非合作关系设定下，智能体的利益是不一致的、甚至是冲突的，智能体各有各的目标函数。该如何判断策略学习的收敛呢？不能用 $J^1 + J^2 + \dots + J^m$ 作为判断收敛的标准。比如在 Predator-Prey（捕食者—猎物）的游戏中，双方的目标函数是冲突的： $J^1 = -J^2$ 。如果捕食者改进策略，那么 J^1 会增长，而 J^2 会下降。自始至终， $J^1 + J^2$ 一直等于零，不论策略学习有没有收敛。

在非合作关系设定下，收敛标准是纳什均衡。一个智能体在制定策略的时候，要考虑到其他各方的策略。在纳什均衡的情况下，每一个智能体都在以最优的方式来应对其他各方的策略。在纳什均衡的情况下，谁也没有动机去单独改变自己的策略，因为改变策略不会增加自己的收益。这样就达到了一种平衡状态，所有智能体都找不到更好的策略。这种平衡状态就被认为是收敛。在实验中，如果所有智能体的平均回报都不再变化，就可以认为达到了纳什均衡。

定义 16.1. 纳什均衡

在多智能体系统中，当其余所有智能体都不改变策略的情况下，一个智能体 i 单独改变策略 θ^i ，无法让其期望回报 $J^i(\theta^1, \dots, \theta^m)$ 变大。



评价策略的优劣：有两种策略学习的方法 \mathcal{M}_+ 和 \mathcal{M}_- ，把它们训练出的策略网络参数分别记作 $\theta_+^1, \dots, \theta_+^m$ 和 $\theta_-^1, \dots, \theta_-^m$ 。该如何评价 \mathcal{M}_+ 和 \mathcal{M}_- 的优劣呢？在合作关系设定下，很容易评价两种方法的好坏。在收敛之后，把两种策略的平均回报记作 J_+ 和 J_- 。如果 $J_+ > J_-$ ，就说明 \mathcal{M}_+ 比 \mathcal{M}_- 好；反之亦然。

在非合作关系的设定下，不能直接用平均回报评价策略的优劣。以捕食者—猎物的游戏为例，我们用两种方法 \mathcal{M}_+ 和 \mathcal{M}_- 训练策略网络，把它们训练出的策略网络记作：

$$\begin{aligned} \pi(a | s, \theta_+^{\text{predator}}), & \quad \pi(a | s, \theta_+^{\text{prey}}), \\ \pi(a | s, \theta_-^{\text{predator}}), & \quad \pi(a | s, \theta_-^{\text{prey}}). \end{aligned}$$

设收敛时的平均回报为：

$$\begin{aligned} J_+^{\text{predator}} &= 0.8, & J_+^{\text{prey}} &= -0.8, \\ J_-^{\text{predator}} &= 0.1, & J_-^{\text{prey}} &= -0.1. \end{aligned}$$

请问 \mathcal{M}_+ 和 \mathcal{M}_- 孰优孰劣呢？假如我们的目标是学习捕食者 (Predator)，能否说明 \mathcal{M}_+ 比 \mathcal{M}_- 好呢？答案是否定的。 $J_+^{\text{predator}} > J_-^{\text{predator}}$ 可能是由于方法 \mathcal{M}_+ 没有训练好猎物 (Pray) 的策略 θ_+^{prey} ，导致捕食者 (Predator) 相对有优势。 $J_+^{\text{predator}} > J_-^{\text{predator}}$ 不能说明策略 $\theta_+^{\text{predator}}$ 优于 $\theta_-^{\text{predator}}$ 。

在非合作关系的设定下，该如何评价两种方法 \mathcal{M}_+ 和 \mathcal{M}_- 的优劣呢？以捕食者—

猎物的游戏为例，我们让一种方法训练出的捕食者与另一种方法训练出的猎物对决：

$$\begin{array}{lll} \pi(a | s, \theta_+^{\text{predator}}) & \text{对决} & \pi(a | s, \theta_-^{\text{prey}}), \\ \pi(a | s, \theta_-^{\text{predator}}) & \text{对决} & \pi(a | s, \theta_+^{\text{prey}}). \end{array}$$

记录下两方捕食者的平均回报，记作 J_+^{predator} 、 J_-^{predator} 。两者的大小可以反映出 \mathcal{M}_+ 和 \mathcal{M}_- 的优劣。

16.2 非合作设定下的多智能体 A2C

本节研究“非合作关系”设定下的多智能体 A2C 方法 (Multi-Agent Non-cooperative A2C)，缩写 MAN-A2C。

16.2.1 策略网络和价值网络

MAN-A2C 中，每个智能体有自己的策略网络和价值网络，记作：

$$\pi(a^i | s; \theta^i) \quad \text{和} \quad v(s; w^i).$$

第 i 个策略网络需要把所有智能体的观测 $s = [o^1, \dots, o^m]$ 作为输入，并输出一个概率分布；第 i 个智能体依据该概率分布抽样得到动作 A^i 。两类神经网络的结构与上一章的 MAC-A2C 完全相同。请注意上一章 MAC-A2C 与本章 MAN-A2C 的区别：

- 上一章的 MAC-A2C 用于完全合作关系，所有智能体有相同的状态价值函数 $V_\pi(s)$ ，所以只用一个神经网络近似 $V_\pi(s)$ ，记作 $v(s; w)$ 。
- 本章的 MAN-A2C 用于非合作关系，每个智能体各有一个状态价值函数 $V_\pi^i(s)$ ，所以每个智能体各自对应一个价值网络 $v(s; w^i)$ 。

MAN-A2C 属于 Actor-Critic 方法：策略网络 $\pi(a^i | s; \theta^i)$ 相当于第 i 个运动员，负责做决策；每个运动员都一个专属的评委 $v(s; w^i)$ ，对运动员 i 的表现予以评价。请注意，虽然评委 $v(s; w^i)$ 是对运动员 i 个人做出评价，但是评委会考虑到全局的状态 $s = [o^1, \dots, o^m]$ 。举个例子，在足球比赛中，评委 i 只对运动员 i 做评价，目的在于改进运动员 i 的技术。在比赛中，想要评价运动员 i 的跑位、传球的好坏，还需要考虑到队友、对手的位置，所以评委 i 会考虑到场上所有球员的表现 $s = [o^1, \dots, o^m]$ 。注意与上一章中 MAC-A2C 的区别：MAC-A2C 中只有一位评委，他会点评整个团队的表现，而不会给每位运动员单独一个评分。

16.2.2 算法推导

在非合作关系设定下，第 i 号智能体的动作价值函数记作 $Q_\pi^i(s, a)$ ，策略网络记作 $\pi(A^i | S; \theta^i)$ 。不难证明下面的策略梯度定理：

定理 16.1. 非合作关系 MARL 的策略梯度定理

设基线 b 为不依赖于 $A = [A^1, \dots, A^m]$ 的函数。那么有

$$\nabla_{\theta^i} J^i(\theta^1, \dots, \theta^m) = \mathbb{E}_{S, A} \left[(Q_\pi^i(S, A) - b) \cdot \nabla_{\theta^i} \ln \pi(A^i | S; \theta^i) \right].$$

期望中的动作 A 的概率质量函数为

$$\pi(A | S; \theta^1, \dots, \theta^m) \triangleq \pi(A^1 | S; \theta^1) \times \dots \times \pi(A^m | S; \theta^m).$$

我们用 $b = V_\pi^i(s)$ 作为定理中的基线，并且用价值网络 $v(s; w^i)$ 近似 $V_\pi^i(s)$ 。按照上

一章的算法推导，我们可以把策略梯度 $\nabla_{\theta^i} J^i(\theta^1, \dots, \theta^m)$ 近似成：

$$\tilde{g}^i(s_t, a_t^i; \theta^i) \triangleq \left(r_t^i + \gamma \cdot v(s_{t+1}; \mathbf{w}^i) - v(s_t; \mathbf{w}^i) \right) \cdot \nabla_{\theta^i} \pi(a_t^i | s_t; \theta^i).$$

观测到状态 s_t 、 s_{t+1} 、动作 a_t^i 、奖励 r_t^i ，这样更新策略网络参数：

$$\theta^i \leftarrow \theta^i + \beta \cdot \tilde{g}^i(s_t, a_t^i; \theta^i).$$

更新价值网络 $v(s; \mathbf{w}^i)$ 的方法与 A2C 基本一样。在观测到状态 s_t 、 s_{t+1} 、奖励 r_t^i 之后，计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot v(s_{t+1}; \mathbf{w}^i).$$

更新参数 \mathbf{w}^i ，使得 $v(s_t; \mathbf{w}^i)$ 更接近 \hat{y}_t^i 。

16.2.3 训练和决策

训练：实现 MAN-A2C 的时候，应当使用目标网络缓解自举造成的偏差。第 i 号智能体的目标网络记作 $v(s; \mathbf{w}^{i-})$ ，它的结构与 $v(s; \mathbf{w}^i)$ 相同，但是参数不同。设第 i 号智能体策略网络、价值网络、目标网络当前的参数是 θ_{now}^i 、 $\mathbf{w}_{\text{now}}^i$ 、 $\mathbf{w}_{\text{now}}^{i-}$ 。MAN-A2C 重复下面的步骤更新参数：

1. 观测到当前状态 $s_t = [o_t^1, \dots, o_t^m]$ ，让每一个智能体独立做随机抽样：

$$a_t^i \sim \pi(\cdot | s_t; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m,$$

并执行选中的动作。

2. 从环境中观测到奖励 r_t^1, \dots, r_t^m 与下一时刻状态 $s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m]$ 。

3. 让价值网络做预测：

$$\hat{v}_t^i = v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

4. 让目标网络做预测：

$$\hat{v}_{t+1}^{i-} = v(s_{t+1}; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

5. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{v}_{t+1}^{i-}, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

6. 更新价值网络参数：

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

7. 更新目标网络参数：

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}, \quad \forall i = 1, \dots, m.$$

8. 更新策略网络参数：

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\theta^i} \ln \pi(a_t^i | s_t; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

MAN-A2C 属于同策略 (On-policy)，不能使用经验回放。

决策：在完成训练之后，不再需要价值网络 $v(s; \mathbf{w}^1), \dots, v(s; \mathbf{w}^m)$ 。每个智能体可以用它自己的策略网络做决策。在时刻 t 观测到全局状态 $s_t = [o_t^1, \dots, o_t^m]$ ，然后做随机抽样得到动作：

$$a_t^i \sim \pi(\cdot | s_t; \theta^i),$$

并执行动作 a_t^i 。智能体并不能独立做决策，因为策略网络需要知道所有的观测 $s_t = [o_t^1, \dots, o_t^m]$ 。

16.3 三种架构

本节介绍 MAN-A2C 的三种实现方法：“中心化训练 + 中心化决策”、“去中心化训练 + 去中心化决策”、“中心化训练 + 去中心化决策”。

16.3.1 中心化训练 + 中心化决策

首先讲解用完全中心化 (Fully Centralized) 的方式实现 MAN-A2C 的训练和决策。这种方式是不实用的，仅帮助大家理解算法而已。图 16.2 描述了系统的架构。最上面是中央控制器 (Central Controller)，里面部署了所有 m 个价值网络和策略网络：

$$\begin{aligned} v(s | \mathbf{w}^1), & \quad v(s | \mathbf{w}^2), \quad \dots, \quad v(s | \mathbf{w}^m), \\ \pi(a^1 | \boldsymbol{\theta}^1), & \quad \pi(a^2 | \boldsymbol{\theta}^2), \quad \dots, \quad \pi(a^m | \boldsymbol{\theta}^m). \end{aligned}$$

训练和决策全部由中央控制器完成。智能体负责与环境交互，执行中央控制器的决策 a^i ，并把观测到的 o^i 和 r^i 汇报给中央控制器。这种中心化的方式严格实现了上一节的算法。

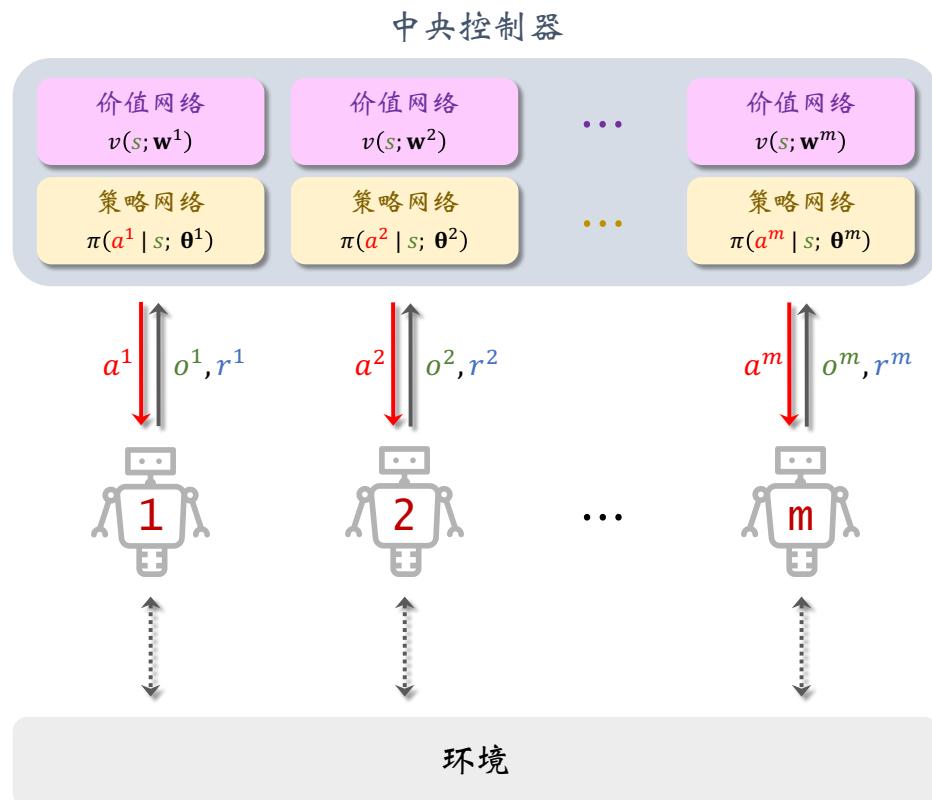


图 16.2：中心化训练 + 中心化决策的系统架构。

在上一章中，我们用完全中心化的方式实现了 MAC-A2C（见图 15.5）。请注意 MAC-A2C 与此处的 MAN-A2C 的区别。第一，MAC-A2C 的中央控制器上只有一个价值网络，而此处 MAN-A2C 则有 m 个价值网络。第二，MAC-A2C 的每一轮只有一个全局的奖励 r ，而 MAN-A2C 的每个智能体都有自己的奖励 r^i 。

16.3.2 去中心化训练 + 去中心化决策

为了避免“完全中心化”中的通信，可以对策略网络和价值网络做近似，做到“完全去中心化”。把 MAN-A2C 中的策略网络和价值网络做近似：

$$\begin{aligned}\pi(a^i | s; \theta^i) &\implies \pi(a^i | o^i; \theta^i), \\ v(s; w^i) &\implies v(o^i; w^i).\end{aligned}$$

图 16.3 描述了“完全去中心化”的系统架构。每个智能体上部署一个策略网络和一个价值网络，它们的参数记作 θ^i 和 w^i ；智能体之间不共享参数。这样一来，训练就可以在智能体本地完成，无需中央控制器的参与，也无需通信。这种实现的本质是单智能体强化学习，而非多智能体强化学习。

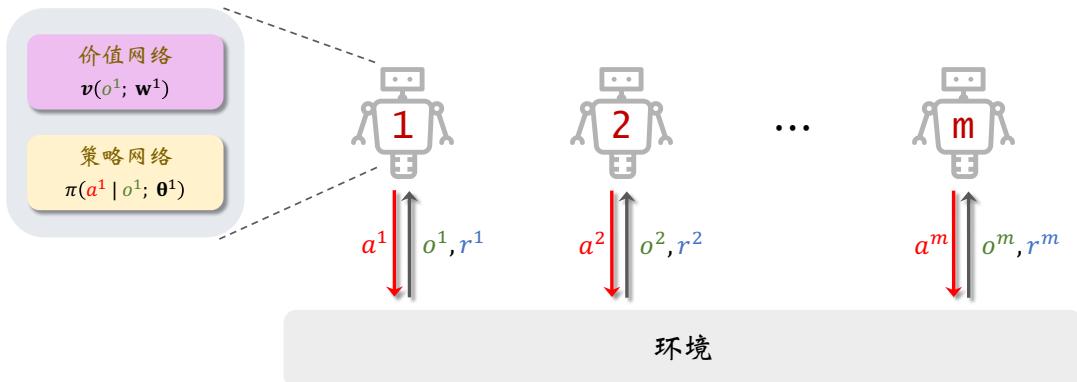


图 16.3：去中心化训练 + 去中心化决策的系统架构。这种方法也叫做 Independent Actor-Critic。

此处的实现与上一章“完全合作关系”设定下的“完全去中心化”几乎完全相同（见图 15.6）。唯一的区别在于此处每个智能体获得的奖励 r^i 是不同的，而上一章“完全合作关系”设定下的奖励是相同的 $r^1 = \dots = r^m = r$.

16.3.3 中心化训练 + 去中心化决策

第三种实现方式是“中心化训练 + 去中心化决策”。与“完全中心化”的 MAN-A2C 相比，唯一的区别在于对策略网络做近似：

$$\pi(a^i | s; \theta^i) \implies \pi(a^i | o^i; \theta^i), \quad \forall i = 1, \dots, m.$$

由于用智能体局部观测 o^i 替换了全局状态 $s = [o^1, \dots, o^m]$ ，策略网络可以部署到每个智能体上。而价值网络仍然是 $v(s; w^i)$ ，没有做近似。

图 16.4 描述了“中心化训练 + 去中心化决策”的系统架构。中央控制器上有所有的价值网络及其目标网络（图中没有画出目标网络）：

$$\begin{aligned}v(s; w^1), \quad v(s; w^2), \quad \dots, \quad v(s; w^m), \\ v(s; w^{1-}), \quad v(s; w^{2-}), \quad \dots, \quad v(s; w^{m-}).\end{aligned}$$

中央控制器用智能体发来的观测 $[o^1, \dots, o^m]$ 和奖励 $[r^1, \dots, r^m]$ 训练这些价值网络。中央控制器把 TD 误差 $\delta^1, \dots, \delta^m$ 反馈给智能体；第 i 号智能体用 δ^i 以及本地的 o^i 、 a^i 来

训练自己的策略网络。

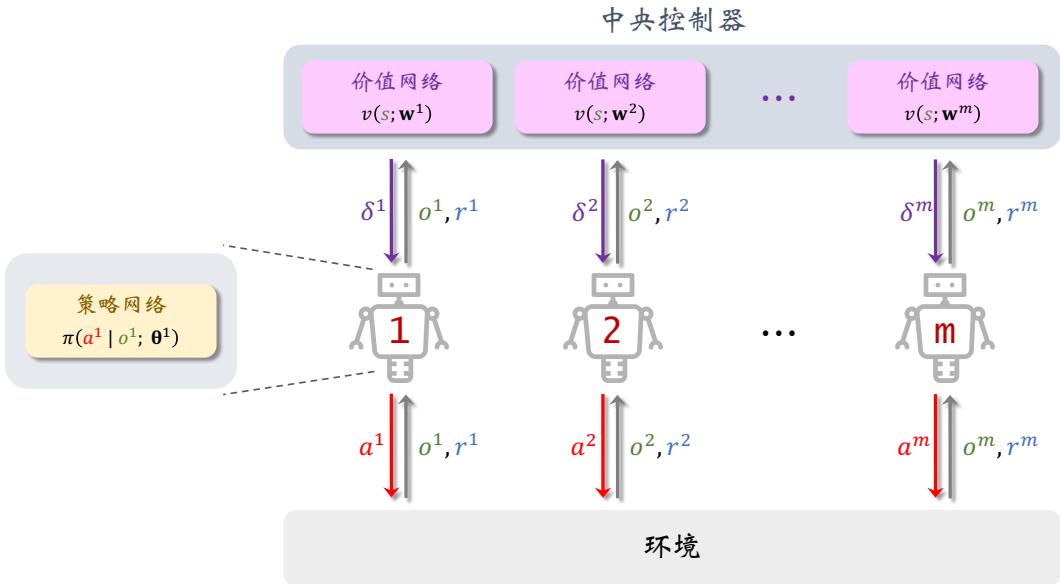


图 16.4: 中心化训练的系统架构。所有 m 个价值网络部署到中央控制器上，策略网络部署到每个智能体上。

上一章“完全合作关系”设定下的“中心化训练”只在中央控制器上部署一个价值网络 $v(s; \mathbf{w})$ 。而此处中央控制器上有 m 个价值网络，每个价值网络对应一个智能体。这是因为此处是“非合作关系”，每个智能体各自对应一个状态价值函数 $V_\pi(s)$ ，而非有共用的 V_π 。

中心化训练：训练的过程需要所有 m 个智能体共同参与，共同改进策略网络参数 $\theta^1, \dots, \theta^m$ 与价值网络参数 $\mathbf{w}^1, \dots, \mathbf{w}^m$ 。设第 i 号智能体的策略网络、价值网络、目标网络当前的参数分别是 θ_{now}^i 、 $\mathbf{w}_{\text{now}}^i$ 和 $\mathbf{w}_{\text{now}}^{i-}$ 。训练的流程如下：

1. 每个智能体 i 与环境交互，获取当前观测 o_t^i ，独立做随机抽样：

$$a_t^i \sim \pi(\cdot | o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m, \quad (16.1)$$

并执行选中的动作。

2. 下一时刻，每个智能体 i 都观测到 o_{t+1}^i 和收到奖励 r_t^i 。
3. 每个智能体 i 向中央控制器传输观测 o_t^i, o_{t+1}^i, r_t^i ；中央控制器得到状态

$$s_t = [o_t^1, \dots, o_t^m] \quad \text{和} \quad s_{t+1} = [o_{t+1}^1, \dots, o_{t+1}^m].$$

4. 让价值网络做预测：

$$\hat{v}_t^i = v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

5. 让目标网络做预测：

$$\hat{v}_{t+1}^{i-} = v(s_{t+1}; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

6. 计算 TD 目标与 TD 误差：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{v}_{t+1}^{i-}, \quad \delta_t^i = \hat{v}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

7. 更新价值网络参数:

$$\mathbf{w}_{\text{new}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} v(s_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

8. 更新目标网络参数:

$$\mathbf{w}_{\text{new}}^{i-} \leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}, \quad \forall i = 1, \dots, m.$$

9. 更新策略网络参数:

$$\boldsymbol{\theta}_{\text{new}}^i \leftarrow \boldsymbol{\theta}_{\text{now}}^i - \beta \cdot \delta_t^i \cdot \nabla_{\boldsymbol{\theta}^i} \ln \pi(a_t^i | o_t^i; \boldsymbol{\theta}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

