第六章　价值函数的近似表示

本章之前的内容介绍的多是规模比较小的强化学习问题，生活中有许多实际问题要复杂得多，有些是属于状态数量巨大甚至是连续的，有些行为数量较大或者是连续的。这些问题要是使用前几章介绍的基本算法效率会很低，甚至会无法得到较好的解决。本章就聚焦于求解那些状态数量多或者是连续状态的强化学习问题。

解决这类问题的常用方法是不再使用字典之类的查表式的方法来存储状态或行为的价值，而是引入适当的参数，选取恰当的描述状态的特征，通过构建一定的函数来近似计算得到状态或行为价值。对于带参数的价值函数，只要参数确定了，对于一个确切的由特征描述的状态就可以计算得到该状态的价值。这种设计的好处是不需要存储每一个状态或行为价值的数据，而只需要存储参数和函数设计就够了，其优点是显而易见的。

在引入近似价值函数后，强化学习中不管是预测问题还是控制问题，就转变成近似函数的设计以及求解近似函数参数这两个问题了。函数近似主要分为线性函数近似和非线性近似两类，其中非线性近似的主流是使用深度神经网络计数。参数则多使用建立目标函数使用梯度下降的办法通过训练来求解。由此诞生了著名的强化学习算法：深度 Q 学习网络 (DNQ)。本章将就这些问题详细展开。

6.1　价值近似的意义

前一章介绍的几个强化学习算法都属于查表式 (table lookup) 算法，其特点在于每一个状态或行为价值都用一个独立的数据进行存储，整体像一张大表格一样。在编程实践中多使用的字典这个数据结构，我们通过查字典的方式来获取状态和行为的价值。这种算法的设计对于状态或行为数量比较少的问题是快速有效的。例如在有风格子世界中，每一个小格子代表一个状态，一共只有 70 个状态，而个体的行为只有标准的 4 个移动方向，总体上也一共只有 280 个价值数据，可以说规模相当的小。

现在我们来考虑如图 6.1 所示的一个比较经典的冰球世界 (PuckWorld) 强化学习问题。环境由一个正方形区域构成代表着冰球场地，场地内大的圆代表着运动员个体，小圆代表着目标冰

球。在这个正方形环境中，小圆会每隔一定的时间随机改变在场地的位置，而代表个体的大圆的任务就是尽可能快的接近冰球目标。大圆可以操作的行为是在水平和竖直共四个方向上施加一个时间步时长的一个大小固定的力，借此来改变大圆的速度。环境会在每一个时间步内告诉个体当前的水平与垂直坐标、当前的速度在水平和垂直方向上的分量以及目标的水平和垂直坐标共 6 项数据，同时确定奖励值为个体与目标两者中心距离的负数，也就是距离越大奖励值越低且最高奖励值为 0。

