深度学习基础

在教程前面部分我们介绍了强化学习的基础内容，包含马尔可夫决策过程、预测与控制等，其中经典的预测与控制算法主要包括动态规划、蒙特卡洛方法和时序差分方法等。

这些算法在解决一些简单的强化学习问题时表现良好，但在面对高维度和复杂环境时，传统方法往往力不从心。为了解决这些问题，深度强化学习应运而生，它结合了深度学习的强大功能，使得强化学习能够处理更复杂的任务。

价值函数表格

在强化学习中，价值函数（如状态价值函数 $V(s)$ 和动作价值函数 $Q(s,a)$）是评估状态或状态-动作对好坏的关键工具，怎么表示这些函数是强化学习中的一个重要问题。

对于小规模的问题，例如前面章节中介绍的网格世界（Grid World）示例，我们可以使用表格形式来存储对应的状态或动作价值。如表 1 所示，我们可以使用一个二维表格来表示动作价值函数 $Q(s,a)$，其中行表示不同的动作 a，列表示不同的状态 s。表格中的每个元素 $Q(s,a)$ 存储了在状态 s 下采取动作 a 所获得的预期回报。

表 1: Q 表格

	s_1	s_2	s_3	s_4
a_1	-1.9	-1.0	-1.0	0.0
a_2	-1.3	-0.9	-0.7	-0.5

表格实际上是一种数学抽象形式，在程序实现上可以使用 `Python` 数组或者字典来表示，如代码 1 所示。

代码 1: 表格的程序实现示例

```python
import numpy as np
# 使用二维数组表示 Q 表格
# 维度为 (状态数, 动作数)，例如 Q[0, 1] 表示状态 s1 下采取动作 a2 的价值
Q_table = np.array([[-1.9, -1.0, -1.0, 0.0],
                    [-1.3, -0.9, -0.7, -0.5]])
# 使用字典表示 Q 表格
# 键为 (状态, 动作) 的元组，值为对应的动作价值，例如 Q_dict[(0, 1)] 表示状态 s1 下采取动作 a2 的价值
Q_dict = {
    (0, 0): -1.9, (0, 1): -1.0, (0, 2): -1.0, (0, 3): 0.0,
    (1, 0): -1.3, (1, 1): -0.9, (1, 2): -0.7, (1, 3): -0.5
}
# 为便于理解可以使用str元组表示状态和动作
Q_dict_str = {
    ('s_1', 'a_1'): -1.9, ('s_2', 'a_1'): -1.0,
    ('s_3', 'a_1'): -1.0, ('s_4', 'a_1'): 0.0,
    ('s_1', 'a_2'): -1.3, ('s_2', 'a_2'): -0.9,
    ('s_3', 'a_2'): -0.7, ('s_4', 'a_2'): -0.5
}
```

然而，随着问题规模的增大，状态和动作空间变得庞大甚至连续，使用表格来存储价值函数变得不可行。一方面，表格的存储内存会随着状态和动作数量的增加而指数级增长，导致计算开销过大。另一方面，表格方法无法处理连续状态或动作空间，因为无法为每一个可能的状态-动作对分配一个独立的表格项。

函数近似表示

为了解决上述问题，我们可以使用函数近似的方法来表示价值函数。函数近似通过参数化的函数（如线性函数、神经网络等）来估计价值函数，从而避免了存储整个表格的需求。

在深度学习出现之前，线性函数是常用的函数近似方法。线性函数通过将状态和动作映射到一个特征空间，并使用线性组合来估计价值函数。例如，动作价值函数 $Q(s,a)$ 可以表示式 (1) 。

$$Q(s,a;\boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s,a) \tag{1}$$

其中 $Q(s,a;\boldsymbol{\theta})$ 有时也协作 $Q_{\boldsymbol{\theta}}(s,a)$，$\boldsymbol{\phi}(s,a)$ 是状态-动作对的特征向量，$\boldsymbol{\theta}$ 是参数向量。通过调整参数 $\boldsymbol{\theta}$，我们可以使得函数近似更好地拟合实际的价值函数。

但是，线性函数在处理复杂的非线性关系时表现有限，参数调整也较为困难，这限制了其在复杂强化学习任务中的应用。

除了线性函数，其他传统的函数近似方法还包括决策树、支持向量机等，这些方法在某些特定任务中也有一定的应用价值，具体总结如表 2 所示。

表 2: 典型函数近似方法总结

方法类型	核心思想	表达能力	优缺点	示例算法
线性近似	$V(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$	弱	高效但不能表达非线性	Linear TD, LSTD
RBF 网络	高斯核函数组合	中等	可表示局部非线性，计算较重	RBF-Q
决策树/GBDT	树状划分状态空间	中高	可解释、难与在线RL结合	Fitted Q-Iteration
局部加权回归	局部样本加权平均	中高	局部准确，全局难泛化	LWPR
Tile Coding	分层网格离散化	中	稀疏更新，高维困难	Sarsa(λ) with Tile
神经网络	层级非线性映射	很强	表达力最强，但有时也不稳定	DQN, PPO, SAC