去中心化决策: 在完成训练之后，不再需要价值网络 $v(s; \mathbf{w}^1), \dots, v(s; \mathbf{w}^m)$ 。智能体只需要用其本地部署的策略网络 $\pi(a^i | o^i; \boldsymbol{\theta}^i)$ 做决策，决策过程无需通信，因此决策速度很快。

16.4 连续控制与 MADDPG

前两节的 MAN-A2C 仅限于离散控制。本节研究连续控制问题，即动作空间 $\mathcal{A}^1, \mathcal{A}^2, \dots, \mathcal{A}^m$ 都是连续集合，动作 $\mathbf{a}^i \in \mathcal{A}^i$ 是向量。本节介绍一种适用于连续控制的多智能体强化学习 (MARL) 方法。多智能体深度确定策略梯度 (Multi-Agent Deep Deterministic Policy Gradient, 缩写 MADDPG) 是一种很有名的 MARL 方法，它的架构是“中心化训练 + 去中心化决策”。

16.4.1 策略网络和价值网络

设系统里有 m 个智能体。每个智能体对应一个策略网络和一个价值网络：

$$\mu(o^i; \theta^i) \quad \text{和} \quad q(s, \mathbf{a}; \mathbf{w}^i).$$

策略网络是确定性的：对于确定的输入 o^i ，输出的动作 $\mathbf{a}^i = \mu(o^i; \theta^i)$ 是确定的。价值网络的输入是全局状态 $s = [o^1, \dots, o^m]$ 与所有智能体的动作 $\mathbf{a} = [\mathbf{a}^1, \dots, \mathbf{a}^m]$ ，输出是一个实数，表示“基于状态 s 执行动作 \mathbf{a} ”的好坏程度。第 i 号策略网络 $\mu(o^i; \theta^i)$ 用于控制第 i 号智能体，而价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ 则用于评价所有动作 \mathbf{a} ，给出的分数可以指导第 i 号策略网络做出改进；见图 16.5。MADDPG 因此可以看做一种 Actor-Critic 方法。

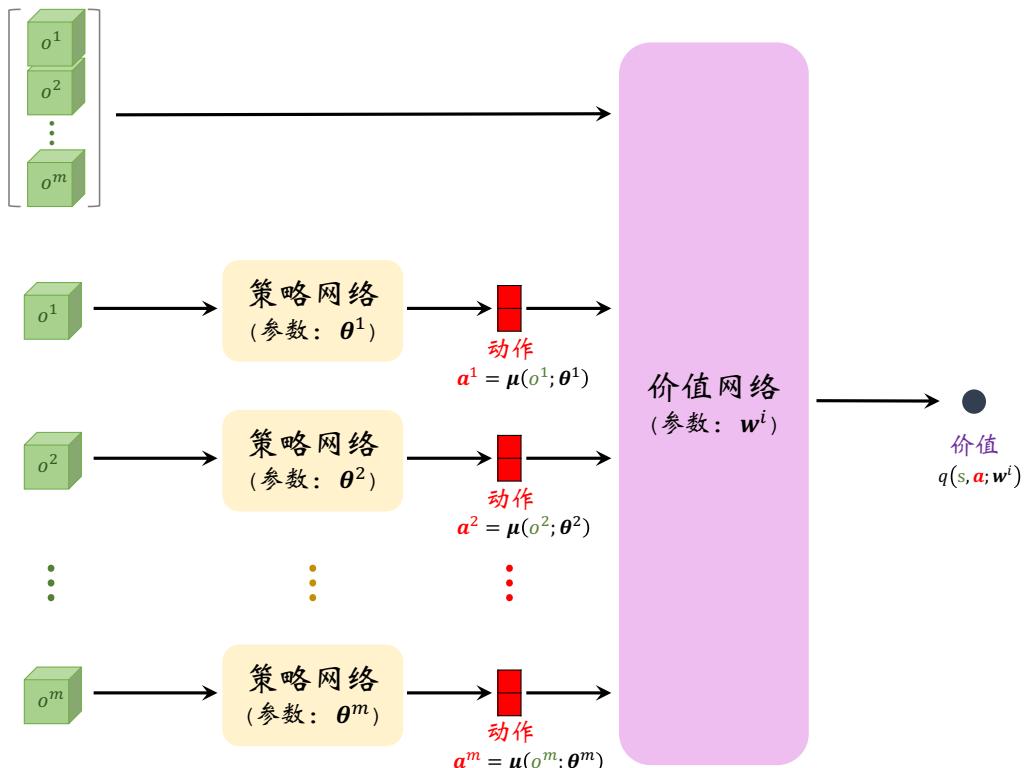


图 16.5：所有智能体的策略网络与第 i 号智能体的价值网络。

16.4.2 算法推导

训练策略网络和价值网络的算法与第 10.2 节的单智能体 DPG 非常类似：用确定策略梯度更新策略网络，用 TD 算法更新价值网络。MADDPG 是异策略(Off-policy)，我们可以使用经验回放，重复利用过去的经验。用一个经验回放数组存储收集到的经验，每一条经验都是 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ 这样一个四元组，其中

$$\begin{aligned} s_t &= [o_t^1, \dots, o_t^m], \\ \mathbf{a}_t &= [\mathbf{a}_t^1, \dots, \mathbf{a}_t^m], \\ s_{t+1} &= [o_{t+1}^1, \dots, o_{t+1}^m], \\ r_t &= [r_t^1, \dots, r_t^m]. \end{aligned}$$

训练策略网络：训练第 i 号策略网络 $\mu(o^i; \theta^i)$ 的目标是改进 θ^i ，增大第 i 号价值网络的平均打分。所以目标函数是：

$$\widehat{J}^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S \left[q(S, [\mu(O^1; \theta^1), \dots, \mu(O^i; \theta^i), \dots, \mu(O^m; \theta^m)]; \mathbf{w}^i) \right].$$

公式中的期望是关于状态 $S = [O^1, \dots, O^m]$ 求的。目标函数的梯度等于：

$$\nabla_{\theta^i} \widehat{J}^i(\theta^1, \dots, \theta^m) = \mathbb{E}_S \left[\nabla_{\theta^i} q(S, [\mu(O^1; \theta^1), \dots, \mu(O^i; \theta^i), \dots, \mu(O^m; \theta^m)]; \mathbf{w}^i) \right].$$

接下来用蒙特卡洛近似公式中的期望。从经验回放数组中随机抽取一个状态：¹

$$s_t = [o_t^1, \dots, o_t^m],$$

它可以看做是随机变量 S 的一个观测值。用所有 m 个策略网络计算动作

$$\widehat{\mathbf{a}}_t^1 = \mu(o_t^1; \theta^1), \dots, \widehat{\mathbf{a}}_t^m = \mu(o_t^m; \theta^m).$$

那么目标函数的梯度 $\nabla_{\theta^i} \widehat{J}^i(\theta^1, \dots, \theta^m)$ 可以近似成为：

$$\begin{aligned} \mathbf{g}_{\theta}^i &= \nabla_{\theta^i} q(s_t, [\mu(o_t^1; \theta^1), \dots, \mu(o_t^i; \theta^i), \dots, \mu(o_t^m; \theta^m)]; \mathbf{w}^i) \\ &= \nabla_{\theta^i} q(s_t, [\widehat{\mathbf{a}}_t^1, \dots, \widehat{\mathbf{a}}_t^m]; \mathbf{w}^i). \end{aligned}$$

由于 $\widehat{\mathbf{a}}_t^i = \mu(o_t^i; \theta^i)$ ，用链式法则可得：

$$\mathbf{g}_{\theta}^i = \nabla_{\theta^i} \mu(o_t^i; \theta^i) \cdot \nabla_{\widehat{\mathbf{a}}^i} q(s_t, [\widehat{\mathbf{a}}_t^1, \dots, \widehat{\mathbf{a}}_t^m]; \mathbf{w}^i).$$

做梯度上升更新参数 θ^i ：

$$\theta^i \leftarrow \theta^i + \beta \cdot \mathbf{g}_{\theta}^i.$$

注意，在更新第 i 号策略网络的时候，除了用到全局状态 s_t ，还需要用到所有智能体的策略网络，以及第 i 号价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ 。

训练价值网络：可以用 TD 算法训练第 i 号价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$ ，让价值网络更好拟

¹更新策略网络只需要四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ 中的 s_t ，没有用其余三个。

合价值函数 $Q_\pi^i(s, \mathbf{a})$ 。给定四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ ，用所有 m 个策略网络计算动作

$$\hat{\mathbf{a}}_{t+1}^1 = \mu(o_{t+1}^1; \theta^1), \dots, \hat{\mathbf{a}}_{t+1}^m = \mu(o_{t+1}^m; \theta^m).$$

设 $\hat{\mathbf{a}}_{t+1} = [\hat{\mathbf{a}}_{t+1}^1, \dots, \hat{\mathbf{a}}_{t+1}^m]$ 。然后计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot q(s_{t+1}, \hat{\mathbf{a}}_{t+1}; \mathbf{w}^i).$$

再计算 TD 误差：

$$\delta_t^i = q(s_t, \mathbf{a}_t; \mathbf{w}^i) - \hat{y}_t^i.$$

最后做梯度下降更新参数 \mathbf{w}^i ：

$$\mathbf{w}^i \leftarrow \mathbf{w}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} q(s_t, \mathbf{a}_t; \mathbf{w}^i).$$

这样可以让价值网络的预测 $q(s_t, \mathbf{a}_t; \mathbf{w}^i)$ 更接近 TD 目标 \hat{y}_t^i 。

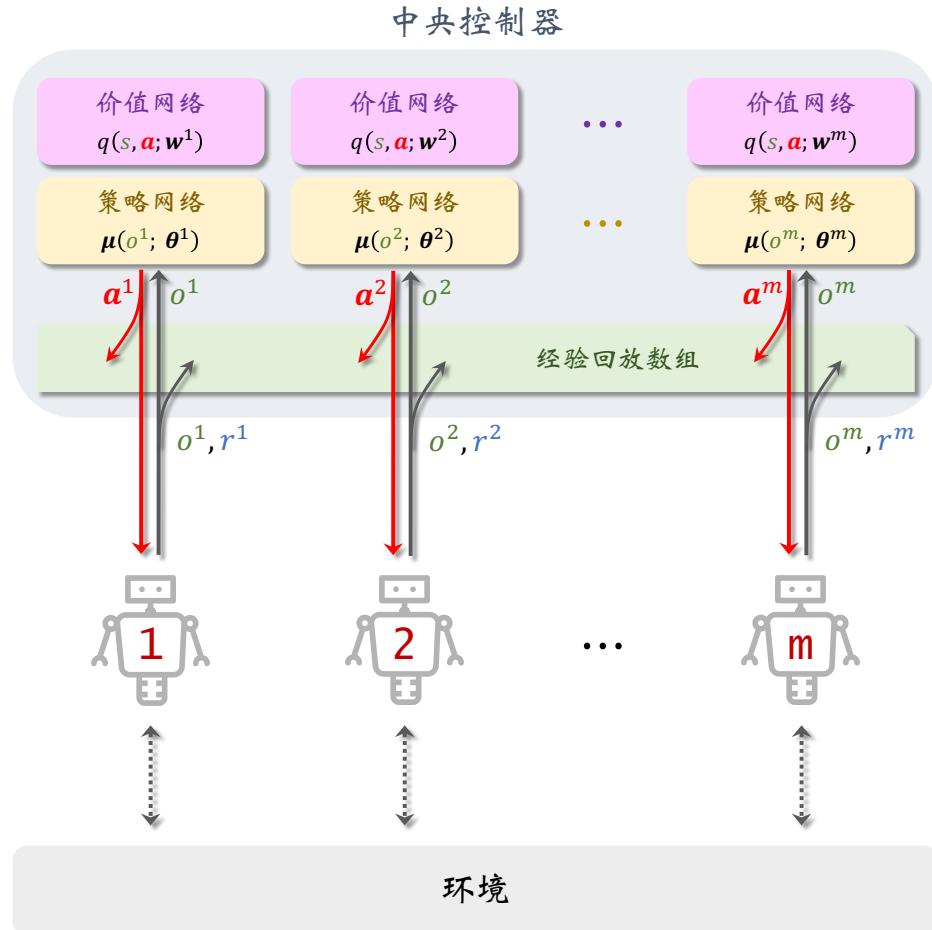


图 16.6: MADDPG 的中心化训练。

16.4.3 中心化训练

为了训练第 i 号策略网络和第 i 号价值网络，我们需要用到如下信息：从经验回放数组中取出的 $s_t, \mathbf{a}_t, s_{t+1}, r_t^i$ 、所有 m 个策略网络、以及第 i 号价值网络。很显然，一

一个智能体不可能有所有这些信息，因此 MADDPG 需要“中心化训练”。

中心化训练的系统架构如图 16.6 所示。有一个中央控制器，上面部署所有的策略网络和价值网络。训练过程中，策略网络部署到中央控制器上，所以智能体不能自主做决策，智能体只是执行中央控制器发来的指令。由于训练使用异策略，可以把收集经验和更新神经网络参数分开做。

用行为策略收集经验。 行为策略 (Behavior Policy) 可以不同于目标策略 (Target Policy)，即 μ 。行为策略是什么都无所谓，比如第 i 个智能体的行为策略可以是

$$\mathbf{a}^i = \mu(o^i; \theta_{\text{old}}^i) + \epsilon,$$

其中 ϵ 是与 \mathbf{a}^i 维度相同的向量，每个元素都是从正态分布中独立抽取的，相当于随机噪声。具体实现的时候，智能体把其观测 o^i 发送给中央控制器。控制器往第 i 号策略网络输出的动作 $\mu(o^i; \theta^i)$ 中加入随机噪声 ϵ ，把动作 \mathbf{a}^i 发送给给第 i 号智能体，智能体执行 \mathbf{a}^i 。随后智能体观测到奖励 r^i ，发送给控制器。控制器把每一轮的 o^i, \mathbf{a}^i, r^i 依次存入经验回放数组。

中央控制器更新策略网络和价值网络： 实际实现的时候，中央控制器上还需要有如下目标网络（图 16.6 中没有画出）：

$$\begin{aligned} \pi(\mathbf{a}^1 | o^1; \theta^{1-}), \quad \pi(\mathbf{a}^2 | o^2; \theta^{2-}), \quad \dots, \quad \pi(\mathbf{a}^m | o^m; \theta^{m-}); \\ q(s, \mathbf{a}; \mathbf{w}^{1-}), \quad q(s, \mathbf{a}; \mathbf{w}^{2-}), \quad \dots, \quad q(s, \mathbf{a}; \mathbf{w}^{m-}). \end{aligned}$$

设第 i 号智能体当前的参数为：

$$\theta_{\text{now}}^i, \quad \theta_{\text{now}}^{i-}, \quad \mathbf{w}_{\text{now}}^i, \quad \mathbf{w}_{\text{now}}^{i-}.$$

中央控制器每次从经验回放数组中随机抽取一个四元组 $(s_t, \mathbf{a}_t, r_t, s_{t+1})$ ，然后按照下面的步骤更新所有策略网络和所有价值网络：

1. 让所有 m 个目标策略网络做预测：

$$\hat{\mathbf{a}}_{t+1}^{i-} = \mu(o_{t+1}^i; \theta_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

把预测汇总成 $\hat{\mathbf{a}}_{t+1}^- = [\hat{\mathbf{a}}_{t+1}^{1-}, \dots, \hat{\mathbf{a}}_{t+1}^{m-}]$ 。

2. 让所有 m 个目标价值网络做出预测：

$$\hat{q}_{t+1}^{i-} = q(s_{t+1}, \hat{\mathbf{a}}_{t+1}^-; \mathbf{w}_{\text{now}}^{i-}), \quad \forall i = 1, \dots, m.$$

3. 计算 TD 目标：

$$\hat{y}_t^i = r_t^i + \gamma \cdot \hat{q}_{t+1}^{i-}, \quad \forall i = 1, \dots, m.$$

4. 让所有 m 个价值网络做预测：

$$\hat{q}_t^i = q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

5. 计算 TD 误差：

$$\delta_t^i = \hat{q}_t^i - \hat{y}_t^i, \quad \forall i = 1, \dots, m.$$

6. 更新所有 m 个价值网络:

$$\mathbf{w}_{\text{now}}^i \leftarrow \mathbf{w}_{\text{now}}^i - \alpha \cdot \delta_t^i \cdot \nabla_{\mathbf{w}^i} q(s_t, \mathbf{a}_t; \mathbf{w}_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

7. 让所有 m 个策略网络做预测:

$$\hat{\mathbf{a}}_t^i = \mu(o_t^i; \theta_{\text{now}}^i), \quad \forall i = 1, \dots, m.$$

把预测汇总成 $\hat{\mathbf{a}}_t = [\hat{\mathbf{a}}_t^1, \dots, \hat{\mathbf{a}}_t^m]$ 。请区别 $\hat{\mathbf{a}}_t$ 与经验回放数组中抽出的 \mathbf{a}_t 。

8. 更新所有 m 个策略网络: $\forall i = 1, \dots, m$,

$$\theta_{\text{new}}^i \leftarrow \theta_{\text{now}}^i - \beta \cdot \nabla_{\theta^i} \mu(o_t^i; \theta_{\text{now}}^i) \cdot \nabla_{\mathbf{a}_t^i} q(s_t, \hat{\mathbf{a}}_t; \mathbf{w}_{\text{now}}^i).$$

9. 更新所有 $2m$ 个目标网络: $\forall i = 1, \dots, m$,

$$\begin{aligned} \theta_{\text{new}}^{i-} &\leftarrow \tau \cdot \theta_{\text{new}}^i + (1 - \tau) \cdot \theta_{\text{now}}^{i-}, \\ \mathbf{w}_{\text{new}}^{i-} &\leftarrow \tau \cdot \mathbf{w}_{\text{new}}^i + (1 - \tau) \cdot \mathbf{w}_{\text{now}}^{i-}. \end{aligned}$$

改进方法: 可以用三种方法改进 MADDPG。第一, 用第 10.4 节中 TD3 的三种技巧改进训练的算法:

- 用截断双 Q 学习 (Clipped Double Q-Learning) 训练价值网络 $q(s, \mathbf{a}; \mathbf{w}^i)$, $\forall i = 1, \dots, m$ 。
- 往训练算法第一步中的 $\hat{\mathbf{a}}_{t+1}^{i-}$ 加入噪声。
- 减小更新策略网络和目标网络的频率, 每更新 $k (> 1)$ 次价值网络, 更新一次策略网络和目标网络。

第二, 按照第 11 章中的方法, 在策略网络和价值网络中使用 RNN, 记忆历史观测。第三, 在价值网络的结构中使用注意力机制, 见下一章。

16.4.4 去中心化决策

在完成训练之后, 不再需要价值网络, 只需要策略网络做决策。如图 16.7 所示, 把策略网络部署到对应的智能体上。第 i 号智能体可以基于本地观测的 o^i , 在本地独立做决策: $\mathbf{a}^i = \mu(o^i; \theta^i)$ 。

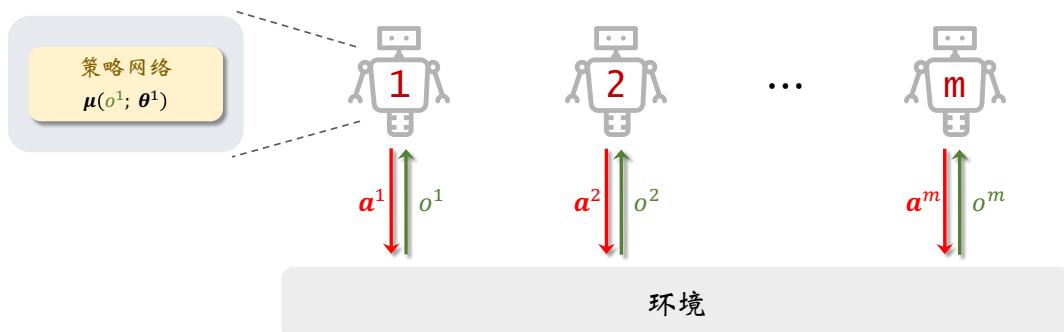


图 16.7: MADDPG 的去中心化决策。

∽ 第十六章 相关文献 ∽

MAN-A2C 是本书设计出来的简单方法，用于讲解非合作设定下的 MARL，方便读者理解。MAN-A2C 这个名字并没有出现在任何文献中。本章介绍的 MADDPG 由 Lowe 等人 2017 年的论文提出 [72]，它的改进版本叫做 MATD3。

第十七章 注意力机制与多智能体强化学习

注意力机制 (Attention) 是一种重要的深度学习方法，它最主要的用途是自然语言处理，比如机器翻译、情感分析。本章的目的不是详细解释注意力机制的原理，而是它在多智能体强化学习 (MARL) 中的应用。第 17.1 简单介绍自注意力机制 (Self-Attention)，它是一种特殊的注意力机制。第 17.2 将自注意力机制应用在 MARL，改进中心化训练或中心化决策。当智能体数量 m 较大时，自注意力机制对 MARL 有明显的效果提升。

17.1 自注意力机制

注意力机制 (Attention) 最初用于改进循环神经网络 (RNN)，提高 Sequence-to-Sequence (Seq2Seq) 模型的表现。自注意力机制 (Self-Attention) 是注意力机制的一种扩展，不局限于 Seq2Seq 模型，可以用于任意的 RNN。后来 Transformer 模型将 RNN 剥离，只保留注意力机制。与 RNN + 注意力机制相比，只用注意力机制居然表现更好，在机器翻译等任务上的效果有大幅提升。本节不深入讨论注意力机制与 RNN、Seq2Seq 之间的关系，而只介绍本章所需的一些知识点。

考虑这样一个问题：输入是长度为 m 的序列 $(\mathbf{x}^1, \dots, \mathbf{x}^m)$ ，序列中的元素都是向量，要求输出长度同样为 m 的序列 $(\mathbf{c}^1, \dots, \mathbf{c}^m)$ ；如图 17.1 所示。问题还有两个要求：

- 第一，序列的长度 m 是不确定的，可以动态变化。但是神经网络的参数数量不能变化。
- 第二，输出的向量 \mathbf{c}^i 不是仅仅依赖于向量 \mathbf{x}^i ，而是依赖于所有的输入向量 $(\mathbf{x}^1, \dots, \mathbf{x}^m)$ 。