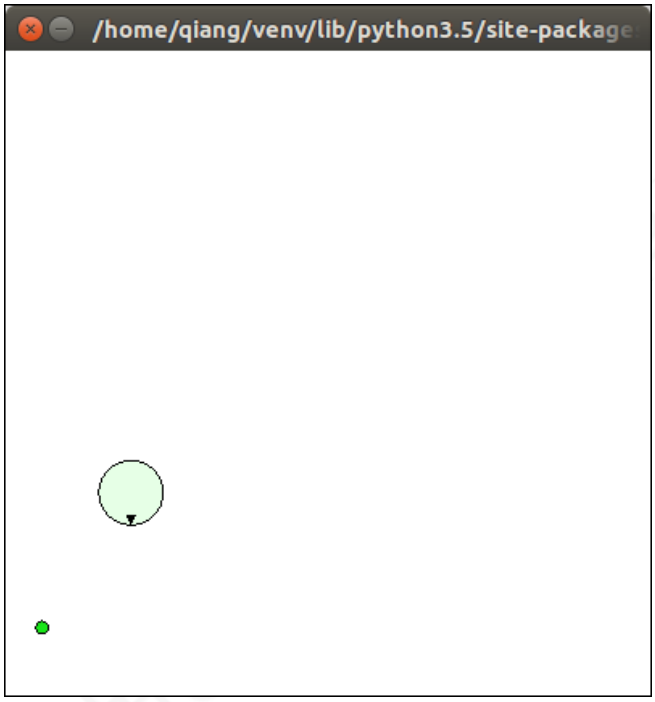


图 6.1: PuckWorld 环境界面

如果打算用强化学习的算法来求解，其中一个重要的问题是如何描述作为大圆的个体的在环境中的状态？根据描述，环境给予的状态空间有 6 个特征，每一个特征都是连续的变量。如果把正方形场地的边长认为是单位值 1 的话，描述个体水平坐标的特征其取值可以在 0 和 1 之间的所有连续变量，其它 5 个特征也类似。如果一定要采取查表式的方式确定每一个状态行为对的价值，那么只能将每一个特征进行分割，例如把 0 到 1 之间平均分为 100 等份，当个体的水平坐标位于这 100 个区间的某一区间内时，其水平坐标的特征值相同。这种近似的求解方式看似有效，但存在一个问题：多少等份才合理。如果分隔得越细，结果势必越准确，但带来状态的数目势必要增大很多。即时每一个特征都做分割成 100 等份区间的话，对于一共有 6 个特征的状态空间和 4 个离散行为的行为空间来说，需要 $100^6 * 4 = 4 * 10^{12}$ 个数据来描述行为价值，如果每个数据用 1 个字节表示，则一共需要 3726G 的内存容量，这无疑是不现实的。如果分隔得

区间数较少，那么问题有可能得不到较好的解决。即时你拥有一台容量足够的计算机能使用查表法解决上述问题，如果你仔细查看表中的数据，会发现某一个特征比较接近的那些行为价值也比较接近。这无疑也不是经济、高效的解决办法。

如果能建立一个函数 $\hat{v}$, 这个函数由参数 w 描述，它可以直接接受表示状态特征的连续变量 s 作为输入，通过计算得到一个状态的价值，通过调整参数 w 的取值，使得其符合基于某一策略 π 的最终状态价值，那么这个函数就是状态价值 $v_\pi(s)$ 的近似表示。

$$\hat{v}(s,w) \approx v_\pi(s)$$

类似的，如果由参数 w 构成的函数 $\hat{q}$ 同时接受状态变量 s 和行为变量 a，计算输出一个行为价值，通过调整参数 w 的取值，使得其符合基于某一策略 π 的最终行为价值，那么这个函数就是行为价值 $q_\pi(s,a)$ 的近似表示。

$$\hat{q}(s,a,w) \approx q_\pi(s,a)$$

此外，如果由参数构成的函数仅接受状态变量作为输入，计算输出针对行为空间的每一个离散行为的价值，这是另一种行为价值的近似表示。在上面的公式中，描述状态的 s 不在是一个字符串或者一个索引，而是由一系列的数据组成的向量，构成向量的每一项称为状态的一个特征，该项的数据值称为特征值；参数 w 需要通过求解确定，通常也是一个向量 (或矩阵、张量等)。

图 6.2 直观的显示了上述三种价值近似表示的特点。

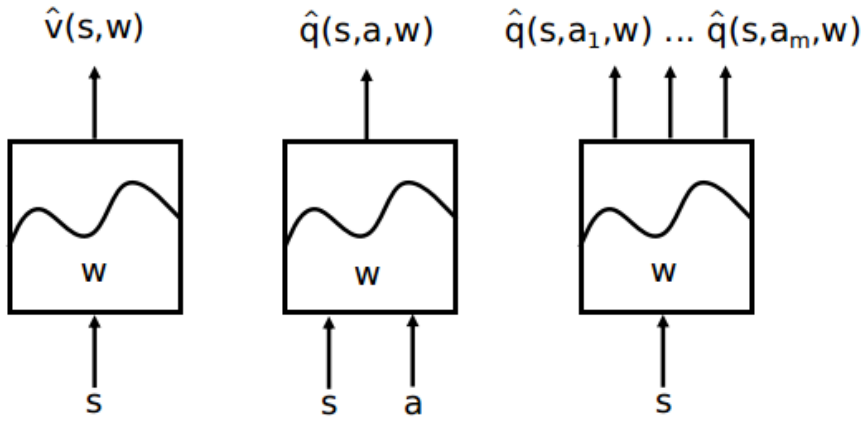


图 6.2: 三种不同类型的价值函数架构

构建了价值函数的近似表示，强化学习中的预测和控制问题就转变为求解近似价值函数参数 w 了。通过建立目标函数，使用梯度下降联合多次迭代的方式可以求解参数 w。

6.2 目标函数与梯度下降

6.2.1 目标函数

先来回顾下上一章中介绍的几个经典学习算法得到最终价值的思想，首先是随机初始化各价值，通过分析每一个时间步产生的状态转换数据，得到一个当前状态的目标价值，这个目标价值由即时奖励和后续价值联合体现。由于学习过程中的各价值都是不准确的，因而在更新价值的时候只是沿着目标价值的方向做一个很小幅度 (α) 的更新:

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

不同的算法的差别体现在目标值 $R+\gamma Q(S',A')$ 的选取上。试想一下，如果价值函数最终收敛不再更新，那意味着对任何状态或状态行为对，其目标值与价值相同。对于预测问题，收敛得到的 Q 就是基于某策略的最终价值函数；对于控制问题，收敛得到的价值函数同时也对应着最优策略。

现在把上式中的所有 $Q(S,A)$ 用 $\hat{Q}(S,A,w)$ 代替，就变成了基于近似价值函数的价值更新方法:

$$\hat{Q}(S,A,w) \leftarrow \hat{Q}(S,A,w) + \alpha(R + \gamma \hat{Q}(S',A',w) - \hat{Q}(S,A,w))$$

假设现在我们已经找到了参数使得价值函数收敛不再更新，那么意味着下式成立:

$$\hat{Q}(S,A,w) = R + \gamma \hat{Q}(S',A',w)$$

同时意味着找到了基于某策略的最终价值函数或者是控制问题中的最优价值函数。事实上，很难找到完美的参数 w 使得上式完全成立。同时由于算法是基于采样数据的，即使上式对于采样得到的状态转换成立，也很难对所有可能的状态转换成立。为了衡量在采样产生的 M 个状态转换上近似价值函数的收敛情况，可以定义目标函数 J_w 为:

$$J(w) = \frac{1}{2M}\sum_{k=1}^{M}\left[(R_k + \gamma \hat{Q}(S'_k, A'_k, w)) - \hat{Q}(S_k, A_k, w)\right]^2 \tag{6.1}$$

公式 (6.1) 中 M 为采样得到的状态转换的总数。近似价值函数 $\hat{Q}(S,A,w)$ 收敛意味着 $J(w)$ 逐渐减小。$J(w)$ 的定义使得它不可能是负数同时存在一个极小值 0。目标函数 J 也称为代价函数 (cost function)。如果只有一个 t 时刻的状态转换，则通常称为损失 (loss)。定义损失 $loss(w)$

为：

$$loss(w) = \frac{1}{2}\left[(R_t + \gamma\hat{Q}(S'_t, A'_t, w)) - \hat{Q}(S_t, A_t, w)\right]^2 \tag{6.2}$$

以上是我们从得到最终结果出发、以 TD 学习、行为价值为例设计的目标函数。对于 MC 学习，使用收获代替目标价值：

$$\begin{aligned} J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[G_t - \hat{V}(S_t, w)\right]^2 \\ J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[G_t - \hat{Q}(S_t, A_t, w)\right]^2 \end{aligned} \tag{6.3}$$

对于 TD(0) 和反向认识 TD(λ) 学习来说，使用 TD 目标代替目标价值：

$$\begin{aligned} J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[R_t + \gamma\hat{V}(S'_t, w)) - \hat{V}(S_t, w)\right]^2 \\ J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[R_t + \gamma\hat{Q}(S'_t, A'_t, w)) - \hat{Q}(S_t, A_t, w)\right]^2 \end{aligned} \tag{6.4}$$

对于前向认识 TD(λ) 学习来说，使用 G^λ 或 q^λ 代替目标价值：

$$\begin{aligned} J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[G_t^\lambda - \hat{V}(S_t, w)\right]^2 \\ J(w) &= \frac{1}{2M}\sum_{t=1}^{M}\left[q_t^\lambda - \hat{Q}(S_t, A_t, w)\right]^2 \end{aligned} \tag{6.5}$$

如果事先存在对于预测问题最终基于某一策略最终价值函数 $V_\pi(S)$ 或 $Q_\pi(S, A)$，或者存在对于控制问题的最优价值函数 $V_*(S)$ 或 $Q_*(S, A)$，那么可以使用这些价值来代替上式中的目标价值，这里集中使用 V_{target} 或 Q_{target} 来代表目标价值，使用期望代替平均值的方式，那么目标价值的表述公式为：

$$\begin{aligned} J(w) &= \frac{1}{2}\mathbb{E}\left[V_{target}(S) - \hat{V}(S, w)\right]^2 \\ J(w) &= \frac{1}{2}\mathbb{E}\left[Q_{target}(S, A) - \hat{Q}(S, A, w)\right]^2 \end{aligned} \tag{6.6}$$