从表中可以看出，神经网络在表达能力上最强，能够处理复杂的非线性关系，因此，现在更**常用的是神经网络作为函数近似器**，它能够捕捉复杂的非线性关系，并且通过反向传播算法进行高效的参数更新，具体内容将在下文展开讲解。

梯度下降

相比于表格方法，函数近似具有更好的泛化能力，但同时也引入额外的参数，因此在训练中除了考虑价值函数预测的准确性（或者说最小化预测误差）之外，还需要考虑参数的优化问题。

参数优化通常使用梯度下降（Gradient Descent）方法，通过计算损失函数关于参数的梯度，并沿着梯度的反方向更新参数，从而逐步逼近最优解。具体来说，假设我们有一个损失函数 $L(\boldsymbol{\theta})$，表示预测值与真实值之间的差异，那么参数更新的公式可以表示为式 (2) 。

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{2}$$

其中 α 是学习率，控制参数更新的步长，$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ 是损失函数关于参数的梯度。

注意，梯度下降的目标是最小化损失函数，有时可能需要最大化某个目标函数（例如累积奖励），此时可使用梯度上升（Gradient Ascent）方法，如式 (3) 所示。

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{3}$$

但在实际应用中，梯度上升可以通过优化负的损失函数来实现，即式 (4) 。

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} (-J(\boldsymbol{\theta})) \tag{4}$$

换句话说，梯度下降和梯度上升在数学上是很容易互相转换的，出于统一性和习惯性考虑，通常都会把问题转化为最小化损失函数的问题，然后使用梯度下降方法来进行参数优化。

在标准的梯度下降基础上，一些改进的优化算法，如随机梯度下降（SGD）、动量法（Momentum）、自适应学习率方法（如 Adam）等，这些方法在实际应用中能够提高收敛速度和稳定性。

除了梯度下降方法之外，还有其他优化方法，如牛顿法、拟牛顿法等，**这些方法通过利用二阶导数信息来加速收敛，但计算复杂度较高，通常在大规模强化学习中不常用**，具体总结如表 3 所示。

表 2: 典型参数优化方法总结

方法类型	核心思想	优点	缺点	常见算法
梯度下降	按误差方向更新参数	简单、通用	收敛慢、需调学习率	TD(λ), SARSA
最小二乘法	一次性解析求解	快速收敛	不适合在线	Monte Carlo Fit
LSTD	TD + 最小二乘	稳定、无学习率	需计算矩阵逆	LSTD(λ), LSPI
RLS	在线更新最小二乘	快、稳定	计算略复杂	Adaptive TD
共轭梯度	近似解线性系统	快、节省内存	需矩阵操作	LSTD-CG
岭回归	带正则的最小二乘	稳定性高	λ需调	L2-TD
贝叶斯线性回归	估计参数分布	不确定性估计	计算复杂	Bayesian TD

在强化学习中，以状态价值为例，将这里的 $V(s)$ 用线性函数近似来替换原来的表格表示，如式 (5) 所示。

$$V_\theta(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s) \tag{5}$$

其中 $\boldsymbol{\phi}(s)$ 是状态 s 的特征向量，$\boldsymbol{\theta}$ 是参数向量。由于是线性函数，因此对应的梯度非常简单，如式 (???) 所示。

$$\nabla_{\boldsymbol{\theta}} V_\theta(s) = \boldsymbol{\phi}(s) \tag{6}$$

如何更新参数呢？可以通过最小化函数近似的状态价值与真实价值之间的均方误差来实现，如式 (7) 所示。

$$L(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}\left[\left(V^\pi(s) - V_\theta(s) \right)^2 \right] \tag{7}$$

这里 $V^\pi(s)$ 是在策略 π 下状态 s 的真实价值，$V_\theta(s)$ 是函数近似的估计值，加上 $\frac{1}{2}$ 是为了在计算梯度时方便抵消平方项的系数 2。其中真实价值 $V^\pi(s)$ 通常是未知的，因此我们需要使用采样得到的目标值来替代它。

具体来说，若使用蒙特卡洛估计，可以将目标值设为完整回合的累积奖励 $V^\pi(s_t) \approx G_t$，结合复合函数求导法则，得到损失函数的梯度为式 (8) 。

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \mathbb{E}\left[(G_t - V_\theta(s_t))(-\nabla_{\boldsymbol{\theta}} V_\theta(s_t)) \right] \tag{8}$$

注意这里目标值 G_t 不依赖于参数 $\boldsymbol{\theta}$，因此梯度只包含一个部分，对应的梯度下降更新公式为式 (9) 。

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha\left[G_t - V_\theta(s_t)\right](-\nabla_{\boldsymbol{\theta}} V_\theta(s_t)) \\ &= \boldsymbol{\theta} - \alpha\left[V_\theta(s_t) - G_t\right]\nabla_{\boldsymbol{\theta}} V_\theta(s_t) \\ &= \boldsymbol{\theta} - \alpha\left[V_\theta(s_t) - G_t\right]\boldsymbol{\phi}(s_t)\end{aligned} \tag{9}$$