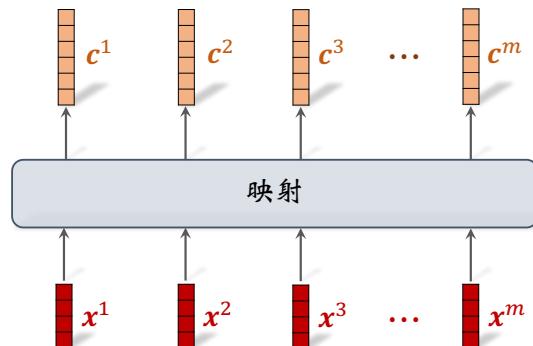


图 17.1：将一个长度为 m 的向量序列映射到另一个同等长度的向量序列。

可以用简单的全连接网络逐个把向量 \mathbf{x}^i 映射到 \mathbf{c}^i ，但是这样得到的 \mathbf{c}^i 仅依赖于 \mathbf{x}^i 一个向量而已，不满足第二个要求。第 13 章介绍的 RNN 也不满足第二个要求；RNN 输出的向量 \mathbf{c}^i 只依赖于 $(\mathbf{x}^1, \dots, \mathbf{x}^i)$ ，而不依赖于 $(\mathbf{x}^{i+1}, \dots, \mathbf{x}^m)$ 。

自注意力层 (Self-Attention Layer) 可以解决上述问题。如图 17.2 所示，自注意力层的输入是序列 $(\mathbf{x}^1, \dots, \mathbf{x}^m)$ ，其中的向量的大小都是 $d_{\text{in}} \times 1$ 。自注意力层有三个参数矩阵：

$$\mathbf{W}_q \in \mathbb{R}^{d_q \times d_{\text{in}}}, \quad \mathbf{W}_k \in \mathbb{R}^{d_k \times d_{\text{in}}}, \quad \mathbf{W}_v \in \mathbb{R}^{d_v \times d_{\text{in}}}.$$

序列长度 m 不会影响参数的数量。不论序列有多长，参数矩阵只有 $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$ 。这三个参数矩阵需要从训练数据中学习。自注意力层通过以下步骤，把输入序列 $(\mathbf{x}^1, \dots, \mathbf{x}^m)$ 映射到输出序列 $(\mathbf{c}^1, \dots, \mathbf{c}^m)$ ，输出向量的大小都是 $d_{\text{out}} \times 1$ 。

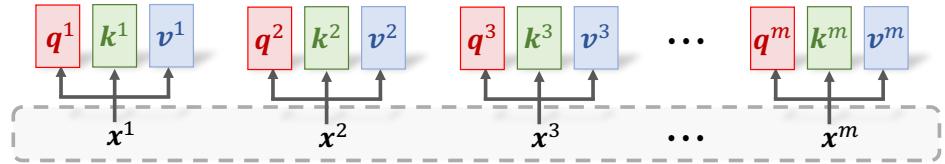


图 17.2: 首先把 x^i 映射到三元组 (q^i, k^i, v^i) , $\forall i = 1, \dots, m$ 。

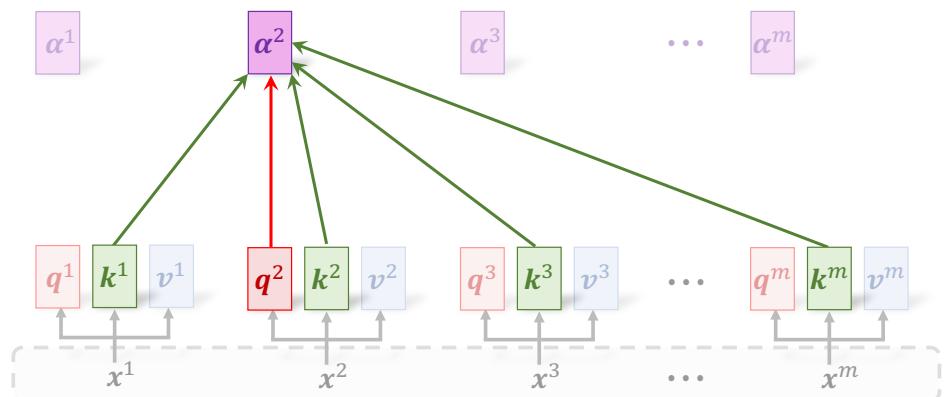


图 17.3: 然后用 q^i 和 (k^1, \dots, k^m) 计算权重向量 $\alpha^i \in \mathbb{R}^m$, $\forall i = 1, \dots, m$ 。

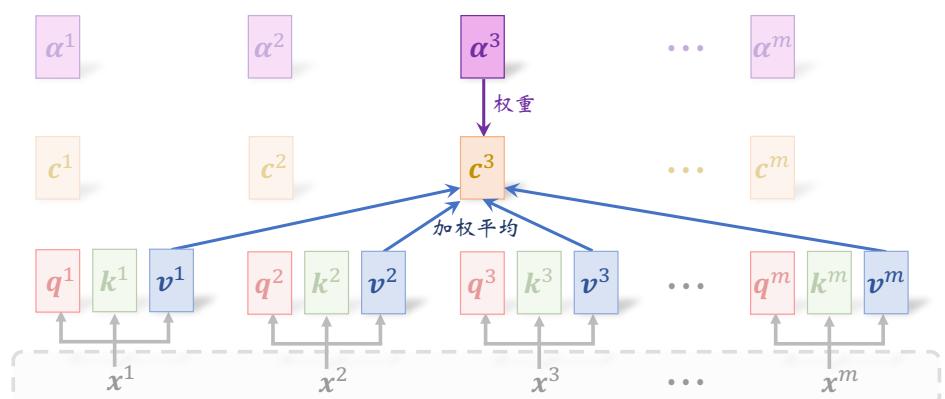


图 17.4: 最后用 α^i 和 (v^1, \dots, v^m) 计算输出向量 $c^i \in \mathbb{R}^{d_{\text{out}}}$, $\forall i = 1, \dots, m$ 。

1. 如图 17.2 所示, 对于所有的 $i = 1, \dots, m$, 把输入的 \mathbf{x}^i 映射到三元组 $(\mathbf{q}^i, \mathbf{k}^i, \mathbf{v}^i)$:

$$\begin{aligned}\mathbf{q}^i &= \mathbf{W}_q \mathbf{x}^i \in \mathbb{R}^{d_q}, \\ \mathbf{k}^i &= \mathbf{W}_k \mathbf{x}^i \in \mathbb{R}^{d_q}, \\ \mathbf{v}^i &= \mathbf{W}_v \mathbf{x}^i \in \mathbb{R}^{d_{\text{out}}}.\end{aligned}$$

2. 如图 17.3 所示, 计算权重向量 $(\boldsymbol{\alpha}^1, \dots, \boldsymbol{\alpha}^m)$, 每个权重向量的大小都是 $m \times 1$ 。第 i 个权重向量 $\boldsymbol{\alpha}^i$ 依赖于 \mathbf{q}^i 和 $(\mathbf{k}^1, \dots, \mathbf{k}^m)$:

$$\boldsymbol{\alpha}^i = \text{softmax} \left(\langle \mathbf{q}^i, \mathbf{k}^1 \rangle, \langle \mathbf{q}^i, \mathbf{k}^2 \rangle, \dots, \langle \mathbf{q}^i, \mathbf{k}^m \rangle \right), \quad \forall i = 1, \dots, m.$$

公式中的 $\langle \cdot, \cdot \rangle$ 是向量内积。由于向量 $\boldsymbol{\alpha}^i$ 是 Softmax 函数的输出, 它的元素都是正实数, 而且相加等于 1。向量 $\boldsymbol{\alpha}^i$ 的第 j 个元素 (记作 α_j^i) 表示 \mathbf{x}_i 与 \mathbf{x}_j 的相关性; \mathbf{x}_i 与 \mathbf{x}_j 越相关, 那么元素 α_j^i 就越大。

3. 如图 17.4 所示, 计算输出向量 $(\mathbf{c}^1, \dots, \mathbf{c}^m)$, 每个输出向量的维度都是 d_{out} 。第 i 个输出向量 \mathbf{c}^i 依赖于 $\boldsymbol{\alpha}^i$ 和 $(\mathbf{v}^1, \dots, \mathbf{v}^m)$:

$$\mathbf{c}^i = [\mathbf{v}^1, \mathbf{v}^2, \dots, \mathbf{v}^m] \cdot \boldsymbol{\alpha}^i = \sum_{j=1}^m \alpha_j^i \mathbf{v}^j, \quad \forall i = 1, \dots, m.$$

\mathbf{c}^i 是向量 $\mathbf{v}^1, \dots, \mathbf{v}^m$ 的加权平均, 权重是 $\boldsymbol{\alpha}^i = [\alpha_1^i, \dots, \alpha_m^i]$ 。

为什么这种神经网络结构叫做注意力 (Attention) 呢? 如图 17.5 所示, 向量 \mathbf{x}^i 位置上的输出是 \mathbf{c}^i , 它是做加权平均计算出来的:

$$\mathbf{c}^i = \alpha_1^i \mathbf{v}^1 + \alpha_2^i \mathbf{v}^2 + \dots + \alpha_m^i \mathbf{v}^m.$$

权重 $\boldsymbol{\alpha}^i = [\alpha_1^i, \dots, \alpha_m^i]$ 反映出 \mathbf{c}^i 最“关注”哪些输入的 $\mathbf{v}^j = \mathbf{W}_v \mathbf{x}^j$ 。如果权重 α_j^i 大, 说明 \mathbf{x}^j 对 \mathbf{c}^i 的影响较大。 \mathbf{c}^i 应当重点关注对其影响较大的 \mathbf{x}^j 。

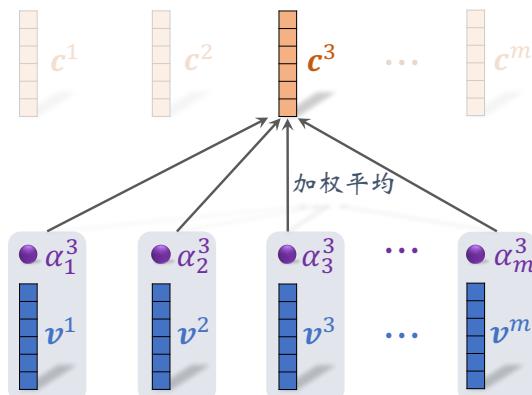


图 17.5: 第 i 个输出向量 \mathbf{c}^i 由权重 $\boldsymbol{\alpha}^i = [\alpha_1^i, \dots, \alpha_m^i]$ 和向量 $(\mathbf{v}^1, \dots, \mathbf{v}^m)$ 决定。

上述自注意力层叫做单头自注意力层 (Single-Head Self-Attention Layer), 简称“单头”。实践中更常用的是多头自注意力层 (Multi-Head Self-Attention Layer), 简称“多头”, 它是多个单头的组合, 见图 17.6。设多头由 l 个单头组成。每个单头有自己的 3 个参数矩阵, 所以多头一共有 $3l$ 个参数矩阵。它们的输入都是序列 $(\mathbf{x}_1, \dots, \mathbf{x}_m)$, 它们的输出都是长度为 m 的向量序列。

第 1 个自注意力层输出: $(\mathbf{c}_1^1, \mathbf{c}_1^2, \mathbf{c}_1^3, \dots, \mathbf{c}_1^m)$,

第 2 个自注意力层输出: $(\mathbf{c}_2^1, \mathbf{c}_2^2, \mathbf{c}_2^3, \dots, \mathbf{c}_2^m)$,

\vdots

第 l 个自注意力层输出: $(\mathbf{c}_l^1, \mathbf{c}_l^2, \mathbf{c}_l^3, \dots, \mathbf{c}_l^m)$.

其中每个向量 c_j^i 的大小都是 $d_{\text{out}} \times 1$ 。多头的输出记作序列 (c^1, \dots, c^m) , 其中每个 c^i 都是做连接 (Concatenation) 得到的:

$$c^i = [c_1^i; c_2^i; \dots; c_l^i] \in \mathbb{R}^{l d_{\text{out}}}, \quad \forall i = 1, \dots, m.$$

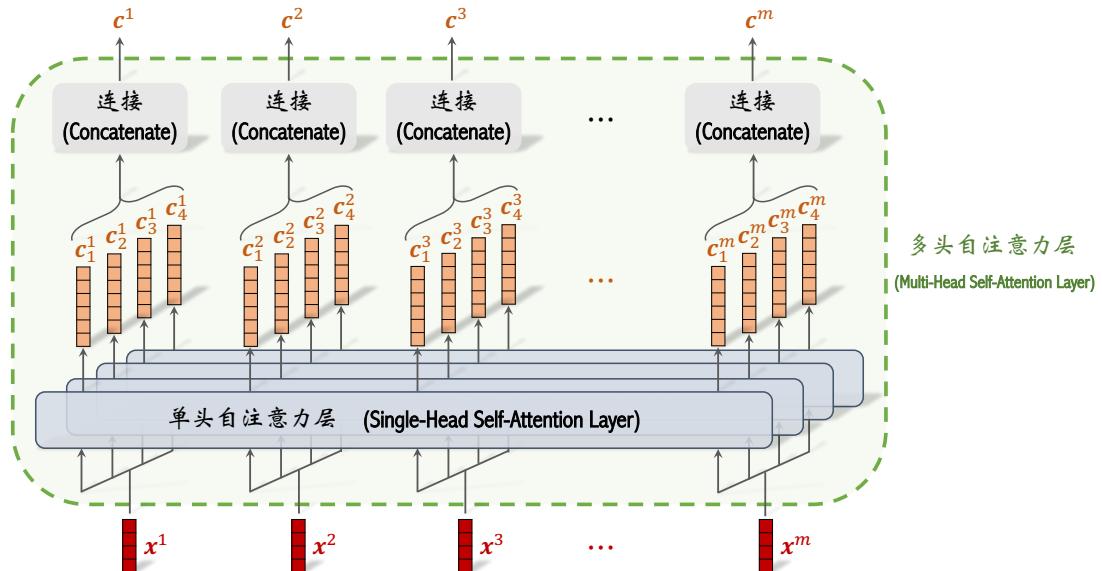


图 17.6: 这个例子中, 多头自注意力层由 $l = 4$ 个单头自注意力层组成。

总结一下, 多头自注意力层把长度为 m 的向量序列映射到同等长度的向量序列。长度 m 可以任意变化, 神经网络结构无需改变。实现一个多头自注意力层需要指定三个超参数: 单头的数量 l 、每个单头输出的大小 d_{out} 、向量 q^i 和 k^i 的大小 d_q 。多头的输出是长度为 m 的向量序列, 每个向量的大小是 $l d_{\text{out}} \times 1$ 。超参数 d_q 不影响输出的大小, 它只在计算权重向量 $\alpha^1, \dots, \alpha^m$ 的时候使用。

17.2 自注意力在中心化训练中的应用

自注意力机制 (Self-Attention) 是改进多智能体强化学习 (MARL) 的一种有效技巧，可以应用在中心化训练或中心化决策当中。多智能体系统中有 m 个智能体，每个智能体有自己的观测（记作 o^1, \dots, o^m ）和动作（记作 a^1, \dots, a^m ）。我们考虑非合作关系的 MARL。如果做中心化训练，需要用到 m 个状态价值网络

$$v([o^1, \dots, o^m]; \mathbf{w}^1), \dots, v([o^1, \dots, o^m]; \mathbf{w}^m),$$

或 m 个动作价值网络

$$q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^1), \dots, q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^m).$$

由于是非合作关系， m 个价值网络有各自的参数，而且它们的输出各不相同。我们首先以状态价值网络 v 为例讲解神经网络的结构。

不使用自注意力的状态价值网络：

图 17.7 是状态价值网络 $v(s; \mathbf{w}^i)$ 最简单的实现。每个价值网络是一个独立的神经网络，有自己的参数。底层提取特征的卷积网络可以在 m 个价值网络中共享（即复用），而上层的全连接网络不能共享。神经网络的输入是所有智能体的观测的连接 (Concatenation)，输出是实数

$$\hat{v}^i = v([o^1, \dots, o^m]; \mathbf{w}^i).$$

这种简单的神经网络结构有几个不足之处。

- 智能体数量 m 越大，神经网络的参数越多。神经网络的输入是 m 个观测的连接，它们被映射到特征向量 \mathbf{x} 。 m 越大，我们就必须把向量 \mathbf{x} 维度设置得越大，否则 \mathbf{x} 无法很好地概括 $[o^1, \dots, o^m]$ 的完整信息。 \mathbf{x} 维度越大，全连接网络的参数就越多，神经网络就越难训练（即需要收集更多的经验才能训练好神经网络）。
- 当 m 很大的时候，并非所有智能体的观测 o^1, \dots, o^m 都与第 i 号智能体密切相关。第 i 号智能体应当学会判断哪些智能体最相关，并重点关注密切相关的智能体，避免决策受无关的智能体干扰。
- 图 17.7 中价值网络的输入是 $[o^1, \dots, o^m]$ ，即所有观测的连接。如果交换其中 o^j 和 o^k 的位置，那么价值网络输出的 \hat{v}^i 会发生变化，这是没有道理的。理想情况下，只要 $j \neq i, k \neq i$ ，那么交换 o^j 和 o^k 的位置就不该改变第 i 号价值网络的输出值 \hat{v}^i 。

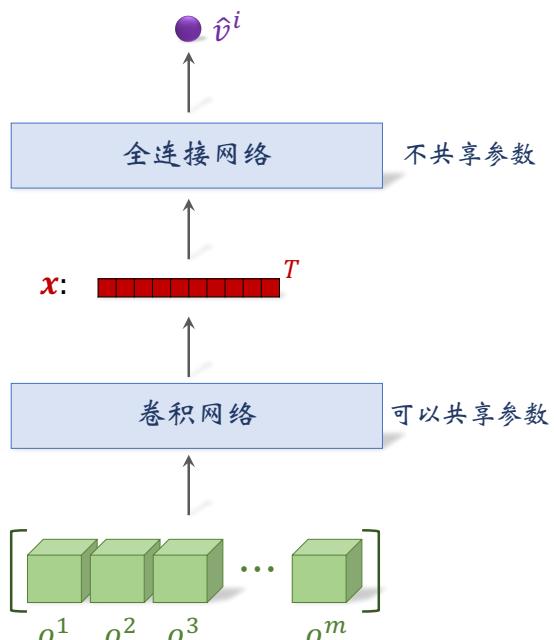


图 17.7：第 i 号状态价值网络最简单的实现。

使用自注意力的状态价值网络：图 17.8 是对状态价值网络更好的实现方式，避免了上面讨论的三种不足之处。神经网络的结构是这样的：

- 输入仍然是所有智能体的观测 o^1, \dots, o^m 。对于所有的 i ，用一个卷积网络把 o^i 映射到特征向量 x^i 。这些卷积网络的参数都是相同的。
- 自注意力层的输入是向量序列 (x^1, \dots, x^m) ，输出是序列 (c^1, \dots, c^m) 。向量 c^i 依赖于所有的观测 x^1, \dots, x^m ，但是 c^i 主要取决于最密切相关的一个或几个 x 。
- 第 i 号全连接网络把向量 c^i 作为输入，输出一个实数 \hat{v}^i ，作为第 i 号价值网络的输出。在非合作关系的设定下， m 个价值网络是不同的，因此 m 个全连接网络不共享参数。

图 17.8 中只用了一个自注意力层。其实可以重复自注意力层，比如：

$\dots \rightarrow$ 自注意力层 \rightarrow 全连接层 \rightarrow 自注意力层 \rightarrow 全连接层 $\rightarrow \dots$

自注意力的层数是一个超参数，需要用户自己调。

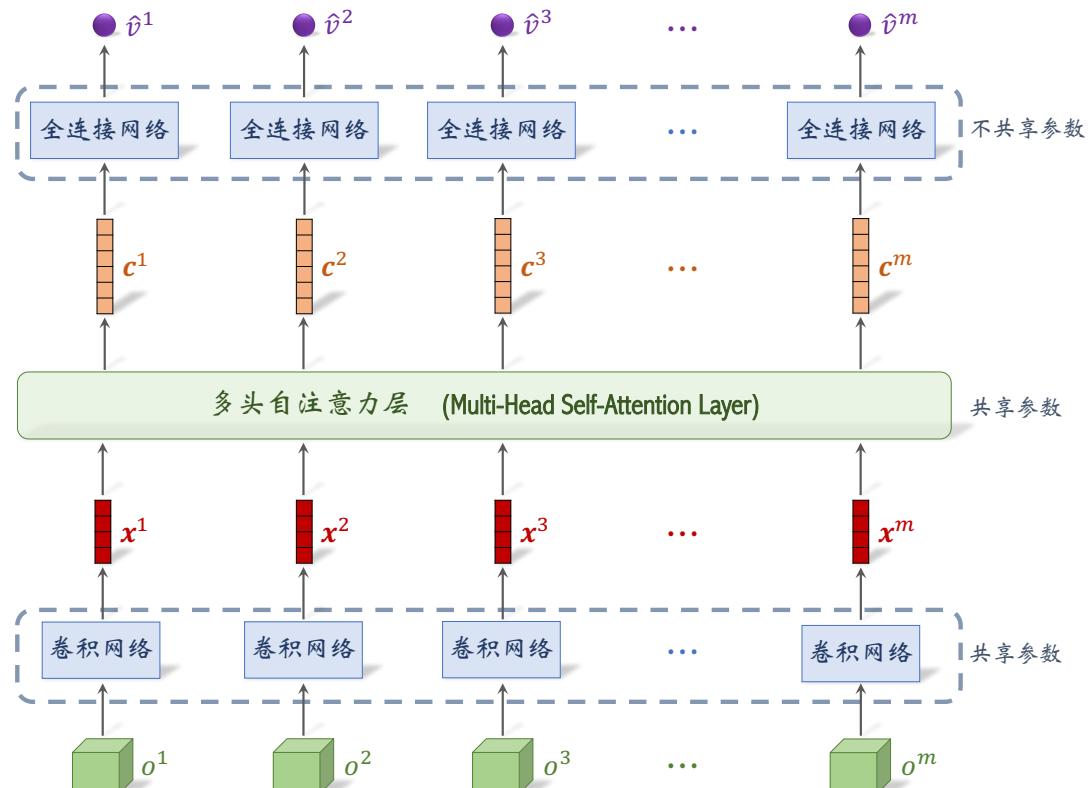


图 17.8：带有自注意力的状态价值网络。图中的 $\hat{v}^i = v([o^1, \dots, o^m]; \mathbf{w}^i)$ 是第 i 个价值网络的输出。