事实上，这些目标价值正是我们要求解的。在实际求解近似价值函数参数 w 的过程中，我们使用基于近似价值函数的目标价值来代替。下文还将继续就这一点做出解释。

我们可以利用梯度下降的方法来逐渐逼近能让 $J(w)$ 取得极小值的参数 w。

6.2.2 梯度和梯度下降

梯度是一个很重要的概念，正确理解梯度对于理解梯度下降、梯度上升等算法有这非常重要的意义，而后两者在基于函数近似的强化学习领域有着广泛的应用。这里对梯度做一个较为详细的介绍。

先考虑一元函数的情况。令 $J(w) = (2w - 3)^2$ 是关于参数 w 的一个函数，这里的 w 是一个一维实变量，那么 J(w) 的函数图像如图 6.3 所示。这是一个抛物线，在点 (1.5,0) 处取得极小值。点 (3,9) 位于抛物线上，抛物线在该点的切线方程为 $S(w) = 12w - 27$。该切线的斜率为 12。12 就是抛物线在点 (3,9) 处的导数，也是梯度。类似的在抛物线上的点 (1.5,0) 处，切线的斜率为 0，其在该点的梯度或导数为 0。对于该抛物线来说，在梯度为 0 的位置取得极小值。

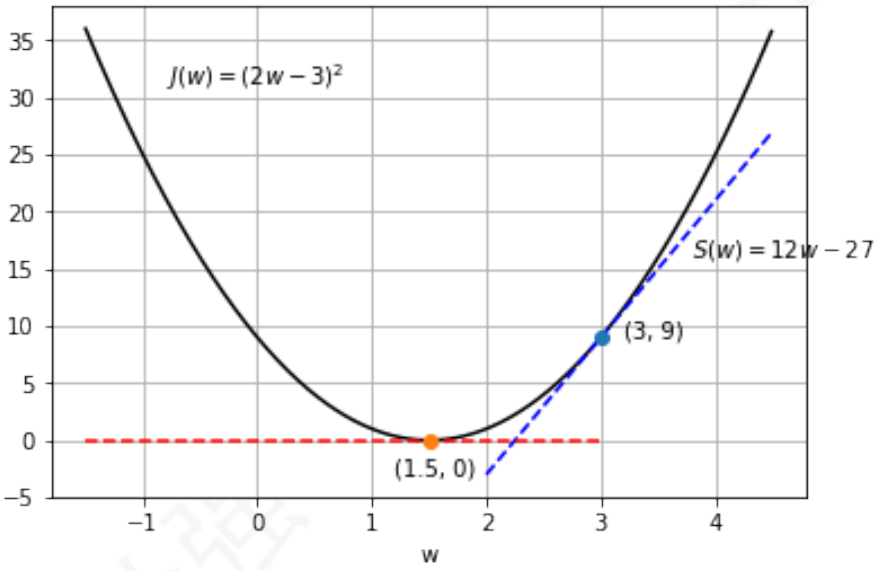


图 6.3: 一元函数的梯度 (导数) 和极值

对于一元可微函数 $y = f(x)$，y 在 x 处的梯度 (或导数) 描述为:

$$\nabla y = f'(x) = \frac{dy}{dx}$$

$f(x)$ 在梯度 (导数) 为 0 处取得一个极大或极小值。

多元函数的情况要复杂些。以二元函数为例，令 $J(w)$ 是关于参数 w_1, w_2 如下的一个二元函数:

$$J(w_1, w_2) = 5(w_1 - 3)^2 + 4(w_2 - 1)^2$$

这里的 $w=(w_1,w_2)$ 是一个实向量。此时作为二元函数的 $J(w_1,w_2)$ 在三维坐标系中是一个曲面，其图像如图 6.4 所示。该曲面呈现开口向上的抛物面，并且在 $w_1=3,w_2=1$ 时取得最小值 0。