若使用时序差分估计，可以将目标值设为单步奖励加上下一个状态的估计价值 $V^\pi(s_t) \approx R_{t+1} + \gamma V_\theta(s_{t+1})$，则梯度如式 (10) 所示。

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \mathbb{E}\left[(R_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t))(\gamma \nabla_{\boldsymbol{\theta}} V_\theta(s_{t+1}) - \nabla_{\boldsymbol{\theta}} V_\theta(s_t))\right] \tag{10}$$

注意，这里目标值 $R_{t+1} + \gamma V_\theta(s_{t+1})$ 也依赖于参数 $\boldsymbol{\theta}$，因此梯度包含了两个部分。但是由于实际应用中通常忽略目标值对参数的依赖，即半梯度更新，对应的梯度下降更新公式为式 (11) 。

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha\left[V_\theta(s_t) - (R_{t+1} + \gamma V_\theta(s_{t+1}))\right]\nabla_{\boldsymbol{\theta}} V_\theta(s_t) \\ &= \boldsymbol{\theta} - \alpha\left[V_\theta(s_t) - R_{t+1} + \gamma V_\theta(s_{t+1})\right]\boldsymbol{\phi}(s_t)\end{aligned} \tag{11}$$

半梯度更新 $\mathrm{TD}(0)$ 方法虽然忽略了目标值对参数的依赖，但这样的做法已经被证明可以收敛到一个较好的解，且计算更为简单。此外如果直接使用完整梯度，反而会破坏时序差分的递推结果，从而导致不稳定。

梯度下降示例

本节将继续以前面章节中的 3×3 网格世界为例，演示如何使用线性函数近似和梯度下降，并结合时序差分方法来估计状态价值函数 $V(s)$。

先回顾一下网格世界的环境设置，如图 1 所示。考虑智能体在 3×3 的网格中使用随机策略进行移动，以左上角为起点，右下角为终点，同样规定每次只能向右或向下移动，动作分别用 a_1 和 a_2 表示。用智能体的位置不同的状态，即 $s_1, s_2, \ldots, s_9$，初始状态为$S_0 = s_1$，终止状态为s_9。

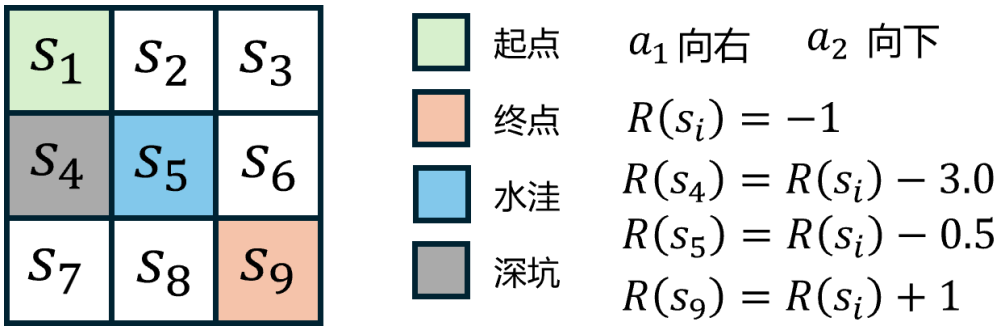

图 1: 3x3 网格示例

除了每走一步接收 -1 的奖励之外，这次我们在网格中增加了一些障碍物，例如在位置 s_4 处设置了一个深坑，智能体走到该位置时会受到一个额外的负奖励 -3，在位置 s_5 处设置了一个水洼，智能体走到该位置时会受到一个额外的负奖励 -0.5。折扣因子 $\gamma = 0.9$，目标是计算各个状态的价值函数 $V(s)$。

在这个网格示例中，我们可以设计一个简单的特征映射 $\boldsymbol{\phi}(s)$，将状态 s 映射为一个包含位置坐标和障碍物信息的特征向量。如何定义这个特征向量的过程叫作**特征工程**，**设计特征时需要考虑特征的表达能力和计算复杂度之间的平衡**，往往也会需要一些编码技巧。

在本例中，状态 s 的特征向量可以设计如表 4 所示。
，状态 s 的特征向量可以设计如表 4 所示。

表 4: 状态特征映射示例

状态 s	行坐标	列坐标	行列乘积	行坐标平方	列坐标平方	是否深坑	是否水洼
s_1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
s_2	0.0	0.5	0.0	0.0	0.25	0.0	0.0
s_3	0.0	1.0	0.0	0.0	1.0	0.0	0.0
s_4	0.5	0.0	0.0	0.25	0.0	1.0	0.0
s_5	0.5	0.5	0.25	0.25	0.25	0.0	1.0
s_6	0.5	1.0	0.5	0.25	1.0	0.0	0.0
s_7	1.0	0.0	0.0	1.0	0.0	0.0	0.0
s_8	1.0	0.5	0.5	1.0	0.25	0.0	0.0
s_9	1.0	1.0	1.0	1.0	1.0	0.0	0.0
特征表示	r	c	$\mathrm{r} \times \mathrm{c}$	r^2	c^2	$\mathrm{is\_pit}$	$\mathrm{is\_puddle}$
权重表示	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7