使用自注意力的动作价值网络：上一章介绍了 MADDPG，它是一种连续控制方法，用于非合作关系的设定。它的架构是“中心化训练 + 去中心化决策”，在中央控制器上部署 m 个动作价值网络，把第 i 个记作：

$$\hat{q}^i = q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^i).$$

它的输入是所有智能体的观测和动作，输出是实数 \hat{q}^i ，表示动作价值。可以按照图 17.9

17.2 自注意力在中心化训练中的应用

实现动作价值网络。在 MADDPG 中使用这样的神经网络结构可以提高 MADDPG 的表现，尤其是当 m 较大的时候，效果的提升较大。

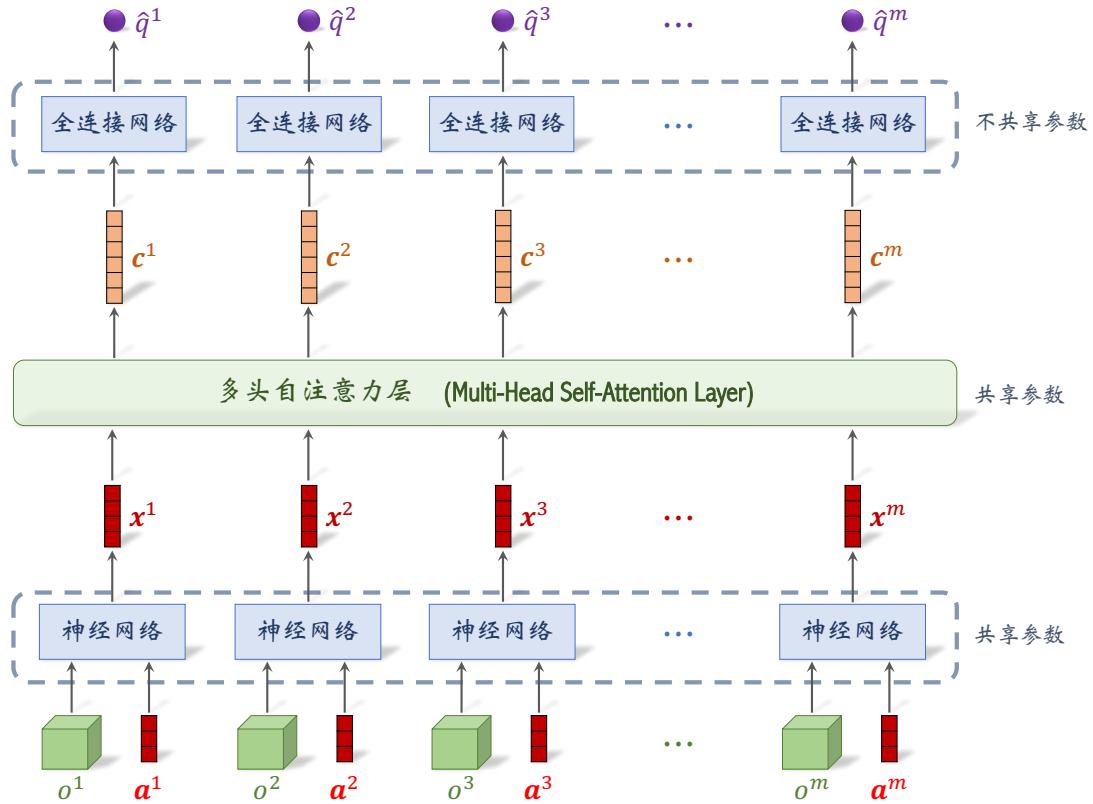


图 17.9: 带有自注意力的动作价值网络。图中的 $\hat{q}^i = q([o^1, \dots, o^m], [a^1, \dots, a^m]; \mathbf{w}^i)$ 是第 i 个动作价值网络的输出。

使用自注意力的中心化策略网络：对于“中心化训练 + 中心化决策”的系统架构，需要在中央控制器上部署 m 个策略网络，每个策略网络都需要知道所有 m 个智能体的观测 o^1, \dots, o^m 。

- 对于离散控制，第 i 号策略网络记作：

$$\hat{\mathbf{f}}^i = \pi(\cdot \mid [o^1, \dots, o^m]; \theta^i).$$

策略网络的输出是向量 $\hat{\mathbf{f}}^i$ ，它的维度是第 i 号动作空间的大小 $|\mathcal{A}^i|$ ， $\hat{\mathbf{f}}^i$ 的元素表示每种动作的概率。根据 $\hat{\mathbf{f}}^i$ 做随机抽样，得到动作 a^i ，第 i 号智能体执行这个动作。

- 对于连续控制，第 i 号策略网络记作：

$$\mathbf{a}^i = \mu([o^1, \dots, o^m]; \theta^i).$$

它的输出是动作 \mathbf{a}^i ，它是 d 维向量， d 是连续控制问题的自由度。第 i 号智能体执行动作 \mathbf{a}^i 。

不管是离散控制还是连续控制，上述两种策略网络中都可以使用自注意力层，神经网络的结构与图 17.8 中的 $v(s; \mathbf{w}^i)$ 几乎一样，唯一区别是神经网络的输出由实数 $\hat{v}^1, \dots, \hat{v}^m$

变成向量 $\hat{\mathbf{f}}^1, \dots, \hat{\mathbf{f}}^m$ 或者 $\mathbf{a}^1, \dots, \mathbf{a}^m$ 。

总结：自注意力机制在**非合作关系**的 MARL 中普遍适用。如果系统架构使用**中心化训练**，那么 m 个**价值网络**可以用一个神经网络实现，其中使用自注意力层。如果系统架构使用**中心化决策**，那么 m 个**策略网络**也可以实现成一个神经网络，其中使用自注意力层。在 m 较大的情况下，使用自注意力层对效果有较大的提升。

∽ 第十七章 相关文献 ∽

注意力机制 (Attention) 由 2015 年的论文 [6] 提出；这篇论文将注意力机制与 RNN 结合，大幅提升 RNN 在机器翻译任务上的表现。2017 年的论文 [119] 提出 Transformer 模型，去掉 RNN，只保留注意力，在机器翻译任务上取得了远优于 RNN 加注意力的表现。2019 年的论文 [56] 将注意力层用到多智能体的 Actor-Critic 中。

第五部分

应用与展望

第十八章 AlphaGo 与蒙特卡洛树搜索

之前章节介绍的强化学习方法都是无模型的强化学习 (Model-Free)，包括价值学习 (Value-Based) 和策略学习 (Policy-Based)。本章介绍的蒙特卡洛树搜索 (Monte Carlo Tree Search，缩写 MCTS) 是一种基于模型的强化学习方法 (Model-Based)。MCTS 比价值学习和策略学习更难理解，所以本章结合 AlphaGo 讲解 MCTS。

AlphaGo 的字面意思是“围棋王”，俗称“阿尔法狗”，它是世界上第一个打败人类围棋冠军的 AI。在 2015 年 10 月，AlphaGo 以 5 : 0 战胜欧洲围棋冠军、职业二段选手樊麾。在 2016 年 3 月，AlphaGo 以 4 : 1 战胜世界冠军李世石。2017 年新版的 AlphaGo Zero 更胜一筹，以 100 : 0 战胜 AlphaGo。

AlphaGo 依靠 MCTS 做决策，而决策的过程中需要策略网络和价值网络的辅助。第 18.1 节用强化学习的语言描述围棋的状态和动作，并且构造策略网络和价值网络。第 18.2 节详细讲解 MCTS 的决策过程。第 18.3 节讲解 AlphaGo 2016 版与 AlphaGo Zero 是如何训练策略网络和价值网络的。

18.1 动作、状态、策略网络、价值网络

围棋的棋盘是 19×19 的网格，可以在两条线交叉的地方放置棋子，一共有 361 个可以放置棋子的位置。两个玩家一方用黑色棋子，另一方用白色棋子，两方交替往棋盘上放置棋子。棋盘上有 361 个可以放置棋子的位置，因此动作空间是 $\mathcal{A} = \{1, \dots, 361\}$ 。比如动作 $a = 123$ 的意思是在第 123 号位置上放棋子。

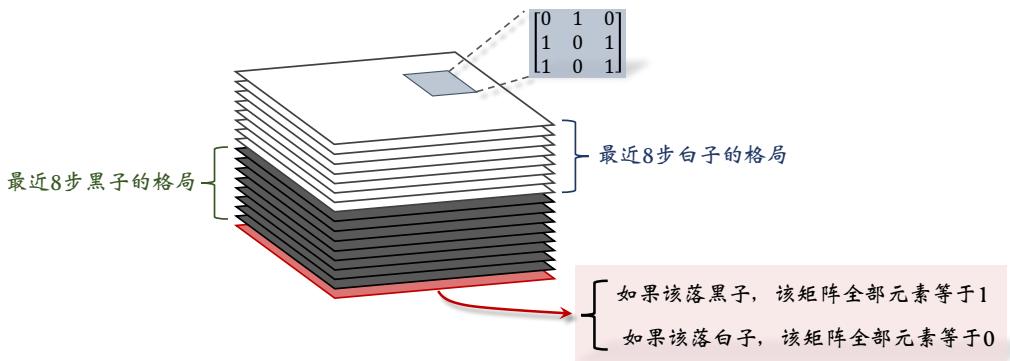


图 18.1：状态可以表示为 $19 \times 19 \times 17$ 的张量。

AlphaGo 2016 版本使用 $19 \times 19 \times 48$ 的张量 (Tensor) 表示一个状态。AlphaGo Zero 使用 $19 \times 19 \times 17$ 的张量表示一个状态。本书只解释后者；见图 18.1。下面解释 $19 \times 19 \times 17$ 的状态张量的意义。

- 张量每个切片 (Slice) 是 19×19 的矩阵，对应 19×19 的棋盘。一个 19×19 的矩阵可以表示棋盘上所有黑子的位置。如果一个位置上有黑子，矩阵对应的元素就是

1，否则就是0。同样的道理，用一个 19×19 的矩阵来表示当前棋盘上所有白子的位置。

- 张量中一共有17个这样的矩阵；17是这样得来的。记录最近8步棋盘上黑子的位置，需要8个矩阵。同理，还需要8个矩阵记录白子的位置。还另外需要一个矩阵表示该哪一方下棋；如果该下黑子，那么该矩阵元素全部等于1；如果该下白子，那么该矩阵的元素全都等于0。

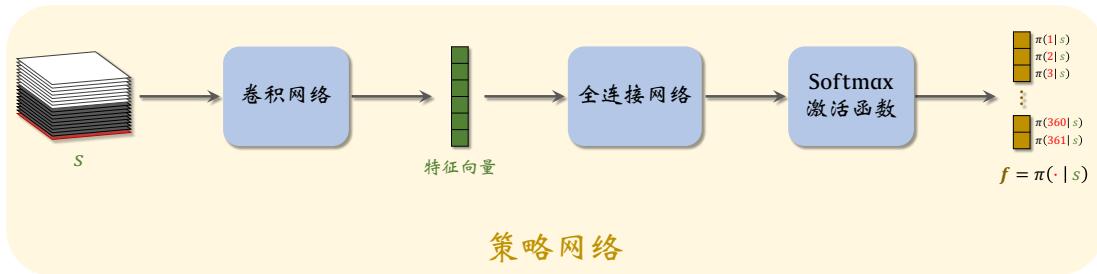


图 18.2: 策略网络的示意图。

策略网络 $\pi(a|s; \theta)$ 的结构如图 18.2 所示。策略网络的输入是 $19 \times 19 \times 17$ 的状态 s 。策略网络的输出是 361 维的向量 f ，它的每个元素对应一个动作（即在棋盘上一个位置放棋子）。向量 f 所有元素都是正数，而且相加等于 1。

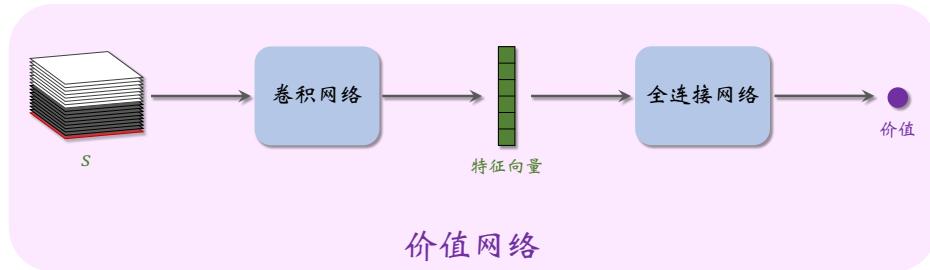


图 18.3: 价值网络的示意图。

AlphaGo 还有一个价值网络 $v(s; w)$ ，它是对状态价值函数 $V_\pi(s)$ 的近似。价值网络的结构如图 18.3 所示。价值网络的输入是 $19 \times 19 \times 17$ 的状态 s 。价值网络的输出是一个实数，它的大小评价当前状态 s 的好坏。

策略网络和价值的输入相同，都是状态 s 。它们都用多个卷积层把 s 映射到特征向量。因此可以让策略网络和价值网络共用卷积层。训练策略网络和价值网络的方法在第 18.3 节解释。

18.2 蒙特卡洛树搜索 (MCTS)

假设此时已经训练好了策略网络 $\pi(a|s; \theta)$ 和价值网络 $v(s; w)$ 。AlphaGo 真正跟人下棋的时候，做决策的不是策略网络或者价值网络，而是蒙特卡洛树搜索 (Monte Carlo Tree Search)，缩写 MCTS。MCTS 不需要训练，可以直接做决策。训练策略网络和价值网络的目的是辅助 MCTS。本节中假设策略网络和价值网络已经训练好，可以直接用；下一节再具体讲解策略网络和价值网络的训练。

18.2.1 MCTS 的基本思想

思考一个问题：人类玩家是怎么下围棋、象棋、五子棋的？人类玩家通常都会向前看几步；越是高手，看得越远。假如现在该我放棋子了，我应该思考这样的问题：当前有几个貌似可行的走法，假如我的动作是 $a_t = 234$ ，对手会怎么走呢？假如接下来对手把棋子放在 $a'_t = 30$ 的位置上，那我下一步的动作 a_{t+1} 应该是什么呢？做当前决策之前，我需要在大脑里做这样的预判，确保几步以后我很可能会占优势。如果我只根据当前格局做判断，不往前看，我肯定赢不了高手。同理，AI 下棋也应该向前看，应该枚举未来可能发生的情况，从而判断当前执行什么动作的胜算最大；这样做远好于用策略网络计算一个动作。

MCTS 的基本原理就是向前看，模拟未来可能发生的情况，从而找出当前最优的动作。AlphaGo 每走一步棋，都要用 MCTS 做成千上万次模拟，从而判断出哪个动作的胜算最大。做模拟的基本思想如下。假设当前有三种看起来很好的动作。每次模拟的时候从三种动作中选出一种，然后将一局游戏进行到底，从而知晓胜负。（只是计算机做模拟而已，不是真的跟对手下完一局。）重复成千上万次模拟，统计一下每种动作的胜负频率，发现三种动作胜率分别是 48%、56%、52%。那么 AlphaGo 应当执行第二种动作，因为它的胜算最大。以上只是 MCTS 的基本想法，实际做起来有很多难点需要解决。

18.2.2 MCTS 的四个步骤

MCTS 的每一次模拟选出一个动作 a ，执行这个动作，然后把一局游戏进行到底，用胜负来评价这个动作的好坏。MCTS 的每一次模拟分为四个步骤：选择 (Selection)、扩展 (Expansion)、求值 (Evaluation)、回溯 (Backup)。

第一步——选择 (Selection): 观测棋盘上当前的格局，找出所有空位，然后判断其中哪些位置符合围棋规则；每个符合规则的位置对应一个可行的动作。每一步至少有几十、甚至上百个可行的动作；假如挨个搜索和评估所有可行动作，计算量会大到无法承受。虽然有几十、上百个可行动作，好在只有少数几个动作有较高的胜算。第一步——选择——的目的就是找出胜算较高的动作，只搜索这些好的动作，忽略掉其他的动作。

如何判断动作 a 的好坏呢？有两个指标：第一，动作 a 的胜率；第二，策略网络给

动作 a 的评分（概率值）。用下面这个分值评价 a 的好坏：

$$\text{score}(a) \triangleq Q(a) + \frac{\eta}{1+N(a)} \cdot \pi(a|s; \theta). \quad (18.1)$$

此处的 η 是个需要调的超参数。公式中 $N(a)$ 、 $Q(a)$ 的定义如下：

- $N(a)$ 是动作 a 已经被访问过的次数。初始的时候，对于所有的 a ，令 $N(a) \leftarrow 0$ 。动作 a 每被选中一次，我们就把 $N(a)$ 加一： $N(a) \leftarrow N(a) + 1$ 。
- $Q(a)$ 是之前 $N(a)$ 次模拟算出来的动作价值，主要由胜率和价值函数决定。 $Q(a)$ 的初始值是 0；动作 a 每被选中一次，就会更新一次 $Q(a)$ ；后面会详解。

可以这样理解公式 (18.1)：

- 如果动作 a 还没被选中过，那么 $Q(a)$ 和 $N(a)$ 都等于零，因此可得

$$\text{score}(a) = \eta \cdot \pi(a|s; \theta),$$

也就是说完全由策略网络评价动作 a 的好坏。

- 如果动作 a 已经被选中过很多次，那么 $N(a)$ 就很大，导致策略网络在 $\text{score}(a)$ 中的权重降低。当 $N(a)$ 很大的时候，有

$$\text{score}(a) \approx Q(a),$$

此时主要基于 $Q(a)$ 判断 a 的好坏，而策略网络已经无关紧要。

- 系数 $\frac{1}{1+N(a)}$ 的另一个作用是鼓励探索，也就是让被选中次数少的动作有更多的机会被选中。假如两个动作有相近的 Q 分数和 π 分数，那么被选中次数少的动作的 score 会更高。

MCTS 根据公式 (18.1) 算出所有动作的分数 $\text{score}(a)$ ， $\forall a$ 。MCTS 选择分数最高的动作。图 18.4 的例子中有 3 个可行动作，分数分别为 0.4、0.3、0.5。第三个动作分数最高，会被选中。这一轮模拟会执行这个动作（只是模拟而已，不是 AlphaGo 真的走一步棋）。

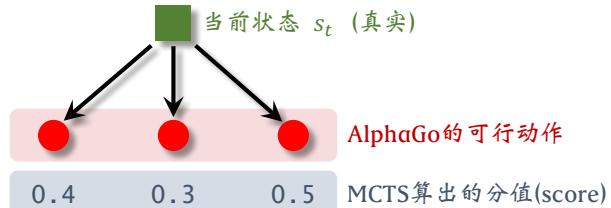


图 18.4：假设有 3 个可行动作，根据公式 (18.1) 算出它们的分数。

第二步——扩展 (Expansion): 把第一步选中的动作记作 a_t ，它只是个假想的动作，只在“模拟器”中执行，而不是 AlphaGo 真正执行的动作。AlphaGo 需要考虑这样一个问题：假如它执行动作 a_t ，那么对手会执行什么动作呢？对手肯定不会把自己的想法告诉 AlphaGo，那么 AlphaGo 只能自己猜测对手的动作。AlphaGo 可以“推己及人”：如果 AlphaGo 认为几个动作很好，对手也会这么认为。所以 AlphaGo 用策略网络模拟对手，根据策略网络随机抽样一个动作：

$$a'_t \sim \pi(\cdot | s'_t; \theta).$$

此处的状态 s' 是站在对手的角度观测到的棋盘上的格局，动作 a'_t 是（假想）对手选择

18.2 蒙特卡洛树搜索 (MCTS)

的动作。图 18.5 的例子中对手有四种可行动作，AlphaGo 用策略网络算出每个动作的概率值，然后根据概率值随机抽样一个对手的动作，记作 a'_t 。假设根据概率值 0.1, 0.3, 0.2, 0.4 做随机抽样，选中第二种动作；见图 18.6。从 AlphaGo 的角度来看，对手的动作就是 AlphaGo 新的状态。

AlphaGo 需要在模拟中跟对手将一局游戏进行下去，所以需要一个模拟器（即环境）。在模拟器中，AlphaGo 每执行一个动作 a_k ，模拟器就会返回一个新的状态 s_{k+1} 。想要搭建一个好的模拟器，关键在于使用正确的状态转移函数 $p(s_{k+1}|s_k, a_k)$ ；如果状态转移函数与事实偏离太远，那么用模拟器做 MCTS 是毫无意义的。

AlphaGo 模拟器利用了围棋游戏的对称性：AlphaGo 的策略，在对手看来是状态转移函数；对手的策略，在 AlphaGo 看来是状态转移函数。最理想的情况下，模拟器的状态转移函数是对手的真实策略；然而 AlphaGo 并不知道对手的真实策略。AlphaGo 退而求其次，用 AlphaGo 自己训练出的策略网络 π 代替对手的策略，作为模拟器的状态转移函数。

想要用 MCTS 做决策，必须要有模拟器，而搭建模拟器的关键在于构造正确的状态转移函数 $p(s_{k+1}|s_k, a_k)$ 。从搭建模拟器的角度来看，围棋是非常简单的问题：由于围棋的对称性，可以用策略网络作为状态转移函数。但是对于大多数的实际问题，构造状态转移函数是非常困难的。比如机器人、无人机等应用，状态转移的构造需要物理模型，要考虑到力、运动、以及外部世界的干扰。如果物理模型不够准确，导致状态转移函数偏离事实太远，那么 MCTS 的模拟结果就不可靠。

第三步——求值 (Evaluation): 从状态 s_{t+1} 开始，双方都用策略网络 π 做决策，在模拟器中交替落子，直到分出胜负；见图 18.7。AlphaGo 基于状态 s_k ，根据策略网络抽样得到动作

$$a_k \sim \pi(\cdot | s_k; \theta).$$

对手基于状态 s'_k （从对手角度观测到的棋盘上的格局），根据策略网络抽样得到动作

$$a'_k \sim \pi(\cdot | s'_k; \theta).$$

当这局游戏结束时，可以观测到奖励 r 。如果 AlphaGo 胜利，则 $r = +1$ ，否则 $r = -1$ 。

回顾一下，棋盘上真实的状态是 s_t ，AlphaGo 在模拟器中执行动作 a_t ，然后模拟器中的对手执行动作 a'_t ，带来新的状态 s_{t+1} 。状态 s_{t+1} 越好，则这局游戏胜算越大。

- 如果 AlphaGo 赢得这局模拟 ($r = +1$)，则说明 s_{t+1} 可能很好；如果输了 ($r = -1$)，则说明 s_{t+1} 可能不好。因此，奖励 r 可以反映出 s_{t+1} 的好坏。
- 此外，还可以用价值网络 v 评价状态 s_{t+1} 的好坏。价值 $v(s_{t+1}; \mathbf{w})$ 越大，则说明状态 s_{t+1} 越好。

奖励 r 是模拟获得的胜负，是对 s_{t+1} 很可靠的评价，但是随机性太大。价值网络的评估 $v(s_{t+1}; \mathbf{w})$ 没有 r 可靠，但是价值网络更稳定、随机性小。AlphaGo 的解决方案是把奖励 r 与价值网络的输出 $v(s_{t+1}; \mathbf{w})$ 取平均，记作：

$$V(s_{t+1}) \triangleq \frac{r + v(s_{t+1}; \mathbf{w})}{2},$$

把它记录下来，作为对状态 s_{t+1} 的评价。

实际实现的时候，AlphaGo 还训练了一个更小的神经网络，它做决策更快。MCTS 在第一步和第二步用大的策略网络，第三步用小的策略网络。读者可能好奇，为什么在且仅在第三步用小的策略网络呢？第三步两个策略网络交替落子，通常要走一两百步，导致第三步成为 MCTS 的瓶颈。用小的策略网络代替大的策略网络，可以大幅加速 MCTS。

第四步——回溯 (Backup): 第三步——求值——算出了第 $t+1$ 步某一个状态的价值，记作 $V(s_{t+1})$ ；每一次模拟都会得出这样一个价值，并且记录下来。模拟会重复很多次，于是第 $t+1$ 步每一种状态下面可以有多条记录；如图 18.8 所示。第 t 步的动作 a_t 下面有多个可能的状态（子节点），每

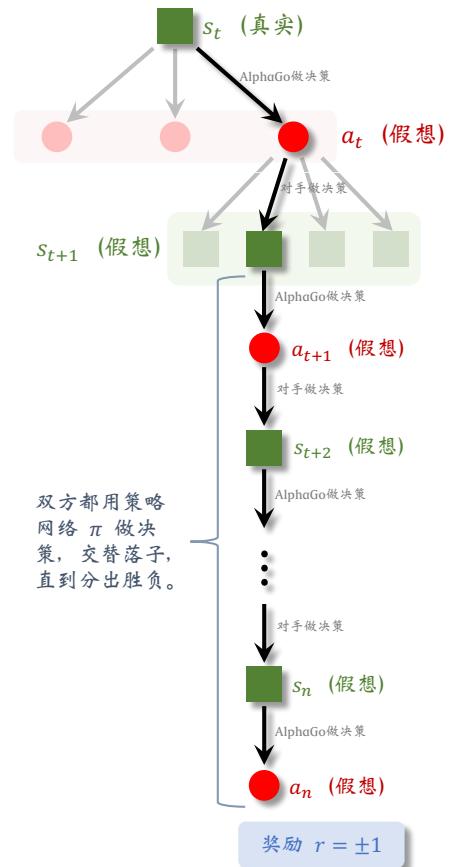


图 18.7：策略网络自我博弈。

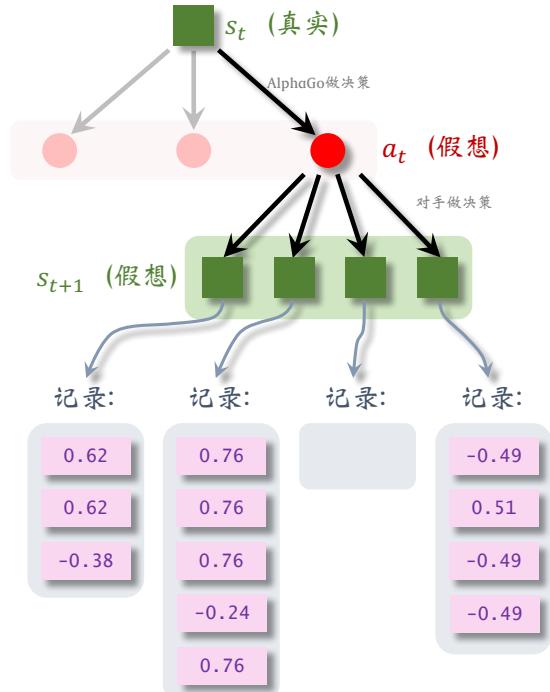


图 18.8：每一个状态 s_{t+1} 下面都有很多条记录，每一条记录是一个 $V(s_{t+1})$ 。

一个状态下面有若干条记录。把 a_t 下面所有的记录取平均，记作价值 $Q(a_t)$ ，它可以反映出动作 a_t 的好坏。在图 18.8 中， a_t 下面一共有 12 条记录， $Q(a_t)$ 是 12 条记录的均值。

给定棋盘上的真实状态 s_t ，有多个动作 a 可供选择。对于所有的 a ，价值 $Q(a)$ 的初始值是零。动作 a 每被选中一次（成为 a_t ），它下面就会多一条记录，我们就对 $Q(a)$ 做一次更新。

回顾第一步——选择 (Selection): 基于棋盘上真实的状态 s_t ，MCTS 需要从可行的动作中选出一个，作为 a_t 。MCTS 计算每一个动作 a 的分数：

$$\text{score}(a) \triangleq Q(a) + \frac{\eta}{1 + N(a)} \cdot \pi(a|s; \theta), \quad \forall a,$$

然后选择分数最高的 a 。MCTS 算出的 $Q(a)$ 的用途就是这里。

18.2.3 MCTS 的决策

上一小节讲解了单次模拟的四个步骤，注意，这只是单次模拟而已。MCTS 想要真正做出一个决策（即往真正的棋盘上落一个棋子），需要做成千上万次模拟。在做了无数次模拟之后，MCTS 做出真正的决策：

$$a_t = \underset{a}{\operatorname{argmax}} N(a).$$

此时 AlphaGo 才会真正往棋盘上放一个棋子。

为什么要依据 $N(a)$ 来做决策呢？在每一次模拟中，MCTS 找出所有可行的动作 $\{a\}$ ，计算它们的分数 $\text{score}(a)$ ，然后选择其中分数最高的动作，然后在模拟器里执行。如果某个动作 a 在模拟中胜率很大，那么它的价值 $Q(a)$ 就会很大，它的分数 $\text{score}(a)$ 会很高，于是它被选中的几率就大。也就是说如果某个动作 a 很好，它被选中的次数 $N(a)$ 就会大。

当观测到棋盘上当前状态 s_t ，MCTS 做成千上万次模拟，记录每个动作 a 被选中的次数 $N(a)$ ，最终做出决策 $a_t = \operatorname{argmax}_a N(a)$ 。到了下一时刻，状态变成了 s_{t+1} 。MCTS 把所有动作 a 的 $Q(a)$ 、 $N(a)$ 全部初始化为零，然后从头开始做模拟，而不能利用上一次的结果。

AlphaGo 下棋非常“暴力”：每走一步棋之前，它先在“脑海里”模拟几千、几万局，它可以预知它每一种动作带来的后果，对手最有可能做出的反应都在 AlphaGo 的算计之内。由于计算量差距悬殊，人类面对 AlphaGo 时不太可能有胜算。这样的比赛对人来说是不公平的；假如李世石下每一颗棋子之前，先跟柯洁模拟一千局，或许李世石的胜算会大于 AlphaGo。

18.3 训练策略网络和价值网络

上一节假设策略网络和价值网络已经训练好，并且用策略网络和价值网络辅助 MCTS。本节具体讲解如何训练两个神经网络。AlphaGo 有多个版本，其中最著名的是 2016、2017 年发表在 Nature 期刊的两个版本，本书称之为 2016 版和 AlphaGo Zero 版。AlphaGo Zero 实力更强：DeepMind 做了实验，让两个版本博弈 100 次，比分是 100 : 0。

18.3.1 AlphaGo 2016 版的训练

AlphaGo 2016 版的训练分为三步：第一，随机初始化策略网络 $\pi(a|s; \theta)$ 之后，用行为克隆 (Behavior Cloning) 从人类棋谱中学习策略网络；第二，让两个策略网络自我博弈，用 REINFORCE 算法改进策略网络；第三，基于已经训练好的策略网络，训练价值网络 $v(s; w)$ 。

第一步：行为克隆：一开始的时候，策略网络的参数都是随机初始化的。假如此时直接让两个策略网络自我博弈，它们会做出纯随机的动作。它们得随机摸索很多很多次，才能做出合理的动作。假如一上来就用 REINFORCE 学习策略网络，最初随机摸索的过程要花很久。这就是为什么 AlphaGo 2016 版基于人类专家的知识初步训练一个策略网络。

有一个叫 KGS 的在线围棋游戏程序，它在 2000 年的时候上线，让玩家在线比赛。KGS 会把每一局游戏都记录下来。KGS 有 16 万局是六段以上的高级玩家的记录。每一局游戏有很多步，每一步棋盘上的格局作为一个状态 s_k ，下一个棋子的位置作为动作 a_k ，这样得到数据集 $\{(s_k, a_k)\}$ 。数据集中一共有 $m = 2.94 \times 10^7$ 个 (s_k, a_k) 这样的二元组。

AlphaGo 用行为克隆训练策略网络 $\pi(a|s; \theta)$ 。第 12.1 节详细介绍了行为克隆，这里只是简单概括一下。设 361 维的向量

$$\mathbf{f}_k = \pi(\cdot | s_k; \theta) = [\pi(1 | s_k; \theta), \pi(2 | s_k; \theta), \dots, \pi(361 | s_k; \theta)]$$

是策略网络的输出，设 $\bar{\mathbf{a}}_k$ 是对动作 a_k 的 One-Hot 编码。函数 $H(\bar{\mathbf{a}}_k, \mathbf{f}_k)$ 是交叉熵 (Cross Entropy)，衡量 $\bar{\mathbf{a}}_k$ 与 \mathbf{f}_k 的差别。行为克隆可以描述成这样一个优化问题：

$$\min_{\theta} \frac{1}{m} \sum_{k=1}^m H(\bar{\mathbf{a}}_k, \mathbf{f}_k).$$

可以用随机梯度下降 (SGD) 求解这个优化问题。每次随机从 $\{1, \dots, m\}$ 中选出一个序号，记作 j 。设当前策略网络参数为 θ_{now} 。用随机梯度更新 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \eta \cdot \nabla_{\theta} H(\bar{\mathbf{a}}_j, \pi(\cdot | s_j; \theta_{\text{now}})).$$

此处的 η 是学习率。这样可以让策略网络的决策 $\pi(\cdot | s_k; \theta)$ 更接近人类高手的动作 $\bar{\mathbf{a}}_j$ 。

KGS 中的 16 万局游戏都是六段以上的高手的博弈。行为克隆得到的策略网络模仿高手的动作，可以做出比较合理的决策。它在实战中可以打败业余玩家，但是打不过职业玩家。第 12.1 节详细讨论过行为克隆的缺点。为了克服行为克隆的缺点，还需要继续用强化学习训练策略网络。在行为克隆之后再做强化学习改进策略网络，可以击败只用行为克隆的策略网络，胜算是 80%。

第二步——用 REINFORCE 训练策略网络：如图 18.9 所示，AlphaGo 让策略网络做自我博弈，用胜负作为奖励，更新策略网络。博弈的双方是两个策略网络，一个叫做“玩家”，用最新的参数，记作 θ_{now} ；另一个叫做“对手”，它的参数是从过时的参数中随机选出来的，记作 θ_{old} 。“对手”的作用相当于模拟器（环境）的状态转移函数，只是陪玩。训练的过程中，只更新“玩家”的参数，不更新“对手”的参数。



图 18.9：让两个策略网络自我博弈。

让“玩家”和“对手”博弈，将一局游戏进行到底，假设走了 n 步。游戏没结束的时候，奖励全都是零：

$$r_1 = r_2 = \cdots = r_{n-1} = 0.$$

游戏结束的时候，如果“玩家”赢了，奖励是 $r_n = +1$ ，于是所有的回报都是 $+1$: ¹

$$u_1 = u_2 = \cdots = u_n = +1.$$

如果“玩家”输了，奖励是 $r_n = -1$ ，于是所有的回报都是 -1 ：

$$u_1 = u_2 = \cdots = u_n = -1.$$

所有 n 步都用同样的回报，这相当于不区分哪一步棋走得好，哪一步走得烂；只要赢了，每一步都被视为“好棋”；假如输了，每一步都被看成“臭棋”。

REINFORCE 是一种策略梯度方法，它用观测到的回报 u 近似动作价值 Q_π 。REINFORCE 更新策略网络的公式是：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^n u_t \cdot \nabla \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

此处的 β 是学习率。

第三步——训练价值网络：价值网络 $v(s; \mathbf{w})$ 是对状态价值函数 $V_\pi(s)$ 的近似，用于评估状态 s 的好坏。在完成第二步——训练策略网络 π ——之后，用 π 辅助训练 v 。虽然此处有一个策略网络 π 和一个价值网络 v ，但这不属于 Actor-Critic 方法。此处先训练 π ，再训练 v ，用 π 辅助训练 v ；而 Actor-Critic 则是同时训练 π 和 v ，用 v 辅助训练 π 。

让训练好的策略网络做自我博弈，记录状态—回报二元组 (s_k, u_k) ，存到一个数组里。自我博弈需要重复非常多次，把最终得到的数据集记作 $\{(s_k, u_k)\}_{k=1}^m$ 。根据定义，状态价值 $V_\pi(s_k)$ 是回报 U_k 的期望：

$$V_\pi(s_k) = \mathbb{E}[U_k | S_k = s_k].$$

我们希望价值网络 $v(s_k; \mathbf{w})$ 接近 V_π ，也就是回报的期望，于是让 $v(s_k; \mathbf{w})$ 去拟合回报

¹回报的定义是 $u_t = r_t + r_{t+1} + \cdots + r_n$ ，折扣率是 $\gamma = 1$ 。

u_k 。定义回归问题 (Regression):

$$\min_{\mathbf{w}} \frac{1}{2m} \sum_{k=1}^m [v(s_k; \mathbf{w}) - u_k]^2.$$

可以用随机梯度下降 (SGD) 求解这个回归问题。设当前价值网络参数为 \mathbf{w}_{now} 每次随机从 $\{1, \dots, m\}$ 中选出一个序号，记作 j 。用价值网络做预测： $\hat{v}_j = v(s_j; \mathbf{w}_{\text{now}})$ 。用随机梯度更新 \mathbf{w} :

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{now}} - \alpha \cdot (\hat{v}_j - u_j) \cdot \nabla_{\mathbf{w}} v(s_j; \mathbf{w}_{\text{now}}).$$

此处的 α 是学习率。

18.3.2 AlphaGo Zero 版本的训练

AlphaGo Zero 与 2016 版本的最大区别在于训练策略网络 $\pi(a|s; \theta)$ 方式。训练 π 的时候，不再从人类棋谱学习，也不用 REINFORCE 方法，而是向 MCTS 学习。其实可以把 AlphaGo Zero 训练 π 的方法看做是模仿学习，被模仿对象是 MCTS。

自我博弈：用 MCTS 控制两个玩家对弈。每走一步棋，MCTS 需要做成千上万次模拟，并记录下每个动作被选中的次数 $N(a)$, $\forall a \in \{1, 2, \dots, 361\}$ 。设当前是 t 时刻，真实棋盘上当前状态是 s_t 。现在执行 MCTS，完成很多次模拟，得到 361 个整数（每种动作被选中的次数）：

$$N(1), N(2), \dots, N(361).$$

对这些 N 做归一化，得到的 361 个数，它们相加等于 1；把这 361 个数记作 361 维的向量：

$$\mathbf{p}_t = \text{normalize} \left(\left[N(1), N(2), \dots, N(361) \right]^T \right).$$

设这局游戏走了 n 步之后游戏分出胜负；奖励 r_n 要么等于 +1，要么等于 -1，取决于游戏的胜负。在游戏结束的时候，得到回报 $u_1 = \dots = u_n = r_n$ 。记录下这些数据：

$$(s_1, \mathbf{p}_1, u_1), (s_2, \mathbf{p}_2, u_2), \dots, (s_n, \mathbf{p}_n, u_n).$$

用这些数据更新策略网络 π 和价值网络 v ；对 π 和 v 的更新同时进行。

更新策略网络：上一节讨论过，MCTS 做出的决策优于策略网络 π 的决策，这就是为什么 AlphaGo 用 MCTS 做决策，而 π 只是用来辅助 MCTS。既然 MCTS 比 π 更好，那么可以把 MCTS 的决策作为目标，让 π 去模仿。这其实是行为克隆，被模仿的对象是 MCTS。我们希望 π 做出的决策

$$\mathbf{f}_t = \pi(\cdot | s_t; \theta) \in \mathbb{R}^{361}$$

尽量接近 $\mathbf{p}_t \in \mathbb{R}^{361}$ ，也就是让交叉熵 $H(\mathbf{p}_t, \mathbf{f}_t)$ 尽量小。定义优化问题：

$$\min_{\theta} \frac{1}{n} \sum_{t=1}^n H(\mathbf{p}_t, \pi(\cdot | s_t; \theta)).$$

设 π 当前参数是 θ_{now} 。做一次梯度下降更新参数：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} - \beta \cdot \frac{1}{n} \sum_{t=1}^n \nabla_{\theta} H(p_t, \pi(\cdot | s_t, \theta_{\text{now}})). \quad (18.2)$$

此处的 β 是学习率。

更新价值网络：训练价值网络的方法与 AlphaGo 2016 版本基本一样，都是让 $v(s_t; w)$ 拟合回报 u_t 。定义回归问题：

$$\min_w \frac{1}{2n} \sum_{t=1}^n [v(s_t; w) - u_t]^2.$$

设价值网络 v 当前参数是 w_{now} 。用价值网络做预测： $\hat{v}_t = v(s_t; w_{\text{now}})$, $\forall t = 1, \dots, n$ 。做一次梯度下降更新 w ：

$$w_{\text{new}} \leftarrow w_{\text{now}} - \alpha \cdot \frac{1}{n} \sum_{t=1}^n (\hat{v}_t - u_t) \cdot \nabla_w v(s_t; w_{\text{now}}). \quad (18.3)$$

训练流程：随机初始化策略网络参数 θ 和价值网络参数 w 。然后让 MCTS 自我博弈，玩很多局游戏；每完成一局游戏，更新一次 θ 和 w 。训练的具体流程就是重复下面三个步骤直到收敛：

1. 让 MCTS 自我博弈，完成一局游戏，收集到 n 个三元组： $(s_1, p_1, u_1), \dots, (s_n, p_n, u_n)$ 。
2. 按照公式 (18.2) 做一次梯度下降，更新策略网络参数 θ 。
3. 按照公式 (18.3) 做一次梯度下降，更新价值网络参数 w 。

∽ 第十八章 相关文献 ∽

早在很多年前，AI 就在棋类游戏中战胜了人类，比如国际象棋 (Chess) [24]，西洋跳棋 (Checker) [92, 91]，黑白棋 (Reversi 或 Othello) [22]，双陆棋 (Backgammon) [111]。这些棋类游戏的状态空间远比围棋的状态空间小，所以做搜索会相对比较容易。

AlphaGo 的论文首先发表在 Nature 2016 [98]。改进版本 AlphaGo Zero 发表在 Nature 2017 [100]。在 AlphaGo 之前一直有对围棋 AI 的探索，尽管 AI 尚无法击败人类围棋冠军。其中最有名的围棋 AI 包括 Pachi [11], Fuego [36], GNU Go (1999 年发布，2009 年停更)，Crazy Stone (2006 年发布)。Crazy Stone 虽然不及人类冠军，但是在对手让 4 子的情况下打败过 9 段高手。有兴趣的读者可以参考这些论文：[4, 79, 115, 18, 33, 36, 11]。

蒙特卡洛树搜索 (MCTS) 的名字最早在 2006 年发表的论文 [32] 中提出。另外两篇 2006 年的论文 [26, 59] 提出了类似的想法。2008 年发表的论文 [25] 将 MCTS 概括为今天我们众所周知的四个步骤。本书篇幅有限，不深入介绍 MCTS。有兴趣的读者可以阅读综述 [21] 和书 [27]。

第十九章 现实世界中的应用

强化学习最成功的应用莫过于 Atari、围棋等游戏，然而在现实中的落地应用还比较少。本章简要介绍强化学习的几个实际应用，希望对读者有一些启发。

19.1 神经网络结构搜索

传统的神经网络结构通常是由人手动设计的。以卷积神经网络 (CNN) 为例，众所周知的神经网络结构包括 LeNet、AlexNet、ResNet、GoogLeNet、MobileNet，它们都是由业内专家根据经验设计的，目的在于最大化测试准确率、或者最小化内存和计算开销。神经网络结构搜索 (Neural Architecture Search, NAS) 的意思是自动寻找最优的神经网络结构，代替手动设计的神经网络。2017 年的论文 [137] 开创性地将强化学习用于 NAS，找到的 CNN 结构优于人工设计的 CNN。这是强化学习非常成功的一个应用。遗憾的是，这种方法很快就被不用强化学习的方法超越。尽管如此，这篇论文的思想仍然具有启发意义。本节简要描述这种方法的思想；关心细节的读者可以去阅读原文。

19.1.1 超参数和交叉验证

为了解释神经网络结构搜索，需要先从**超参数 (Hyper-parameter)** 讲起。深度学习中有两类超参数：

- **结构超参数**包括层数、层的类别、层的大小等数值。以一个卷积层为例，其中的超参数包括卷积核 (filter) 的大小，卷积核的数量，步长 (stride) 的大小。这些超参数决定了神经网络的结构。
- **算法超参数**包括学习率 (learning rate)、批大小 (batch size)、epoch 数量、正则等。由于神经网络的非凸性，用不同的算法超参数会得到不同的解。

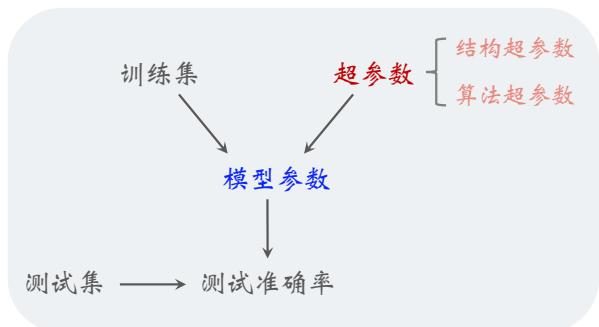


图 19.1：参数与超参数的关系。

图 19.1 解释了超参数与参数之间的关系。模型参数受超参数的控制；用不同的超参数，会学出不同的模型参数，从而会有不同的测试准确率。超参数与参数的区别是什么呢？两者之间未必有严格的界限。但通常来说，损失函数关于模型参数可微，因此可以用梯度算法学出模型参数。而损失函数关于超参数不可微，无法直接用梯度算法学出超参数。通常需要用**交叉验证 (Cross Validation)** 等方法搜索超参数。

在搜索超参数之前，需要手动指定候选的超参数。假设我们搭建 20 个卷积层，想要搜索其中的结构超参数。举个例子，我们手动指定这些候选超参数：

- 卷积核数量：{ 24, 36, 48, 64 }；
- 卷积核大小：{ 3 × 3, 5 × 5, 7 × 7 }；

- 步长大小: $\{1 \times 1, 2 \times 2\}$ 。

搜索空间 (Search Space) 是一个集合, 其中包含所有超参数的组合。在上述例子中, 搜索空间是这个笛卡尔积:

$$\{24, 36, 48, 64\}^{20} \times \{3 \times 3, 5 \times 5, 7 \times 7\}^{20} \times \{1 \times 1, 2 \times 2\}^{20}.$$