注意行列坐标为了了避免数值过大，均做了归一化处理，取值范围为 $[0, 1]$。此外，还引入了两个二值特征，分别表示当前状态是否为深坑（$\mathrm{is\_pit}$）或水洼（$\mathrm{is\_puddle}$）。

然后再加上偏置项，总共八个特征。这样，状态 s 的价值函数估计 (5) 可以展开如式 (12) 所示。

$$V_\theta(s) = \theta_0 + \theta_1 \cdot \mathrm{r} + \theta_2 \cdot \mathrm{c} + \theta_3 \cdot (\mathrm{r} \times \mathrm{c}) + \theta_4 \cdot \mathrm{r}^2 + \theta_5 \cdot \mathrm{c}^2 + \theta_6 \cdot \mathrm{is\_pit} + \theta_7 \cdot \mathrm{is\_puddle} \tag{12}$$

基于上述设置，我们可以实现一个简单的半梯度 $\mathrm{TD}(0)$ 算法来估计状态价值函数 $V(s)$，具体如代码 2 所示。

代码 2: 使用线性函数近似和半梯度 TD(0) 估计状态价值函数

```
import numpy as np
import pandas as pd
import random

# ----- MDP setup -----
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2),
}

def legal_actions(s):
    r,c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
```

```
    return acts

def step(s, a):
    r,c = coords[s]
    if a == "right": r2,c2 = r, c+1
    else:            r2,c2 = r+1, c
    # next state
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2=="s4": reward -= 3.0      # pit
    if s2=="s5": reward -= 0.5      # puddle
    if s2=="s9": reward += 1.0      # terminal bonus -> net 0
    done = (s2==terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----- feature map -----
def phi(s):
    r, c = coords[s]
    rn, cn = r/2.0, c/2.0
    is_pit = 1.0 if s=="s4" else 0.0
    is_puddle = 1.0 if s=="s5" else 0.0
    return np.array([1.0, rn, cn, rn*cn, rn*rn, cn*cn, is_pit, is_puddle], dtype=float)

# ----- semi-gradient TD(0) -----
def td0_linear_value(episodes=20000, alpha=0.05, gamma=0.9):
    w = np.zeros(len(phi("s1")), dtype=float)
    for _ in range(episodes):
        s = "s1"
        while s != terminal:
            a = random_policy(s)
            s2, r, done = step(s, a)
            v_s  = float(w @ phi(s))
            v_s2 = 0.0 if done else float(w @ phi(s2))
            td_error = r + gamma*v_s2 - v_s
            w += alpha * td_error * phi(s)
            s = s2
    return w

w = td0_linear_value()

def V_hat(s): return float(w @ phi(s)) if s!="s9" else 0.0
grid = np.array([[V_hat(f"s{r*3+c+1}") for c in range(3)] for r in range(3)])
print(pd.DataFrame(np.round(grid,3),
                   index=["row1","row2","row3"],
                   columns=["col1","col2","col3"]))
print("\nweights:", np.round(w,3))
```

执行代码后，可以得到结果如代码 3 所示。**注意由于随机性的存在，每次运行结果会有些许差异，但整体趋势是一致的。这个现象在所有的迭代更新中都存在，尤其是在使用随机梯度下降方法时更为明显。**

代码 3: 线性函数近似估计的状态价值函数结果

```
       col1   col2   col3
row1 -4.456 -2.148 -0.906
row2 -2.030 -0.866  0.117
row3 -0.875  0.185  0.000

weights: [-4.456  5.501  5.679 -2.494 -1.921 -2.13   0.155 -0.364]
```

可以看到，状态价值函数 $V(s)$ 的估计结果与之前使用表格方法得到的结果较为接近，说明线性函数近似结合梯度下降和时序差分方法在这个示例问题中能够有效地估计状态价值函数。

独热编码

前面在示例中，我们使用了手工设计的特征映射 $\boldsymbol{\phi}(s)$ 来表示状态 s，这种方法在状态空间较小且结构明确的情况下效果较好，但在更复杂的环境中，手工设计特征可能变得困难且不够灵活，因此需要更通用的状态表示方法。

注意到，状态 s 在这个示例中是离散的，是不是可以直接用整数来表示状态呢？例如用 $1, 2, 3, \cdots$ 来分别表示状态 $s_1, s_2, s_3, \cdots$。虽然这样做简单直接，但可能会导致神经网络难以学习到有效的特征。因为这些状态之间并没有实际的数值关系，更多的是不同的类别或标签，直接使用整数表示可能会导致神经网络难以学习到有效的特征。

对于离散的状态，为了兼顾表达的通用性和神经网络的学习难度，我们可以使用独热编码（One-Hot Encoding）来表示状态 s。独热编码将每个离散状态映射为一个高维向量，其中只有对应状态的位置为 1，其他位置为 0。例如，在 3×3 网格示例中，假设有九个离散状态 $s_1, s_2, \ldots, s_9$，它们的独热编码表示如式 (13) 所示。

$$
\begin{aligned}
s_1 &: [1,0,0,0,0,0,0,0,0] \\
s_2 &: [0,1,0,0,0,0,0,0,0] \\
s_3 &: [0,0,1,0,0,0,0,0,0] \\
s_4 &: [0,0,0,1,0,0,0,0,0] \\
s_5 &: [0,0,0,0,1,0,0,0,0] \\
s_6 &: [0,0,0,0,0,1,0,0,0] \\
s_7 &: [0,0,0,0,0,0,1,0,0] \\
s_8 &: [0,0,0,0,0,0,0,1,0] \\
s_9 &: [0,0,0,0,0,0,0,0,1]
\end{aligned}
\tag{13}
$$

对应的函数表达式为式 (14) 。

$$
\boldsymbol{\phi}(s_i) = [0, 0, \ldots, 1, \ldots, 0]^T \in \mathbb{R}^n, \quad (\text{第 i 个位置为 } 1，\text{其余为 } 0) \tag{14}
$$

其中 n 是状态的总数。使用独热编码后，函数近似的状态价值函数可以表示为式 (15) 。

$$
V_\theta(s_i) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s_i) = \theta_i \tag{15}
$$

也就是说，使用独热编码时，线性函数近似实际上等价于表格方法，每个状态对应一个独立的参数 θ_i，从而实现对每个状态价值的单独估计。

基于独热编码，我们同样可以实现一个简单的半梯度 $\mathrm{TD}(0)$ 算法来估计状态价值函数 $V(s)$，具体如代码 4 所示。

代码 4: 使用独热编码和半梯度 TD(0) 估计状态价值函数

```
import numpy as np
import random
```

```
import pandas as pd

# ---------- 环境 ----------
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
    "s7": (2,0), "s8": (2,1), "s9": (2,2),
}

def legal_actions(s):
    r,c = coords[s]
    acts=[]
    if c<2: acts.append("right")
    if r<2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a=="right": r2,c2 = r, c+1
    else:          r2,c2 = r+1, c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2=="s4": reward -= 3.0     # 深坑
    if s2=="s5": reward -= 0.5     # 水洼
    if s2=="s9": reward += 1.0     # 终点净0
    done = (s2==terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ---------- One-hot 特征 ----------
def phi(s):
    vec = np.zeros(9)
    vec[int(s[1:]) - 1] = 1.0
    return vec

d = 9                       # 参数维度
theta = np.zeros(d)         # 线性权重
alpha = 0.1
episodes = 20000

def v_hat(s):
    return np.dot(theta, phi(s))

# ---------- TD(0) 半梯度 ----------
for ep in range(episodes):
    s = "s1"
    while s != terminal:
        a = random_policy(s)
```

```
        s2, r, done = step(s, a)
        target = r + (0 if done else gamma * v_hat(s2))
        delta = target - v_hat(s)
        theta += alpha * delta * phi(s)   # 线性半梯度更新
        s = s2

# ---------- 输出 ----------
V_est = {s: (0.0 if s==terminal else v_hat(s)) for s in states}
grid = np.array([[V_est[f"s{r*3+c+1}"] for c in range(3)] for r in range(3)])
df = pd.DataFrame(np.round(grid,3), index=["row1","row2","row3"], columns=
["col1","col2","col3"])

print("线性函数近似 (one-hot) + TD(0) 学到的状态价值：")
print(df)

print("\n参数向量 θ（对应每个状态的估计值）:")
print(pd.Series(np.round(theta,3), index=states))
```

执行代码后，参考结果如代码 5 所示。

代码 5: 独热编码估计的状态价值函数结果

```
线性函数近似 (one-hot) + TD(0) 学到的状态价值：
       col1  col2  col3
row1 -4.445 -2.09  -1.0
row2 -2.128 -1.00   0.0
row3 -1.000  0.00   0.0

参数向量 θ（对应每个状态的估计值）:
s1   -4.445
s2   -2.090
s3   -1.000
s4   -2.128
s5   -1.000
s6    0.000
s7   -1.000
s8    0.000
s9    0.000
dtype: float64
```

可以看到，得到的结果跟之前的状态价值估计方法是接近的，说明独热编码作为一种通用的状态表示方法，能够有效地支持线性函数近似和梯度下降方法来估计状态价值函数。

然而当状态空间变得更大或更复杂时，独热编码的维度也会随之增加，导致计算和存储开销变大，并且无法捕捉状态之间的相似性和结构信息。因此，在更复杂的环境中，需要不同的编码方式，例如嵌入式表示（embedding）等，然后再输入到函数近似模型或神经网络中进行处理，这样可以更有效地利用状态之间的关系和特征。

神经网络近似

前面讲到，线性函数近似或者拟合价值函数在某些简单问题中表现良好，但在面对复杂的环境和高维状态空间时，线性函数的表达能力有限，难以捕捉复杂的非线性关系。

随着深度学习的发展，神经网络成为了更强大的函数近似工具。神经网络通过多层非线性变换，能够捕捉复杂的模式和关系，从而更准确地估计价值函数，如式 (7) 所示。

$$Q_\theta(s,a) = \mathrm{NN}(s,a;\boldsymbol{\theta}) \tag{16}$$

神经网络一般包含三个主要部分：输入层、隐藏层和输出层，其中隐藏层可以有多层，每层包含多个神经元，通过激活函数实现非线性变换。如图 2 所示，假设我们使用一个简单的前馈神经网络来近似动作价值函数 $Q(s,a)$。输入层包含状态 s （有时也包含动作 a ）的特征表示，经过隐藏层的非线性变换，最终输出对应的动作价值估计。