公式中的 20 是指 20 个卷积层。搜索空间中元素的数量等于 $(4 \times 3 \times 2)^{20} \approx 4 \times 10^{27}$ 。尽管每个超参数只有 2 ~ 4 个候选方案, 但搜索空间却无比巨大。

如何用交叉验证搜索超参数呢? 首先将训练数据随机划分成两部分, 比如 80% 做训练集 (Training Set), 20% 做验证集 (Validation Set)。然后重复下面的步骤很多次:

- 从搜索空间中均匀随机选出一组超参数的组合, 搭建卷积神经网络。
- 在训练集上训练神经网络, 从随机初始化开始, 一直到梯度算法收敛。
- 在验证集评价神经网络, 记录下验证准确率。

最后, 选出最高的验证准确率对应的超参数组合, 完成超参数搜索。上述随机超参数搜索的缺点是显而易见的:

- 第一, 每次搜索的代价都很大。从随机初始化到算法收敛, 花费的时间少则几十分钟, 多则几天。如果 GPU 数量有限的话, 顶多只能尝试几千、几万种超参数组合。
- 第二, 搜索空间过于巨大。在上述例子中, 搜索空间中有 4×10^{27} 种超参数组合。如果把搜索空间比做海洋, 那么几万种超参数组合相当于一克的水。随机搜索超参数就像是海底捞针。
- 第三, 由于随机性, 验证准确率最高的超参数组合未必是最好的。随机性来自于随机初始化、随机梯度、数据集的随机划分。在验证集上, 某个超参数的组合取得最高的准确率, 其中有很大的运气成分; 在测试集上, 这个超参数的组合未必能取得很高的准确率。

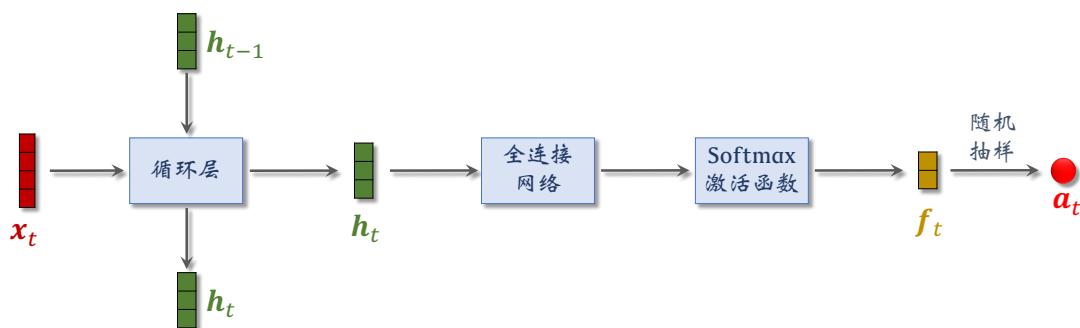


图 19.2: 图中是 RNN 策略网络, 它输出概率分布 f_t 。根据 f_t 抽样得到的动作 a_t 是一个超参数。

19.1.2 强化学习方法

2017 年的论文 [137] 设计一种强化学习方法, 用于学习神经网络结构。如图 19.2 所示, 策略网络是一个循环神经网络 (RNN); 不熟悉 RNN 的读者请回顾第 11 章。策略网络的输入向量 x_t 是对上一个超参数 a_{t-1} 做 Embedding 得到的¹。循环层的向量 h_t 可以

¹向量 x_0 是例外; x_0 是用一种特殊的方法随机生成的。

看做从序列 $[x_1, \dots, x_t]$ 中提取的特征。可以把 $s_t = [x_t; h_{t-1}]$ 看做第 t 个状态。策略网络的输出向量 f_t 是一个概率分布。根据 f_t 做随机抽样，得到动作 a_t ，即第 t 个超参数。

策略网络是如何生成神经网络结构的？下面举一个具体的例子。假设我们搭建 20 个卷积层，每个层有 3 个超参数，那么一共有 60 个超参数。每一层的 3 个超参数从下面的候选方案中选择。

- 卷积核数量: $\{24, 36, 48, 64\}$;
- 卷积核大小: $\{3 \times 3, 5 \times 5, 7 \times 7\}$;
- 步长大小: $\{1 \times 1, 2 \times 2\}$ 。

按照图 19.3 的描述，依次生成每一层的卷积核数量、卷积核大小、步长大小。在 RNN 运行 60 步之后，得到 60 个超参数，也就确定了 20 个卷积层的结构。

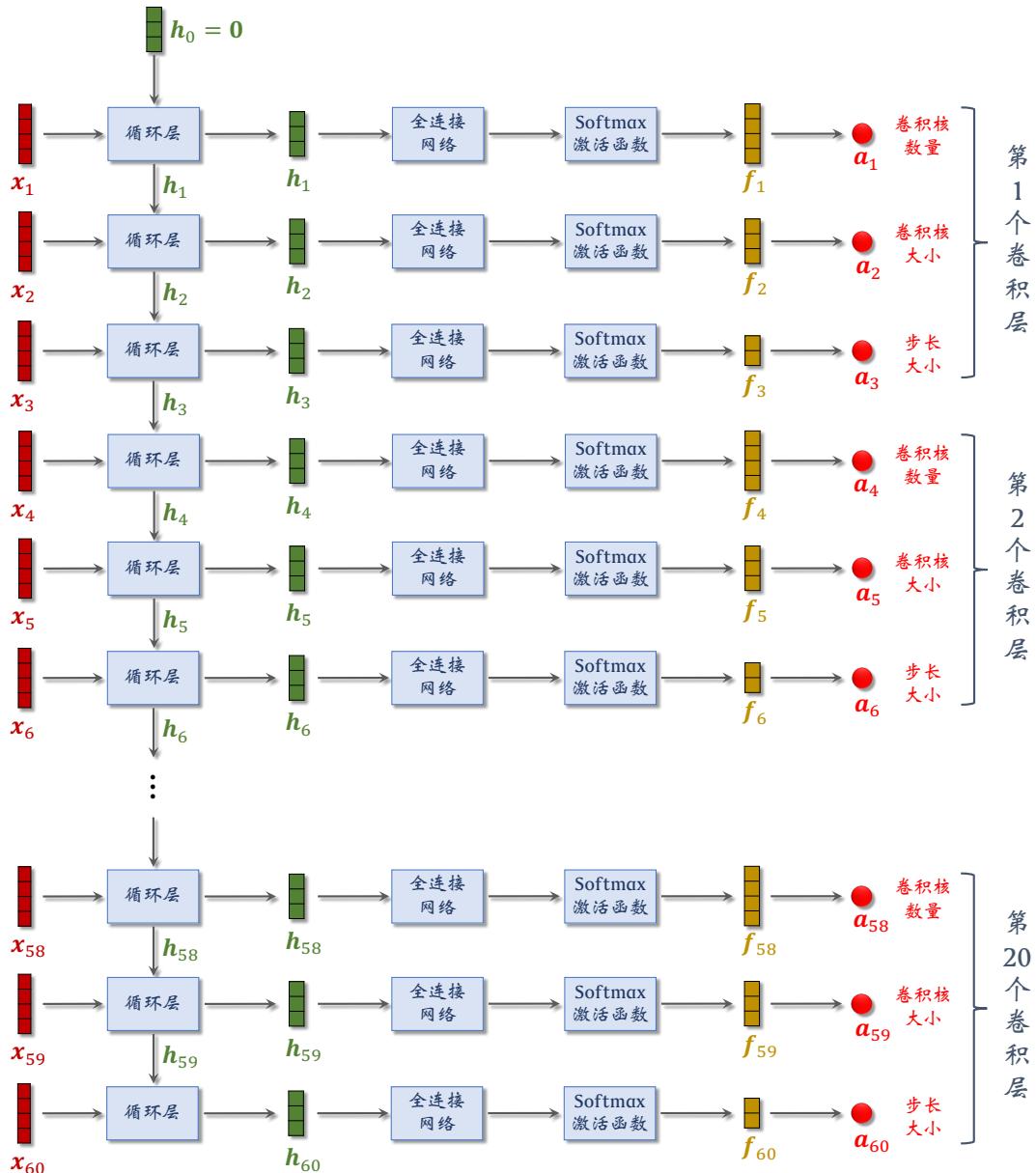


图 19.3：用 RNN 策略网络依次生成每一层的 3 个超参数。图中的向量 x_t 是 a_{t-1} 做 Embedding 得到的。循环层共享参数，而全连接层、Embedding 层不共享参数。

该如何训练策略网络呢？为了训练策略网络，我们需要定义奖励 r_t 。在前 59 步，奖励全都是零： $r_1 = \dots = r_{59} = 0$ 。在第 60 步之后，得到了全部的超参数，确定了神经网络结构。然后搭建神经网络，在训练集上学习神经网络参数，直到梯度算法收敛。在验证集上评价神经网络，得到验证准确率，作为奖励 r_{60} 。由回报的定义 $u_t = r_1 + \dots + r_t$ 可得：

$$u_1 = u_2 = \dots = u_{60} = \text{验证准确率}.$$

我们希望通过更新 RNN 策略网络的参数，使得回报越来越大，即生成的 CNN 的验证准确率越来越高。把策略网络记作

$$\pi(a_t | s_t; \theta),$$

其中 a_t 是动作（即超参数）， $s_t = [x_t, h_{t-1}]$ 是状态， θ 是 RNN 策略网络的参数。可以用 REINFORCE 算法更新参数 θ ：

$$\theta_{\text{new}} \leftarrow \theta_{\text{now}} + \beta \cdot \sum_{t=1}^{60} u_t \cdot \nabla_{\theta} \ln \pi(a_t | s_t; \theta_{\text{now}}).$$

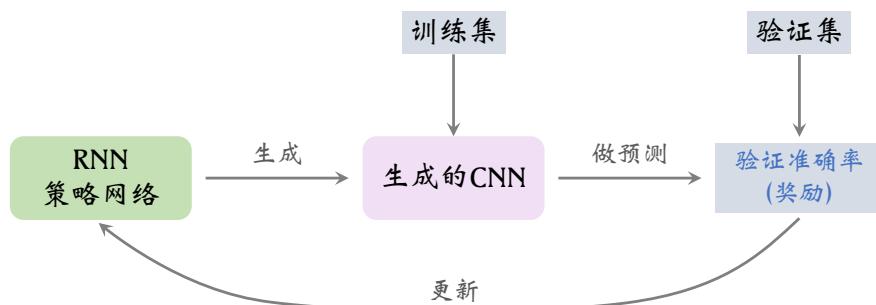


图 19.4：训练 RNN 策略网络的流程。

训练 RNN 策略网络的流程如图 19.4 所示。我们的目标是找到一个好的 CNN 结构，但是需要借助一个 RNN 策略网络。因为目标是让 CNN 获得尽量高的验证准确率，所以用验证准确率作为奖励。这种神经网络结构搜索的计算量非常巨大。每获得一个奖励 r_{60} ，都需要从随机初始化开始训练 CNN，直到梯度算法收敛；这个过程少则几十分钟、多则几天。需要重复图 19.4 中流程上万次才能训练好 RNN 策略网络，其计算代价可想而知。

请读者思考一个问题：为什么一定要用强化学习方法来训练 RNN 策略网络？是不是因为强化学习比传统监督学习更有优势？答案恰恰相反，强化学习并不好，只是此处不得不用而已。如果想要做传统的监督学习，那么奖励或损失必须关于 RNN 策略网络参数 θ 可微；本节介绍的方法显然不符合这个条件，所以不能用监督学习训练 RNN 策略网络。强化学习的奖励可以是任意的，无需关于 θ 可微，因此在这里适用。应用强化学习的代价是需要大量的训练样本，至少上万个奖励，即从初始化开始训练几万个 CNN。这种强化学习 NAS 方法的计算量非常大。在这种强化学习 NAS 方法提出之后，很快就有更好的 NAS 方法出现，无需使用强化学习。有兴趣的读者可以了解一下 DARTS 方法 [71]；DARTS 及其变体是比较实用的 NAS 方法。

19.2 自动生成 SQL 语句

Structured Query Language (结构化查询语言), 缩写 SQL, 用于管理关系数据库。SQL 支持数据插入、查询、更新、删除。将人的语言转化成 SQL 是自然语言处理领域的一个重要问题。举个例子, 在订票网站自动对话系统中, 用户提出一个问题:

“请找出 2021 年 10 月 1 日从北京直飞纽约的航班, 按照价格从低到高排序。”

程序需要生成 SQL 语言, 查找符合日期、起点、终点的直飞航班, 并且按照价格排序。解决这个问题的方法类似于机器翻译, 即用 Transformer 等 Seq2Seq 模型将一句自然语言翻译成 SQL 语言; 见图 19.5。

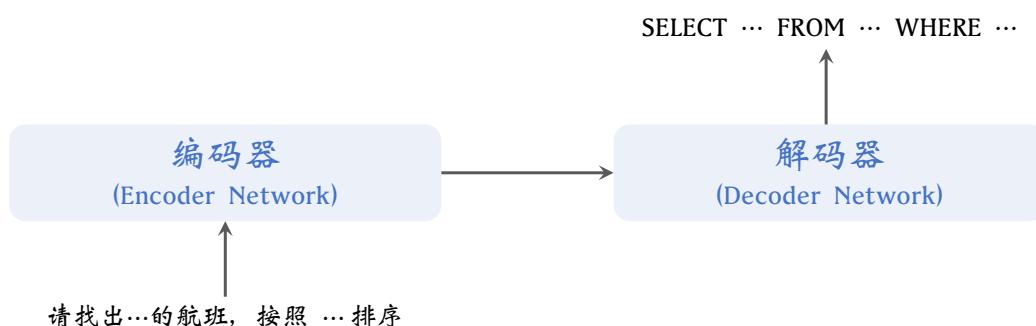


图 19.5: 用 Transformer 等 Seq2Seq 模型将自然语言翻译成 SQL。

该如何训练图 19.5 这样的 Seq2Seq 机器翻译模型呢? 最简单的方式就是用监督学习。事先准备一个数据集, 由人工将自然语言逐一翻译成 SQL 语句。训练的目标是鼓励解码器输出的 SQL 语句接近人工标注的 SQL 语句。把解码器的输出、人工标注的 SQL 两者的区别作为损失函数, 通过最小化损失函数的方式训练模型。这种单词匹配的训练方式是可行的, 然而其存在一些局限性。

与标准机器翻译问题相比, SQL 语句的生成有其特殊性。如果是将一句汉语译作英语, 那么个别单词的翻译错误、顺序错误不太影响人类对翻译结果的理解。对于汉语翻译英语, 可以把单词的匹配作为评价机器翻译质量的标准。但是这种评价标准不适用于 SQL 语句。

- 即使两个 SQL 语句高度相似, 它们在数据库中执行得到的结果可能完全不同。即便是一个字符的错误, 也可能导致生成的 SQL 语法错误, 无法执行。
- 哪怕两个 SQL 语句看似区别很大, 它们的作用是完全相同的, 它们在数据库中执行得到的结果是相同的。
- SQL 的写法会影响执行的效率, 而从 SQL 语句的字面上难以看出它的效率。只有真正在数据库中执行, 才知道 SQL 语句究竟花了多长时间。

以上论点说明不该用单词的匹配来衡量生成 SQL 语句的质量, 而应该看 SQL 语言实际执行的结果是否符合预期。

2017 年的论文 [135] 提出一种强化学习的方法训练 Seq2Seq 模型，如图 19.6 所示。可以把 Seq2Seq 模型看做策略网络，把输入的自然语言看做状态，把生成的 SQL 看做动作。他们这样定义奖励：

$$r = \begin{cases} -2, & \text{生成的 SQL 语句不能运行;} \\ -1, & \text{生成的 SQL 语句可以运行, 但是结果不符合预期;} \\ +1, & \text{生成的 SQL 语句可以运行, 而且结果符合预期.} \end{cases}$$

有了奖励，可以用任意的策略学习算法，比如 REINFORCE 和 Actor-Critic。论文 [135] 使用 REINFORCE 算法训练 Seq2Seq 模型。

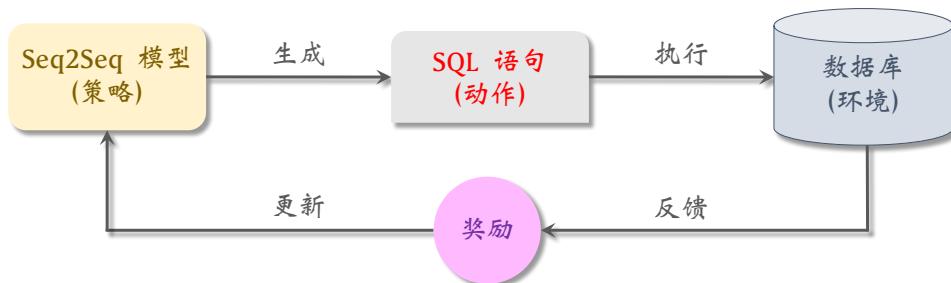


图 19.6：用强化学习训练 Seq2Seq 模型的流程。

对比一下监督学习和强化学习方法。监督学习鼓励模型生成的 SQL 语句接近人类专家写的 SQL，其本质是行为克隆，即鼓励模型的决策接近人类专家的动作。而上述强化学习则不同，并没有简单模仿人类专家，而是在数据库中实际执行 SQL 语句，根据执行的结果来更新策略。强化学习让策略 (Seq2Seq 模型) 与环境 (数据库) 实际交互，而监督学习 (即行为克隆) 并没有与环境交互。

本书介绍论文 [135]，是因为这篇论文的想法比较有意思，非常符合强化学习的设定，强化学习可以克服传统监督学习的局限性。这篇论文的实验结果不够强，很可能只是这篇论文的方法和实现不够好而已，不意味着强化学习不适用于 SQL 语句的生成。强化学习的效果好坏取决于多重因素，比如策略网络的设计、策略网络的初始化、策略学习的算法、奖励的定义、甚至是超参数调得是否够好。每个因素都严重影响强化学习的实验效果。除了本节介绍的 SQL 语句生成，强化学习在 Seq2Seq 模型上有很多应用，读者可以参考 2019 年综述 [58] 以及其中的文献。

19.3 推荐系统

网站有海量的物品，比如 YouTube 的视频、京东的商品、美团外面的店铺。网站有百万、甚至上亿的用户，每个用户有各自的喜好，喜好可以从他的点击、观看、购买等历史记录中反映出来。个性化推荐的目标是将用户感兴趣的物品展示给用户，从而最大化某些指标（比如点击率、观看时长、购买率、消费金额）。

推荐系统是工业界最推崇的机器学习技术之一，好的推荐系统可以带来大量的流量和营收。推荐系统是一个历史悠久、而又热门的研究领域。近年来，在应用深度学习技术之后，推荐系统的效果取得了大幅的提升。强化学习在推荐系统中有一些应用，但应用远不如传统监督学习推荐系统广泛。

推荐系统的背景知识很多，本书无法用较短的篇幅讲清楚强化学习推荐系统的原理。下面只简单介绍其基本思想。对强化学习推荐系统感兴趣的读者可以阅读以下论文：YouTube 的推荐系统 [28]、京东的推荐系统 [134]、阿里巴巴的推荐系统 [55]。

如图 19.7 所示，强化学习推荐系统的**策略**是指根据用户的兴趣点，从海量物品中选出一个或几个，展示给用户。用户的兴趣点就是**状态 s** ，可以从用户人口信息、地理位置、社交关系、历史活动记录（包括点击、观看、购买记录）这些数据中反映出来。被选中的物品就是**动作 a** 。策略网络输出的向量 f 的维度是动作空间的大小 $|A|$ 。商家的物品种类非常多，因此动作空间 A 非常巨大， f 的维度非常高。简单粗暴地训练策略网络是行不通的，必须使用很多技巧做训练；具体可以参考 YouTube 论文 [28]。

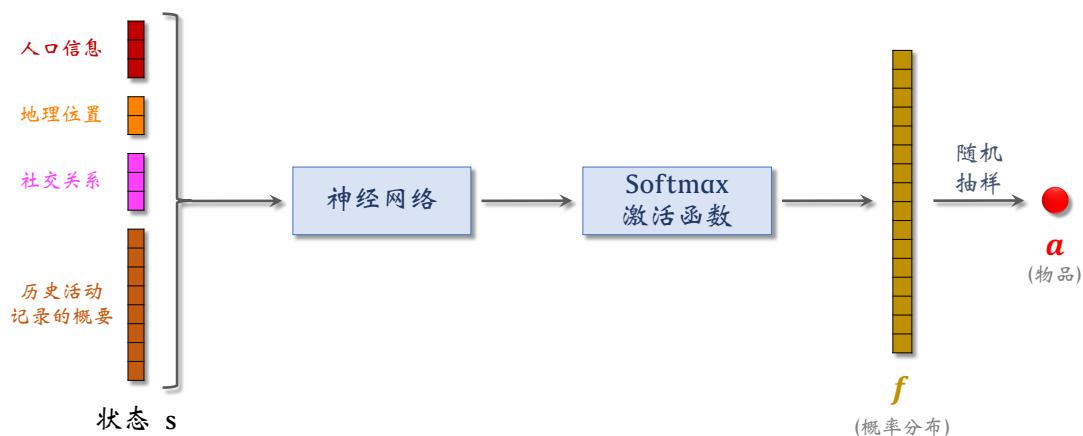


图 19.7：策略网络的一种设计方法。

强化学习推荐系统的奖励需要根据实际问题，由系统的开发者自己来定。比如在 YouTube 视频网站上，点击、观看时长、点赞都可以作为奖励。比如在京东购物网站上，点击、浏览时间、加购物车、购买、消费金额都可以作为奖励。在设计奖励的时候，需要格外小心，避免造成意想不到的结果：

- 某视频网站、某新闻网站想提升点击率，把点击率作为重要的奖励之一。结果大量骗点击的标题党的排序大幅提升，用户满屏尽是“吓尿了”、“震惊了”。
- 某外卖平台想要提高用户使用 APP 的时长，把时长作为重要的奖励之一。结果系