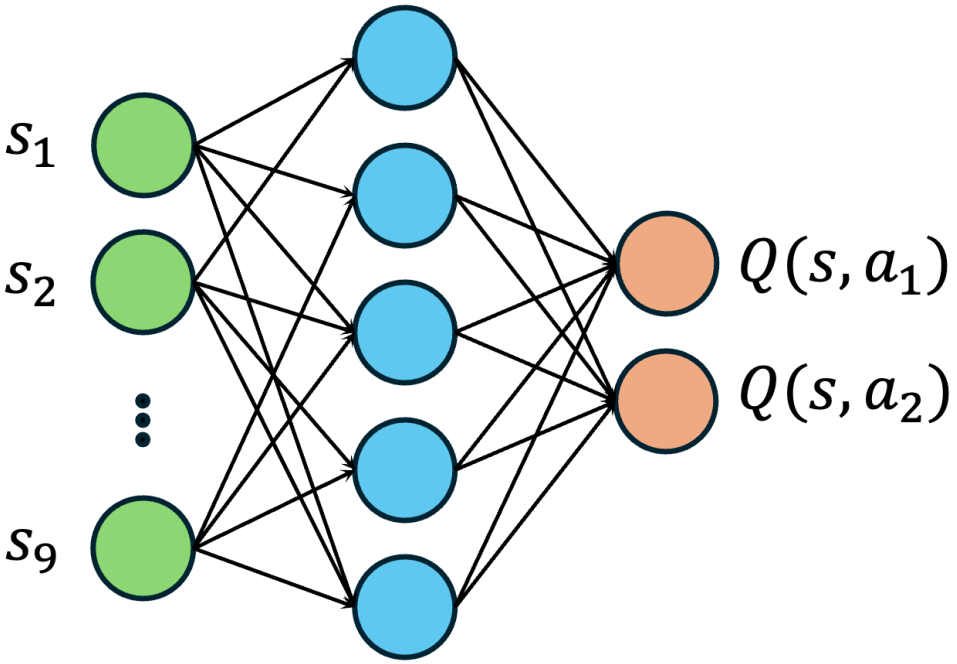

图 2: 使用神经网络近似动作价值函数 $Q(s,a)$

对于线性输入，即一维特征向量，一般可以用全连接层（fully connected layer，简称 FC）将输入映射到隐藏层，然后通过激活函数（如 ReLU、Sigmoid 等）引入非线性，最后再通过输出层得到价值估计。对应的图示可以简化为图 3 所示。

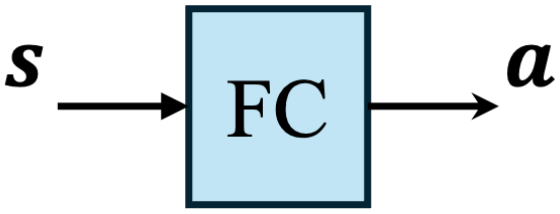

图 3: 神经网络全连接层示意图

再回到前面的 3×3 网格示例，我们可以使用神经网络来近似状态价值函数 $V(s)$，并结合半梯度 TD(0) 方法进行训练。具体实现如代码 6 所示。

代码 6: 使用神经网络和半梯度 TD(0) 估计状态价值函数

```
"""
3x3 网格：NN 近似 V_pi(s)
- 动作：right / down（越界不可选）
- 奖励：每步 -1；进 s4 额外 -3；进 s5 额外 -0.5；到 s9 +1（等价进终点净0）
- 策略：在可行动作间均匀随机
- 训练：TD(0) 半梯度（也可切换成蒙特卡洛回归）
"""
import math, random, numpy as np
import torch, torch.nn as nn, torch.optim as optim
```

```
from collections import defaultdict

# ----------------- 环境 -----------------
gamma = 0.9
states = [f"s{i}" for i in range(1,10)]
terminal = "s9"
coords = {
    "s1":(0,0), "s2":(0,1), "s3":(0,2),
    "s4":(1,0), "s5":(1,1), "s6":(1,2),
    "s7":(2,0), "s8":(2,1), "s9":(2,2),
}
def legal_actions(s):
    r,c = coords[s]
    acts=[]
    if c<2: acts.append("right")
    if r<2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a=="right": r2,c2=r,c+1
    else:          r2,c2=r+1,c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    # 奖励
    rwd = -1.0
    if s2=="s4": rwd -= 3.0
    if s2=="s5": rwd -= 0.5
    if s2=="s9": rwd += 1.0   # 到终点净0
    done = (s2==terminal)
    return s2, rwd, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ----------------- 特征（可换成 one-hot） -----------------
def phi(s):
    r,c = coords[s]
    rn, cn = r/2.0, c/2.0                    # 归一化到 [0,1]
    is_pit    = 1.0 if s=="s4" else 0.0
    is_puddle = 1.0 if s=="s5" else 0.0
    return np.array([1.0, rn, cn, rn*cn, rn*rn, cn*cn, is_pit, is_puddle],
dtype=np.float32)

feat_dim = len(phi("s1"))

# ----------------- 神经网络 -----------------
class ValueNet(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, 1)
```

```
        )
    def forward(self, x):    # x: [B, in_dim]
        return self.net(x).squeeze(-1)

device = torch.device("cpu")
net = ValueNet(feat_dim).to(device)
opt = optim.Adam(net.parameters(), lr=1e-3)

# ----------------- 训练开关 -----------------
USE_TD0 = True   # True: TD(0)；False: 蒙特卡洛回归

# ----------------- 训练 -----------------
episodes = 40000
batch_buf_s, batch_buf_target = [], []

def tensorify(ss):  # list[str] -> tensor
    feats = np.stack([phi(s) for s in ss], axis=0)
    return torch.tensor(feats, dtype=torch.float32, device=device)

for ep in range(episodes):
    # Exploring starts 也行；此处固定从 s1
    s = "s1"
    traj = []  # for MC
    while s != terminal:
        a = random_policy(s)
        s2, rwd, done = step(s, a)

        if USE_TD0:
            # TD(0) 半梯度：target = r + γ V(s')，并 detach V(s')
            with torch.no_grad():
                v_sp = 0.0 if done else net(tensorify([s2]))[0].item()
            target = rwd + gamma * v_sp
            preds = net(tensorify([s]))
            loss = nn.MSELoss()(preds, torch.tensor([target], dtype=torch.float32,
device=device))
            opt.zero_grad(); loss.backward(); opt.step()
        else:
            # 先记轨迹，等 episode 结束再做 MC 回归
            traj.append((s, rwd))
        s = s2

    if not USE_TD0:
        # 蒙特卡洛：从后往前算 G，并回归到 V(s)
        G = 0.0
        for s, rwd in reversed(traj):
            G = rwd + gamma*G
            batch_buf_s.append(s); batch_buf_target.append(G)
        # 小批量更新
        if len(batch_buf_s) >= 64:
            X = tensorify(batch_buf_s)
            y = torch.tensor(batch_buf_target, dtype=torch.float32, device=device)
            pred = net(X)
            loss = nn.MSELoss()(pred, y)
```

```
            opt.zero_grad(); loss.backward(); opt.step()
            batch_buf_s.clear(); batch_buf_target.clear()

# ----------------- 评估与打印 -----------------
with torch.no_grad():
    grid = np.zeros((3,3), dtype=np.float32)
    for r in range(3):
        for c in range(3):
            sid = f"s{r*3+c+1}"
            grid[r,c] = 0.0 if sid==terminal else net(tensorify([sid]))[0].item()
    print("Estimated V_pi(s) by NN (rows=row1..row3):")
    print(np.round(grid, 3))

# 热力图:
# import matplotlib.pyplot as plt
# plt.imshow(grid, origin='upper');
# for i in range(3):
#   for j in range(3): plt.text(j,i,f"{grid[i,j]:.2f}",ha='center',va='center')
# plt.title("NN-approximated V(s)"); plt.colorbar(); plt.show()
```

执行代码后，参考结果如代码 7 所示。

代码 7: 神经网络估计的状态价值函数结果

```
Estimated V_pi(s) by NN (rows=row1..row3):
[[-4.223 -2.105 -1.029]
 [-2.089 -1.019  0.01 ]
 [-1.004  0.034  0.   ]]
```

可以看到，神经网络成功地近似了状态价值函数 $V(s)$，并且结果与之前的方法相似，说明神经网络作为一种强大的函数近似工具，能够有效地支持强化学习中的价值函数估计任务。