统把每个用户经常吃的店铺排到后面，用户需要花更多时间翻页寻找自己想要的店铺，增加了使用 APP 的时间。

以上是网上流传的段子，未必真实。但是如果你这样设计奖励，你的产品可能会成为新的段子。

强化学习推荐系统的一个难点在于探索过程的代价很大；此处的代价不是计算代价，而是实实在在的金钱代价。强化学习要求智能体（即推荐系统）与环境（即用户）交互，用收集到的奖励更新策略。如果直接把一个随机初始化的策略上线，那么在初始探索阶段，这个策略会做出纯随机的推荐，严重影响用户体验，导致点击、观看、购买数量暴跌，给公司业务造成损失。在上线之前，必须在线下用历史数据初步训练策略。最简单的方法是在线下用监督学习的方式训练策略网络，这很类似传统的深度学习推荐系统。阿里巴巴提出的“虚拟淘宝”系统 [95] 模仿人类用户，得到很多虚拟用户，把这些虚拟用户作为模拟器的环境。把推荐系统作为智能体，让它与虚拟用户交互，利用虚拟的交互记录来更新推荐系统的策略。等到在模拟器中把策略训练得足够好，再让策略上线，与真实用户交互，进一步更新策略。

19.4 网约车调度

滴滴是中国最大的网约车平台。乘客在手机 APP 中指定起点和终点，得到预估报价；在乘客确认订单之后，滴滴把订单派发给临近的司机。在同一时刻，有多个用户下单，附近有多辆空车，该如何派发订单才能最大化网约车司机的收入呢？滴滴用强化学习方法解决订单派发问题，显著提高了网约车司机的收入 [110]。

在讲解强化学习方法之前，先来看两个具体的例子。如图 19.8 所示，两个乘客同时下单，而附近只有一辆空车，该给司机派发谁的订单？如图 19.9 所示，一个乘客下单，而附近有两辆空车，该把订单派发给哪个司机？请注意，滴滴派发订单的目的在于最大化司机的总收入，这样既有利于留住司机，也可以最大化滴滴公司的抽成收入。



图 19.8：两个乘客同时下单，附近只有一辆空车，该给司机派发谁的订单？



图 19.9：一个乘客下单，附近有两辆空车，该把订单派发给哪个司机？

对于图 19.8 中的例子，假如不考虑目的地的热门程度（即附近接单的容易程度），则应该给司机派发上面蓝色目的地的订单，这样可以让司机在较短的时间内取得更高的收入。但是这样其实不利于司机的总收入：在司机到达冷门地点之后，需要等待较长的时间才会有新的订单。假如给司机派发下面热门目的地的订单，司机在完成这笔订单后，立刻就能接到下一笔订单；这样虽然单笔收入低，但是总收入高。

对于图 19.9 中的例子，很显然应该把订单派送给冷门地点的司机更合适。热门地点的司机得不到这笔订单几乎没有损失，因为在很短的时间之后就会有新的订单。而这笔订单对冷门地点的司机比较重要，如果没有这笔订单，司机还需要空等很久才有下一笔订单。

19.4.1 价值学习

该如何量化一个地点的热门程度呢？把司机每一笔订单的收入作为奖励，把折扣回报的期望作为状态价值函数 $V_\pi(s, w)$ ，用它来衡量热门程度。公式中 $s = (\text{地点}, \text{时间})$ 是状态， π 是派单的策略。 $V_\pi(s, w)$ 可以衡量一个地点在具体某个时间的热门程度。滴滴 2019 年论文 [110] 的目标是学习 $V_\pi(s, w)$ ，从而指导订单派发。这种强化学习方法属于价值学习。

状态价值函数 V_π 的作用在于预判某个地点在某个时间的热门程度。比如在早高峰，车流从居民区开往商业区，导致商业区是冷门地点，附近空车多，订单少。而到了晚高峰，商业区是热门地点，此时下班回家的需求大，订单数量多。从大数据中不难找出这种规律。

滴滴用价值网络近似 $V_\pi(s)$ ，并且用 TD 算法训练价值网络。具体的实现比较复杂，此处就不具体描述了。值得注意的是，在学习的过程中要用正则项，使得价值网络是光滑的。为什么呢？当状态 $s = (\text{地点}, \text{时间})$ 中的地点、时间发生较小的变化时，价值网络的输出不应该剧烈变化。

19.4.2 订单派单机制

在学到状态价值函数 $V_\pi(\text{地点}, \text{时间})$ 之后，可以用它来预估任意地点、时间的网约车的价值，并利用这一信息来给网约车派发订单。主要想法是用负的 TD 误差来评价一个订单给一个网约车带来的额外收益。在同一时刻，某区域内有 m 笔订单，有 n 个空车，那么计算所有（订单，空车）二元组的 TD 误差，得到一个 $m \times n$ 的矩阵。用二部图 (Bipartite Graph) 匹配算法，找订单—空车的最大匹配，完成订单派发。



图 19.10：某乘客在 9:10 的时候下单，滴滴计算在 (起点, 9:10) 和 (终点, 9:43) 的状态价值，从而计算出 TD 误差。

首先用图 19.10 中的例子解释如何计算 TD 误差。简单起见，此处设折扣率 $\gamma = 1$ ，尽管滴滴使用的折扣率小于 1。对于图中的例子，TD 目标等于：

$$\hat{y} = r + V_\pi(\text{终点}, 9:43) = 40 + 480 = 520.$$

可以这样理解 TD 目标 \hat{y} ：假设给该空车派发该订单，那么该笔订单的价值 $r = 40$ 加上未来的状态价值，一共等于 $\hat{y} = 520$ 。但是司机接这笔订单是有机会成本的；假如不接这笔订单，马上就会有别的订单，可能会获得更高的 TD 目标。机会成本是 $V_\pi(\text{起点}, 9:10) = 500$ ，即从当前开始的一定时间内获得的总收入的期望等于 500。用 TD 目标减去机会成本，即

负的 TD 目标:

$$-\delta = \hat{y} - V_\pi(\text{起点}, 9:10) = 520 - 500 = 20.$$

这意味着接这笔订单，司机的收入高于期望收入 20 元。

滴滴的订单派发正是基于上述 TD 误差。举个例子，在某个区域，当前有 3 笔订单，有 4 辆空车。滴滴计算每个（订单，空车）二元组的 TD 误差，得到图 19.11 中大小为 3×4 矩阵。

	空车 #1	空车 #2	空车 #3	空车 #4
订单 #1	$-\delta_{1,1} = 20$	$-\delta_{1,2} = 10$	$-\delta_{1,3} = 12$	$-\delta_{1,4} = -5$
订单 #2	$-\delta_{2,1} = -2$	$-\delta_{2,2} = 7$	$-\delta_{2,3} = 0$	$-\delta_{2,4} = -1$
订单 #3	$-\delta_{3,1} = 12$	$-\delta_{3,2} = -3$	$-\delta_{3,3} = 3$	$-\delta_{3,4} = 3$

图 19.11: 在某个区域，当前有 3 笔订单，有 4 辆空车。滴滴计算每个（订单，空车）二元组的 TD 误差，得到这个矩阵。

有了上面的矩阵，可以调用二部图匹配算法（比如匈牙利算法）来匹配订单和空车。图 19.12(左) 是最大匹配，三条边的权重之和等于 31，滴滴按照这种匹配派发订单。图 19.12(右) 也是一种匹配方式，但是三条边的权重之和只有 30，说明它不是最大匹配，滴滴不会这样派发订单。

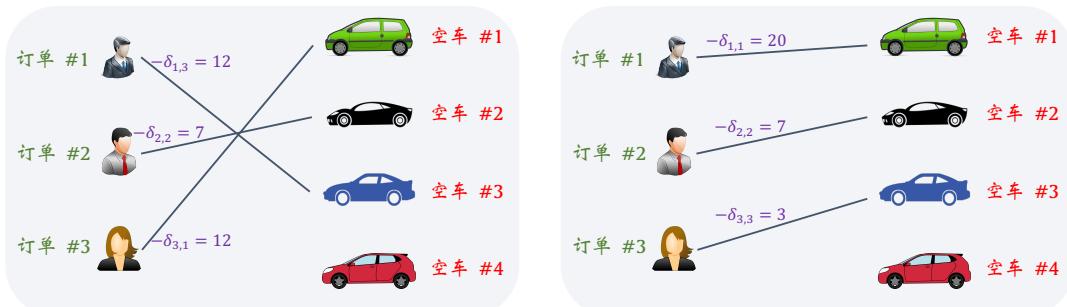


图 19.12: 左图是最大匹配，三条边的权重之和等于 31。右图是另一种匹配，但不是最大匹配，三条边的权重之和等于 30。

19.5 强化学习与监督学习的对比

强化学习哪里都可以用，但是多数场景下毫无使用强化学习的必要；能用监督学习很好解决的，没必要用强化学习。本节讨论强化学习与监督学习的区别，举例分析几种强化学习有优势的场景。希望读者在理解本节内容之后，有能力判断那些是强化学习有前景的应用，哪些是强化学习的“伪应用”。

19.5.1 决策是否改变环境？

监督学习假设模型的决策不会影响环境，而强化学习假设模型的决策会改变环境。在实际问题中，模型的决策究竟会不会影响环境呢？举个例子，如果你是小散户，你的交易（即动作）几乎不会影响股价（即环境）；如果你是大投资机构，你的大笔交易肯定会改变股价。如果你是小散户，你手中有 100 支某股票，股价是 50 元；全部卖出得到的现金是 5,000 元。如果你是投资机构，你手上有一千万支该股票，你在二级市场全部卖出；卖出的过程可能会持续几个小时，期间股价肯定会连续下跌，你最终得到的现金会远小于五亿元。假如投资机构用想用机器学习做股票交易，必须要考虑到决策对环境的影响。

再举个例子，如图 19.13 所示，在 Zillow 等房地产网站上，待售房屋有卖家的标价，下面还有 Zillow 自动评估出的参考价格。究竟 Zillow 具体如何给房屋估价，我们无从得知。假设由你来开发房屋估价模型，请问你应该用监督学习，还是用强化学习？答案取决于 Zillow 给出的估价是否会干扰成交价。如果 Zillow 给出的估价不影响买家心理，不干扰成交价，那么直接用回归模型去拟合成交价即可。如果 Zillow 给出的估价会影响成交价，那么强化学习或许更为合适。可以把估价模型看做策略，把计算出的价格看做动作。将估价展示在 Zillow 上，可能会影响买家心理，因此改变房地产市场（环境），影响成交价。

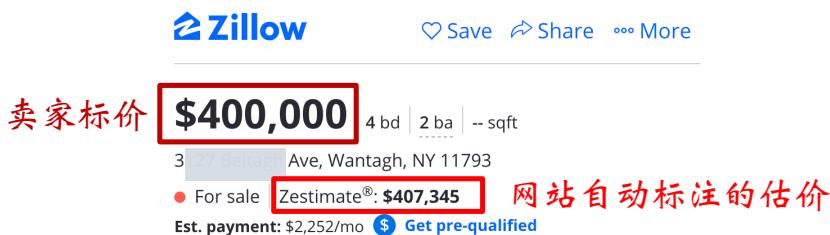


图 19.13：Zillow 网站上待售房屋有两个价格，一个是卖家标价，另一个是 Zillow 给出的估价。

在推荐系统中，推荐算法相当于策略，而用户的兴趣点相当于环境，推荐的内容（动作）会改变用户兴趣点（环境）。举个例子，我原本对养殖业没有兴趣，但是在 YouTube 给我推送竹鼠养殖的视频之后，我对此产生很大兴趣，喜欢点击竹鼠的视频。这说明推荐系统并非只能被动迎合用户喜好，推荐系统完全可以主动创造用户的兴趣点。监督学习假设用户兴趣点（环境）是固定的，推荐系统只会拟合用户的喜好，推荐相似的物品。而强化学习则假设用户的兴趣点可以被改变，学出的推荐策略会发掘用户新的兴趣点。

19.5.2 是否需要探索未知的动作？

继续讨论推荐系统。思考一下，为什么强化学习推荐系统可以发现用户新的兴趣点，而监督学习推荐系统却不可以呢？这是因为强化学习允许探索，尝试历史数据中不存在的动作。比如，给一个不看美食节目的用户推荐厨师王刚，给不看农牧的用户推荐竹鼠养殖，给不懂编程的人推荐 Python 编程。说不定用户就点击视频了，而且在看完之后对此类内容产生浓厚兴趣，观看更多此类视频。在这种情况下，给策略反馈较高的奖励。受到奖励的引导，推荐策略学会开发用户新的兴趣点，并在已有兴趣和新兴趣之间寻找平衡。与强化学习不同，监督学习通常不做探索，只是拟合历史记录，根据用户已有的兴趣点做推荐，无法学会挖掘用户新的兴趣点。

打个比方，两位皮鞋推销员去了某国，发现当地的人不穿鞋。推销员甲：“既然当地人不穿鞋，那么当地没有市场，我们还是走吧。”推销员乙：“既然当地人不穿鞋，那么每个人都是潜在的客户，应当给他们试穿，培养他们穿鞋的习惯。”推销员甲相当于监督学习，依据已有兴趣点做推荐。推销员乙相当于强化学习，会尝试新的动作，发掘潜在的兴趣点。

之前章节中讲过强化学习比行为克隆（监督学习的一种）的效果更好。其原因就在于强化学习会探索尽量多的状态和动作，不至于见到不熟悉的状态就不知所措。而行为克隆只会模仿专家的动作，不做探索，在见到不熟悉的状态是会做出很差的决策。

传统监督学习通常不做探索，但这也不是绝对的，监督学习也可以做探索。比如试验设计 (Experimental Design)、贝叶斯优化 (Bayesian Optimization) 研究的问题就是“在什么地方探索”。具体来说，我们想要训练函数 $f(\mathbf{x})$ 拟合 y ，而样本 (\mathbf{x}, y) 的数量非常有限，每获得一个新的样本的代价都非常大，比如钻一个几百米的洞勘探矿藏、撞毁一辆车判断其安全性、电话访谈一位客户了解其满意度。实验设计的目的是根据已知的 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ ，计算出 \mathbf{x}_{n+1} ，然后基于 \mathbf{x}_{n+1} 做试验得到 y_{n+1} 。比如，已经在 $\mathbf{x}_1, \dots, \mathbf{x}_n$ 这 n 个位置钻了洞，得到 n 组数据，下一步该在哪里钻洞，即 \mathbf{x}_{n+1} 取多少？

19.5.3 当前的奖励还是长线的回报？

使用监督学习或是强化学习，还取决于目标是当前的奖励还是长线的回报。人脸识别这类问题属于“一锤子买卖”，只需要关注当前的奖励即可，因此适用于监督学习。象棋等游戏则应该考虑长线回报：吃掉对方一个马，虽然得到了眼前的利益，但是可能不利于赢得这局棋。

在滴滴派发订单的应用中，存在当前奖励和长线回报的问题。眼前奖励就是从当前订单中获取的收益，即单位时间内获得的收入；以图 19.14 为例，单位时间的奖励是 $\frac{40}{33}$ 元。我们之前讨论过，仅仅最大化眼前利益是不行的，这样无法最大化长期回报（即总收入）。一方面，目的地有“冷”和“热”之分，会影响司机后续的等待时间和收入。另一方面，接单虽然能立刻赚到钱，但是会花费“机会成本”，如果稍等一下可能会接到更好的单。出于这两方面的考虑，滴滴使用强化学习的方法，最大化长线回报（总收入），而不是眼前的奖励（单笔订单的收入）。



在视频网站推荐系统的应用中，推荐通常不是“一锤子买卖”，而是为了最大化用户的观看时长。因此，长线的回报比当前的奖励更重要。如图 19.15 所示，根据已有兴趣做推荐，立刻获得较高的奖励；而尝试挖掘新的兴趣爱好，眼前收益较小，但是有利于获得很高的长期回报。这就是为什么工业界用动力去尝试强化学习推荐系统。

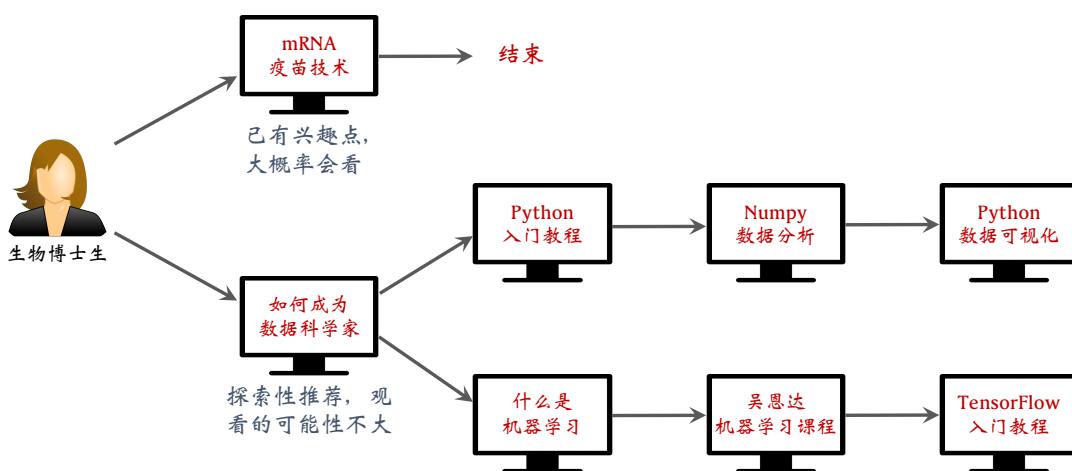


图 19.15：给用户推荐她感兴趣的内容，点击率会比较高。如果尝试新的兴趣点，点击率会很低。可是一旦给用户培养了新的兴趣点，用户会看更多相关内容，总的观看时长会大幅增长。

19.6 什么在制约深度强化学习的应用？

到目前为止，深度强化学习最成功、最有名的应用仍然是 Atari 游戏、围棋游戏、星际争霸游戏。深度强化学习有很多现实中的应用，但其中成功的应用并不多。本节探讨究竟是什么在制约深度强化学习的落地应用。

19.6.1 所需的样本数量过大

深度强化学习一个严重的问题在于需要巨大的样本量。举个例子，如图 19.17 所示，Atari 游戏属于最简单的电子游戏，在现实世界中找不到这么简单的问题。2015 年的论文 [77] 用 DQN 玩 Atari 游戏，取得了超越人类玩家的分数，在学术界内引起了轰动。2015 年提出的原始的 DQN 存在诸多问题，实验效果不够好。2018 年的论文 [49] 提出 Rainbow DQN，将多种技巧结合，让 DQN 的训练变得更快更好。论文 [49] 在 57 种 Atari 游戏上比较了原始 DQN、多种高级技巧、以及 Rainbow DQN。图 19.17 中纵轴是算法的分数与人类分数的比值，并关于 57 种游戏求中位数；100% 表示达到人类玩家的水准。图中横轴是收集到的游戏帧数，即样本数量。Rainbow DQN 需要 1 千 8 百万帧才能达到人类玩家水平，超过 1 亿帧还未收敛；前提是已经调优了超过 10 种超参数。

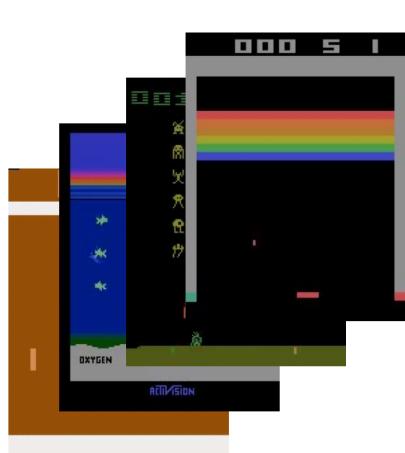


图 19.16: Atari 游戏。

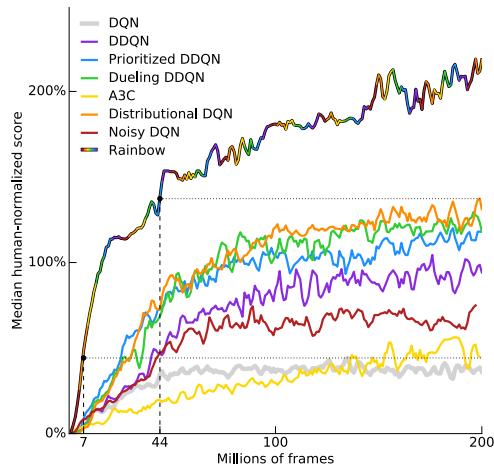


图 19.17: 使用多种技巧训练 DQN 玩 Atari 游戏。图片来自于论文 [49]。

再举几个例子。AlphaGo Zero [100] 用了 2 千 9 百万局自我博弈，每一局约有 100 个状态和动作。TD3 算法 [42] 在 MuJoCo 物理仿真环境中训练 Half-Cheetah、Ant、Hopper 等模拟机器人，虽然只有几个关节需要控制，但是在样本数量 100 万时尚未收敛。甚至连 Pendulum、Reacher 这种只有一两个关节的最简单的控制问题，TD3 也需要超过 10 万个样本。

现实世界中的问题远远比 Atari、MuJoCo 复杂，其状态空间、动作空间都远大于 Atari、MuJoCo。哪怕是现代的电子游戏，其复杂度也远大于上述的简单问题。对于简单的问题，强化学习尚需要百万、千万级的样本；那么对于现实世界中复杂的问题，强化学习需要多少样本呢？

在电子游戏中获取上亿样本并不困难，但是在现实问题中每获取一个样本都是比较困难的。在神经网络结构搜索的例子中，每获取一个奖励，需要训练一个 CNN，从初始化到梯度算法收敛，需要一个 GPU 约一小时的计算量。物理世界的应用中获取奖励更为困难。举个例子，用机械手臂抓取一个物体至少需要几秒钟时间，那么一天只能收集一万个样本；同时用十个机械手臂，连续运转一百天，才能收集到一千万个样本，未必够训练一个深度强化学习模型。强化学习所需的样本量太大，这会限制强化学习在现实中的应用。

19.6.2 探索阶段代价太大

强化学习要求智能体与环境交互，用收集到的经验去更新策略。在交互的过程中，智能体会改变环境。在仿真、游戏的环境中，智能体对环境造成任何影响都无所谓。但是在现实世界中，智能体对环境的影响可能会造成巨大的代价。

在强化学习初始的探索阶段，策略几乎是随机的。如果是物理世界中的应用，智能体的动作难免造成很大的代价。如果应用到推荐系统中，如果上线一个随机的推荐策略，那么用户的体验会极差，很低的点击率也会给网站造成收入的损失。如果应用到自动驾驶中，随机的控制策略会导致车辆撞毁。如果应用到医疗中，随机的治疗方案会致死致残。

在物理世界的应用中，不能直接让初始的随机策略与环境交互，而应该先对策略做预训练，再在真实环境中部署。一种方法是事先准备一个数据集，用行为克隆等监督学习方法做预训练。另一种方法是搭建模拟器，在模拟器中预训练策略。比如阿里巴巴提出的“虚拟淘宝”系统 [95] 是对真实用户的模仿，用这样的模拟器预训练推荐策略。离线强化学习 (Offline RL) 是一个热门而又有价值的研究方向，建议读者阅读文献 [65]。