我们再使用独热编码来表示状态输入，并使用相同的神经网络结构进行训练，具体实现如代码 8 所示。

代码 8: 使用独热编码和神经网络估计状态价值函数

```
"""
TD(0) 半梯度 + 神经网络 + one-hot 状态表示
3×3 网格：right/down；深坑-3，水洼-0.5，每步-1，终点+1（净0）
"""
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random

# ---------- 环境 ----------
gamma = 0.9
states = [f"s{i}" for i in range(1, 10)]
terminal = "s9"
coords = {
    "s1": (0,0), "s2": (0,1), "s3": (0,2),
    "s4": (1,0), "s5": (1,1), "s6": (1,2),
```

```
    "s7": (2,0), "s8": (2,1), "s9": (2,2),
}

def legal_actions(s):
    r,c = coords[s]
    acts = []
    if c < 2: acts.append("right")
    if r < 2: acts.append("down")
    return acts

def step(s,a):
    r,c = coords[s]
    if a == "right": r2,c2 = r, c+1
    else:            r2,c2 = r+1, c
    s2 = next(k for k,v in coords.items() if v==(r2,c2))
    reward = -1.0
    if s2 == "s4": reward -= 3.0
    if s2 == "s5": reward -= 0.5
    if s2 == "s9": reward += 1.0  # 到终点净0
    done = (s2 == terminal)
    return s2, reward, done

def random_policy(s):
    return random.choice(legal_actions(s))

# ---------- One-hot 状态编码 ----------
def onehot(s):
    idx = int(s[1:]) - 1  # s1->0, s2->1, ...
    vec = np.zeros(9, dtype=np.float32)
    vec[idx] = 1.0
    return vec

# ---------- 神经网络 ----------
class ValueNet(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 32), nn.ReLU(),
            nn.Linear(32, 1)
        )
    def forward(self, x):
        return self.net(x).squeeze(-1)

device = torch.device("cpu")
net = ValueNet(9).to(device)
opt = optim.Adam(net.parameters(), lr=1e-3)

# ---------- 训练 ----------
episodes = 40000
alpha = 0.1

def tensorify(slist):
    feats = np.stack([onehot(s) for s in slist], axis=0)
```

```
    return torch.tensor(feats, dtype=torch.float32, device=device)

for ep in range(episodes):
    s = "s1"
    while s != terminal:
        a = random_policy(s)
        s2, r, done = step(s, a)

        with torch.no_grad():
            v_next = 0.0 if done else net(tensorify([s2]))[0].item()
        target = r + gamma * v_next
        v_pred = net(tensorify([s]))
        loss = (v_pred - target) ** 2

        opt.zero_grad()
        loss.backward()
        opt.step()
        s = s2

# ---------- 结果可视化 ----------
with torch.no_grad():
    grid = np.zeros((3,3))
    for r in range(3):
        for c in range(3):
            sid = f"s{r*3+c+1}"
            grid[r,c] = 0.0 if sid==terminal else net(tensorify([sid]))[0].item()

print("Estimated V_pi(s) by NN with onehot (rows=row1..row3):")
print(np.round(grid, 3))

# ---------- 热力图展示 ----------
# import matplotlib.pyplot as plt
# plt.imshow(grid, cmap='coolwarm', origin='upper')
# for i in range(3):
#     for j in range(3):
#         plt.text(j,i,f"{grid[i,j]:.2f}",ha='center',va='center',color='black')
# plt.title("Value Approximation with One-hot Encoding (TD(0))")
# plt.colorbar(label="V(s)")
# plt.tight_layout()
# plt.show()
```

执行代码后，参考结果如代码 9 所示。

代码 9: 独热编码神经网络估计的状态价值函数结果

```
Estimated V_pi(s) by NN with onehot (rows=row1..row3):
[[-4.525e+00 -2.156e+00 -9.900e-01]
 [-2.092e+00 -1.004e+00 -1.300e-02]
 [-1.005e+00  2.000e-03  0.000e+00]]
```

可以看到，使用独热编码作为状态表示，神经网络同样能够有效地近似状态价值函数 $V(s)$，并且结果与之前的方法是相似的。

到目前为止，结合前面章节内容，对于 3×3 网格示例，我们使用了多种方法来估计状态价值函数 $V(s)$，包括表格方法、线性函数近似（手工特征和独热编码）以及神经网络近似（手工特征和独热编码），并主要使用时序差分方法（$\mathrm{TD}(0)$）进行训练迭代。这些方法的结果都较为接近，说明它们在这个简单环境中都能有效地估计价值函数。

然而，随着环境复杂度的增加，例如状态空间变大、状态和动作的关系更复杂，简单的表格或者线性函数近似可能无法捕捉到足够的信息，而神经网络由于其强大的表达能力，往往能够更好地适应复杂环境中的价值函数估计任务。

神经网络拓展

前面我们介绍了使用基础的前馈神经网络来近似价值函数的方法，这种神经网络通常由若干个全连接层组成，并使用非线性激活函数来增强表达能力，也就是我们常说的多层感知机（multi-layer perceptron，MLP）或全连接网络（fully connected network）。这种神经网络结构适用于处理低维的、结构化的输入数据，例如前面示例中的手工设计特征或者独热编码等。

然而，在实际应用中，针对不同类型的数据和任务，可能需要使用更复杂的神经网络结构来更好地捕捉数据的特征和模式。例如，卷积神经网络（convolutional neural network，CNN）适用于处理图像数据，能够有效地提取空间特征；循环神经网络（recurrent neural network，RNN）适用于处理序列数据，能够捕捉时间依赖关系，具体总结如表 1 所示。

表 1: 常用神经网络类型及其特点

神经网络类型	适用场景	主要特点
全连接网络（MLP）	低维结构化数据	多层全连接层，适用于一般函数近似
卷积神经网络（CNN）	图像、视频等网格数据	局部感受野、权重共享、池化层
循环神经网络（RNN）	序列数据（文本、时间序列）	循环连接，捕捉时间依赖关系
LSTM / GRU	长序列数据	门机制，解决梯度消失问题
Transformer	序列数据（文本、时间序列）	自注意力机制，适合并行计算
图神经网络（GNN）	图结构数据	节点和边的特征传播

在强化学习中，选择合适的神经网络结构对于成功应用深度强化学习算法至关重要。此外，复杂的神经网络通常需要更多的数据和计算资源来进行训练，因此在设计神经网络时需要权衡模型复杂度和计算效率，以确保模型能够在合理的时间内收敛并达到良好的性能。