19.6.3 超参数的影响非常大

深度强化学习对超参数的设置极其敏感，需要很小心调参才能找到好的超参数。超参数分两种：神经网络结构超参数、算法超参数。这两类超参数的设置都严重影响实验效果。换句话说，完全相同的方法，由不同的人实现，效果会有天壤之别。

结构超参数： 神经网络结构超参数包括层的数量、宽度、激活函数，这些都对结果有很大影响。拿激活函数来说，在监督学习中，在隐层中用不同的激活函数（比如 ReLU、Leaky ReLU）对结果影响很小，因此总是用 ReLU 就可以。但是在深度强化学习中，隐层激活函数对结果的影响很大；有时 ReLU 远好于 Leaky ReLU，而有时 Leaky ReLU 远好于 ReLU [48]。由于这种不一致性，我们在实践中不得不尝试不同的激活函数。

算法超参数： 强化学习中的算法超参数很多，包括学习率、批大小 (Batch Size)、经验回放的参数、探索用的噪声。比如 Rainbow 的论文 [49] 调了超过 10 种算法超参数。

- 学习率（即梯度算法的步长）对结果的影响非常大，必须要很仔细地调。DDPG、TD3、A2C 等方法中不止有一个学习率。策略网络、价值网络、目标网络中都有各自的学习率。

- 如果用经验回放，那么还需要调几个超参数，比如回放数组的大小、经验回放的起始时间等。论文 [37] 中的实验显示回放数组的大小对结果有影响，过大或者过小的数组都不好。经验回放的起始时间需要调，比如 Rainbow 在收集到 8 万条四元组的时候开始经验回放，而标准的 DQN 则最好是在收集到 20 万条之后开始经验回放 [49]。
- 在探索阶段，DQN、DPG 等方法的动作中应当加入一定噪声。噪声的大小是需要调的超参数，它可以平衡探索 (Exploration) 和利用 (Exploitation)。除了设置初始的噪声的幅度，我们还需要设置噪声的衰减率，让噪声逐渐变小。

实验效果严重依赖于实现的好坏：上面的讨论目的在于说明超参数对结果有重大影响。对于相同的方法，不同的人会有不同的实现，比如用不同的网络结构、激活函数、训练算法、学习率、经验回放、噪声。哪怕是一些细微的区别，也会影响最终的效果。论文 [48] 使用了几个比较有名的开源代码，它们都有 TRPO 和 DDPG 方法在 Half-Cheetah 环境中的实验。论文使用了它们的默认设置，比较了实验结果，如图 19.18 所示。很显然，相同的方法，不同人的编程实现，实验效果差距巨大。

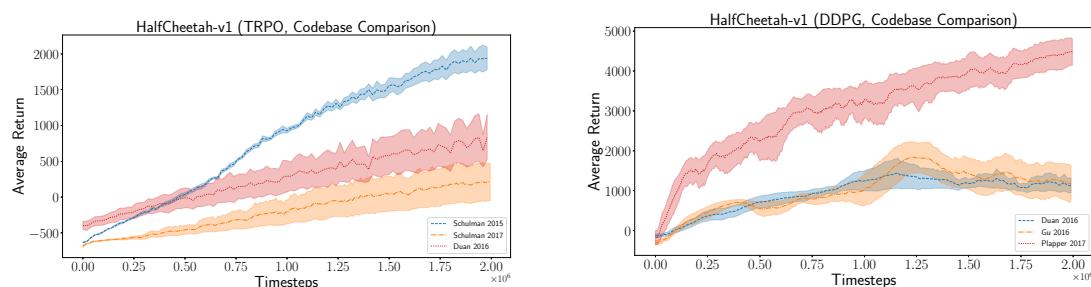


图 19.18：左图是 TRPO 的三种实现，右图是 DDPG 的三种实现。图片来自论文 [48]。

实验对比的可靠性问题：如果一篇学术论文提出一种新的方法，往往要在 Atari、MuJoCo 等标准的实验环境中做实验，并与 DQN、DDPG、TD3、A2C、TRPO 等有名的基本线做实验对照。通常只有当新的方法效果显著优于基线时，论文才有可能发表。但是论文实验中报告的结果真的可信吗？从图 19.18 中不难看出，基线算法的表现严重依赖于编程实现的好坏。如果你提出一种新的方法，你把自己的方法实现得非常好，而你从开源的实现中选一个不那么好的基线做实验对比，那么你可以轻松打败基线算法。

19.6.4 稳定性极差

强化学习训练的过程中充满了随机性。除了环境的随机性之外，随机性还来自于神经网络随机初始化、决策的随机性、经验回放的随机性。想必大家都有这样的经历：用完全相同的程序、完全相同的超参数，仅仅更改随机种子 (Random Seed)，就会导致训练的效果有天壤之别。如示意图 19.19 所示，如果重复训练十次，往往会有几次完全不收敛。哪怕是非常简单的问题，也会出现这种不收敛的情形。

在监督学习中，由于随机初始化和随机梯度中的随机性，即使用同样的超参数，训练出的模型表现也会不一致，测试准确率可能会差几个百分点。但是监督学习中几乎不

会出现图 19.19 中这种情形；如果出现了，几乎可以肯定代码中有错。但是强化学习确实会出现完全不收敛的情形，哪怕代码和超参数都是对的。

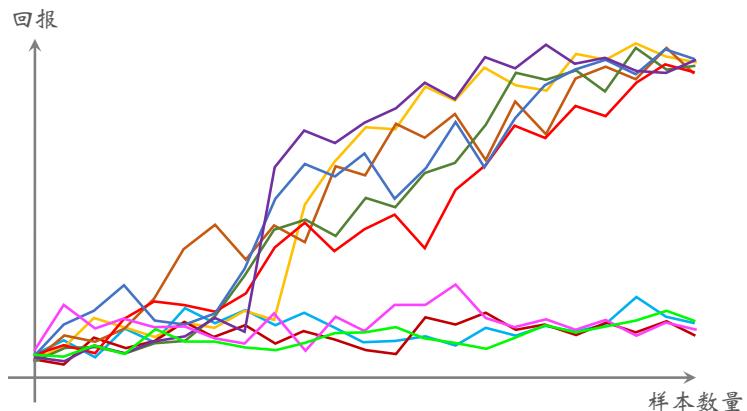


图 19.19：用完全相同的超参数，用不同的随机种子，往往回得到截然不同的收敛曲线。

附录 A 贝尔曼方程

定理 A.1. 贝尔曼方程 (将 Q_π 表示成 Q_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} [R_t + \gamma \cdot Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t].$$



证明 根据回报的定义 $U_t = \sum_{k=t}^n \gamma^{k-t} \cdot R_k$, 不难验证这个等式:

$$U_t = R_t + \gamma \cdot U_{t+1}.$$

用符号 $\mathcal{S}_{t+1:} = \{S_{t+1}, S_{t+2}, \dots\}$ 和 $\mathcal{A}_{t+1:} = \{A_{t+1}, A_{t+2}, \dots\}$ 表示从 $t+1$ 时刻起所有的状态和动作随机变量。根据动作价值函数 Q_π 的定义,

$$Q_\pi(s_t, a_t) = \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_t \mid S_t = s_t, A_t = a_t].$$

把 U_t 替换成 $R_t + \gamma \cdot U_{t+1}$, 那么

$$\begin{aligned} Q_\pi(s_t, a_t) &= \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t + \gamma \cdot U_{t+1} \mid S_t = s_t, A_t = a_t] \\ &= \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t \mid S_t = s_t, A_t = a_t] + \gamma \cdot \mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_{t+1} \mid S_t = s_t, A_t = a_t]. \end{aligned} \quad (\text{A.1})$$

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么, 给定 s_t 和 a_t , 则 R_t 随机性唯一的来源就是 S_{t+1} , 所以

$$\mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [R_t \mid S_t = s_t, A_t = a_t] = \mathbb{E}_{S_{t+1}} [R_t \mid S_t = s_t, A_t = a_t]. \quad (\text{A.2})$$

等式 (A.1) 右边 U_{t+1} 的期望可以写成

$$\begin{aligned} &\mathbb{E}_{\mathcal{S}_{t+1:}, \mathcal{A}_{t+1:}} [U_{t+1} \mid S_t = s_t, A_t = a_t] \\ &= \mathbb{E}_{S_{t+1}, A_{t+1}} \left[\mathbb{E}_{S_{t+2:}, A_{t+2:}} [U_{t+1} \mid S_{t+1}, A_{t+1}] \mid S_t = s_t, A_t = a_t \right] \\ &= \mathbb{E}_{S_{t+1}, A_{t+1}} [Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t]. \end{aligned} \quad (\text{A.3})$$

由公式 (A.1)、(A.2)、(A.3) 可得定理。 □

定理 A.2. 贝尔曼方程 (将 Q_π 表示成 V_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$Q_\pi(s_t, a_t) = \mathbb{E}_{S_{t+1}} [R_t + \gamma \cdot V_\pi(S_{t+1}) \mid S_t = s_t, A_t = a_t].$$



证明 由于 $V_\pi(S_{t+1}) = \mathbb{E}_{A_{t+1}} [Q(S_{t+1}, A_{t+1})]$, 由定理 A.1 可得定理 A.2。 □

定理 A.3. 贝尔曼方程 (将 V_π 表示成 V_π)

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

$$V_\pi(s_t) = \mathbb{E}_{A_t, S_{t+1}} [R_t + \gamma \cdot V_\pi(S_{t+1}) \mid S_t = s_t].$$



证明 由于 $V_\pi(S_t) = \mathbb{E}_{A_t} [Q(S_t, A_t)]$, 由定理 A.2 可得定理 A.3。 □

定理 A.4. 最优贝尔曼方程

假设 R_t 是 S_t 、 A_t 、 S_{t+1} 的函数。那么

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

证明 设最优策略函数为 $\pi^* = \operatorname{argmax}_\pi Q_\pi(s, a)$, $\forall s \in \mathcal{S}, a \in \mathcal{A}$ 。由贝尔曼方程可得：

$$Q_{\pi^*}(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} \left[R_t + \gamma \cdot Q_{\pi^*}(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t \right].$$

根据定义，最优动作价值函数是

$$Q_\star(s, a) \triangleq \max_{\pi} Q_\pi(s, a), \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

所以 $Q_{\pi^*}(s, a)$ 就是 $Q_\star(s, a)$ 。于是

$$Q_\star(s_t, a_t) = \mathbb{E}_{S_{t+1}, A_{t+1}} \left[R_t + \gamma \cdot Q_\star(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t \right].$$

因为动作 $A_{t+1} = \operatorname{argmax}_A Q_\star(S_{t+1}, A)$ 是状态 S_{t+1} 的确定性函数，所以

$$Q_\star(s_t, a_t) = \mathbb{E}_{S_{t+1}} \left[R_t + \gamma \cdot \max_{A \in \mathcal{A}} Q_\star(S_{t+1}, A) \mid S_t = s_t, A_t = a_t \right].$$

□

参考文献

- [1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2004.
- [2] M. S. Abdulla and S. Bhatnagar. Reinforcement learning based algorithms for average cost markov decision processes. *Discrete Event Dynamic Systems*, 17(1):23–52, 2007.
- [3] Z. Ahmed, N. Le Roux, M. Norouzi, and D. Schuurmans. Understanding the impact of entropy on policy optimization. In *International Conference on Machine Learning (ICML)*, 2019.
- [4] L. V. Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.
- [5] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.
- [6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.
- [7] M. Bain and C. Sammut. A framework for behavioural cloning. In *Machine Intelligence*, pages 103–129, 1995.
- [8] L. Baird. Residual algorithms: reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.
- [9] N. Bard, J. N. Foerster, S. Chandar, N. Burch, M. Lanctot, H. F. Song, E. Parisotto, V. Dumoulin, S. Moitra, E. Hughes, et al. The Hanabi challenge: a new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020.
- [10] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.
- [11] P. Baudiš and J.-l. Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011.
- [12] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2017.
- [13] D. P. Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.
- [14] S. Bhatnagar and S. Kumar. A simultaneous perturbation stochastic approximation-based actor-critic algorithm for markov decision processes. *IEEE Transactions on Automatic Control*, 49(4):592–598, 2004.
- [15] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee. Natural actor-critic algorithms. *Automatica*, 45(11):2471–2482, 2009.
- [16] A. Boulieras, J. Kober, and J. Peters. Relative entropy inverse reinforcement learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.
- [17] C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *Conference on Theoretical Aspects of Rationality and Knowledge*, 1996.
- [18] B. Bouzy and B. Helmstetter. Monte-Carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004.
- [19] S. Boyd, S. P. Boyd, and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [20] I. Bratko and T. Urbancic. Transfer of control skill by machine learning. *Engineering Applications of Artificial Intelligence*, 10(1):63–71, 1997.
- [21] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.
- [22] M. Buro. From simple features to sophisticated evaluation functions. In *International Conference on Computers and Games*, pages 126–145. Springer, 1998.
- [23] L. Buşoniu, R. Babuška, and B. De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations*

- in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.
- [24] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.
- [25] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo tree search: A new framework for game AI. In *AIIDE*, 2008.
- [26] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. J. Van Den Herik. Monte-Carlo strategies for computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, 2006.
- [27] G. M. J.-B. C. Chaslot. *Monte-Carlo tree search*. Maastricht University, 2010.
- [28] M. Chen, A. Beutel, P. Covington, S. Jain, F. Belletti, and E. H. Chi. Top-k off-policy correction for a REINFORCE recommender system. In *ACM International Conference on Web Search and Data Mining*, 2019.
- [29] K. Cho, B. v. M. C. Gulcehre, D. Bahdanau, F. B. H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. 2014.
- [30] Y. Chow, O. Nachum, and M. Ghavamzadeh. Path consistency learning in Tsallis entropy regularized mdps. In *International Conference on Machine Learning (ICML)*, pages 979–988, 2018.
- [31] A. R. Conn, N. I. Gould, and P. L. Toint. *Trust region methods*. SIAM, 2000.
- [32] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.
- [33] R. Coulom. Computing “elo ratings” of move patterns in the game of Go. *ICGA journal*, 30(4):198–208, 2007.
- [34] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [35] T. Degris, P. M. Pilarski, and R. S. Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference (ACC)*, 2012.
- [36] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego: an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.
- [37] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning (ICML)*, 2020.
- [38] C. Finn, S. Levine, and P. Abbeel. Guided cost learning: deep inverse optimal control via policy optimization. In *International Conference on Machine Learning (ICML)*, 2016.
- [39] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual multi-agent policy gradients. In *AAAI Conference on Artificial Intelligence*, 2018.
- [40] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. Torr, P. Kohli, and S. Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2017.
- [41] M. Fortunato, M. G. Azar, B. Piot, J. Menick, M. Hessel, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, et al. Noisy networks for exploration. In *International Conference on Learning Representations (ICLR)*, 2018.
- [42] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning (ICML)*, 2018.
- [43] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in Neural Information Processing Systems (NIPS)*, 2014.
- [44] J. K. Gupta, M. Egorov, and M. Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, 2017.
- [45] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning (ICML)*, 2017.
- [46] R. Hafner and M. Riedmiller. Reinforcement learning in feedback control. *Machine learning*, 84(1-2):137–169, 2011.
- [47] M. Hausknecht and P. Stone. Deep recurrent Q-learning for partially observable MDPs. In *AAAI Fall*

- Symposium on Sequential Decision Making for Intelligent Agents, 2015.
- [48] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *AAAI Conference on Artificial Intelligence*, 2018.
 - [49] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
 - [50] J. Ho and S. Ermon. Generative adversarial imitation learning. In *Conference on Neural Information Processing Systems (NIPS)*, 2016.
 - [51] Y.-C. Ho. Team decision theory and information structures. *Proceedings of the IEEE*, 68(6):644–654, 1980.
 - [52] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
 - [53] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.
 - [54] J. Hu and M. P. Wellman. Nash Q-learning for general-sum stochastic games. *Journal of Machine Learning Research*, 4(Nov):1039–1069, 2003.
 - [55] Y. Hu, Q. Da, A. Zeng, Y. Yu, and Y. Xu. Reinforcement learning to rank in e-commerce search engine: Formalization, analysis, and application. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.
 - [56] S. Iqbal and F. Sha. Actor-attention-critic for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2019.
 - [57] T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural computation*, 6(6):1185–1201, 1994.
 - [58] Y. Keneshloo, T. Shi, N. Ramakrishnan, and C. K. Reddy. Deep reinforcement learning for sequence-to-sequence models. *IEEE transactions on neural networks and learning systems*, 31(7):2469–2489, 2019.
 - [59] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.
 - [60] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, 2000.
 - [61] M. G. Lagoudakis and R. Parr. Learning in zero-sum team Markov games using factored value functions. *Advances in Neural Information Processing Systems (NIPS)*, 2002.
 - [62] M. Lauer and M. Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *International Conference on Machine Learning (ICML)*, 2000.
 - [63] K. Lee, S. Choi, and S. Oh. Sparse Markov decision processes with causal sparse Tsallis entropy regularization for reinforcement learning. *IEEE Robotics and Automation Letters*, 3(3):1466–1473, 2018.
 - [64] S. Levine and V. Koltun. Continuous inverse optimal control with locally optimal examples. In *International Conference on Machine Learning (ICML)*, 2012.
 - [65] S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.
 - [66] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
 - [67] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2016.
 - [68] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.
 - [69] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 1994.
 - [70] M. L. Littman. Friend-or-foe Q-learning in general-sum games. In *International Conference on Machine*

- Learning (ICML)*, 2001.
- [71] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2018.
- [72] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
- [73] P. Marbach and J. N. Tsitsiklis. Simulation-based optimization of Markov reward processes: Implementation issues. In *IEEE Conference on Decision and Control*, 1999.
- [74] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.
- [75] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2016.
- [76] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [77] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
- [78] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: a distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.
- [79] M. Müller. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002.
- [80] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.
- [81] A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2000.
- [82] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.
- [83] B. O’Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih. Combining policy gradient and Q-learning. In *International Conference on Learning Representations (ICLR)*, 2017.
- [84] F. A. Oliehoek, M. T. Spaan, and N. Vlassis. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32:289–353, 2008.
- [85] D. V. Prokhorov and D. C. Wunsch. Adaptive critic designs. *IEEE transactions on Neural Networks*, 8(5):997–1007, 1997.
- [86] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.
- [87] S. Ross and D. Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.
- [88] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.
- [89] M. Samvelyan, T. Rashid, C. Schroeder de Witt, G. Farquhar, N. Nardelli, T. G. Rudner, C.-M. Hung, P. H. Torr, J. Foerster, and S. Whiteson. The StarCraft Multi-Agent Challenge. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2019.
- [90] S. Schaal. Learning from demonstration. In *Advances in Neural Information Processing Systems (NIPS)*, 1997.
- [91] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.
- [92] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers

- program. *Artificial Intelligence*, 53(2-3):273–289, 1992.
- [93] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*, 2015.
- [94] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning (ICML)*, 2015.
- [95] J.-C. Shi, Y. Yu, Q. Da, S.-Y. Chen, and A.-X. Zeng. Virtual-Taobao: virtualizing real-world online retail environment for reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2019.
- [96] W. Shi, S. Song, and C. Wu. Soft policy gradient method for maximum entropy deep reinforcement learning. *arXiv preprint arXiv:1909.03198*, 2019.
- [97] Y. Shoham and K. Leyton-Brown. *Multiagent systems: algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.
- [98] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [99] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning (ICML)*, 2014.
- [100] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.
- [101] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.
- [102] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2018.
- [103] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems (NIPS)*, 1996.
- [104] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [105] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, 2000.
- [106] U. Syed, M. Bowling, and R. E. Schapire. Apprenticeship learning using linear programming. In *International Conference on Machine Learning (ICML)*, 2008.
- [107] U. Syed and R. E. Schapire. A reduction from apprenticeship learning to classification. *Advances in Neural Information Processing Systems (NIPS)*, 23, 2010.
- [108] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4):e0172395, 2017.
- [109] M. Tan. Multi-agent reinforcement learning: independent vs. cooperative agents. In *International Conference on Machine Learning (ICML)*, 1993.
- [110] X. Tang, Z. Qin, F. Zhang, Z. Wang, Z. Xu, Y. Ma, H. Zhu, and J. Ye. A deep value-network based approach for multi-driver order dispatching. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.
- [111] G. Tesauro and G. R. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems*, pages 1068–1074, 1997.
- [112] C. Tsallis. Possible generalization of Boltzmann-Gibbs statistics. *Journal of statistical physics*, 52(1-2):479–487, 1988.
- [113] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine learning*, 16(3):185–202, 1994.
- [114] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.

- [115] H. J. Van Den Herik, J. W. Uiterwijk, and J. Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.
- [116] H. van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2010.
- [117] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
- [118] H. van Seijen. Effective multi-step temporal-difference learning for non-linear function approximation. *arXiv preprint arXiv:1608.05151*, 2016.
- [119] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.
- [120] N. Vlassis. A concise introduction to multiagent systems and distributed artificial intelligence. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 1(1):1–71, 2007.
- [121] X. Wang and T. Sandholm. Reinforcement learning to play an optimal Nash equilibrium in team Markov games. *Advances in Neural Information Processing Systems (NIPS)*, 2002.
- [122] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2016.
- [123] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [124] C. J. C. H. Watkins. Learning from delayed rewards. 1989.
- [125] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.
- [126] R. J. Williams. *Reinforcement-learning connectionist systems*. College of Computer Science, Northeastern University, 1987.
- [127] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- [128] R. J. Williams and J. Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.
- [129] W. Yang, X. Li, and Z. Zhang. A regularized approach to sparse optimal policy in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 5940–5950, 2019.
- [130] Z. Yang, Y. Chen, M. Hong, and Z. Wang. Provably global convergence of actor-critic: A case for linear quadratic regulator with ergodic cost. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 8353–8365, 2019.
- [131] T. Yoshikawa. Decomposition of dynamic team decision problems. *IEEE Transactions on Automatic Control*, 23(4):627–632, 1978.
- [132] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
- [133] K. Zhang, Z. Yang, and T. Basar. Multi-agent reinforcement learning: a selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*, 2019.
- [134] X. Zhao, L. Zhang, L. Xia, Z. Ding, D. Yin, and J. Tang. Deep reinforcement learning for list-wise recommendations. *arXiv preprint arXiv:1801.00209*, 2017.
- [135] V. Zhong, C. Xiong, and R. Socher. SEQ2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.
- [136] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 2008.
- [137] B. Zoph and Q. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

致谢

由于本书篇幅较长，难免出现错误。真诚感谢王嘉晨、张梦娇、陈传玺、常海德、张翠娟、梅椰诚、张大康、单思远、陆浩、徐嘉诚、汪天祥、贺晨龙、邹笑寒、石金升、李凯、陈刚、钱超、杨典、新代、谢宇航提供的反馈与批评指正。