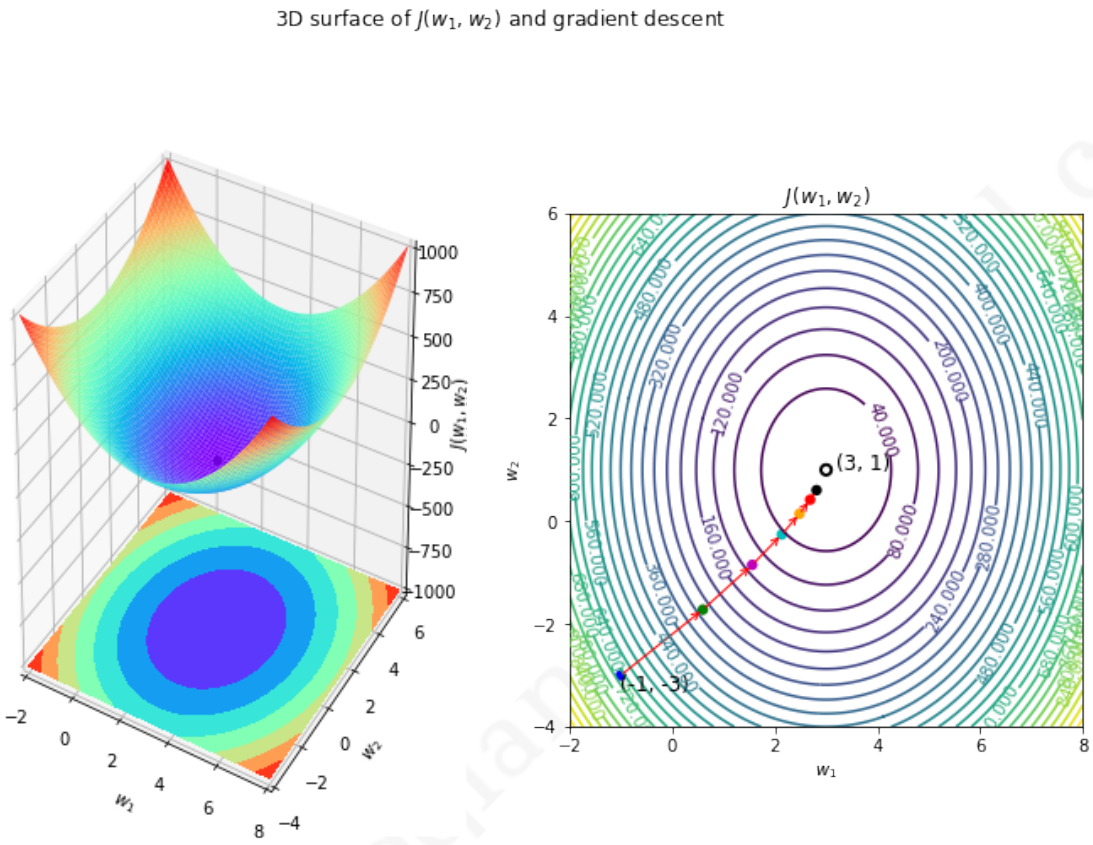


图 6.4: 二元函数对应的曲面及梯度下降演示

对于二元可微函数 $y=f(x_1,x_2)$，y 在 (x_1,x_2) 处的梯度是一个向量，因而是有方向的，其元素由 y 分别对 x_1 和 x_2 的偏导数构成：

$$\nabla y=\left(\frac{\partial y}{\partial x_1},\frac{\partial y}{\partial x_2}\right)$$

梯度的意义在于在某一位置沿着该位置梯度向量所指的方向，函数值增加的速度最快；而梯度向量的反方向就是函数值减少速度最快的方向。当某位置的梯度为 0 时，那么函数在该处取得一个极大值或极小值。图 6.4 右侧部分是左侧曲面在 w_1Ow_2 平面投影的等高线，每一条闭合的圆上的点的函数值相同。某一点梯度的方向就是过该点垂直于该点所在等高线且指向函数值增大的方向。

根据偏导数的计算公式，可以得出前面的二元函数 $J(w_1,w_2)$ 的对于 w_1 和 w_2 的偏导数分

别为：

$$\frac{\partial J}{\partial w_1} = 10(w_1 - 3),\ \frac{\partial J}{\partial w_2} = 8(w_2 - 1)$$

可以计算得出 $J(w_1, w_2)$ 在点 $(-1, -3)$ 和 $(3, 1)$ 处的梯度分别为：

$$\frac{\partial J}{\partial w_1}|_{(-1,-3)} = -40,\ \frac{\partial J}{\partial w_2}|_{(-1,-3)} = -32$$

即：

$$\nabla J_{(-1,-3)} = (-40, -32)$$

类似的，该函数在点 $(3, 1)$ 处的梯度为：

$$\nabla J_{(3,1)} = (0, 0)$$

这两个点的位置对应于图 6.4 右侧中的左下角一点和正中的空心圆点。

二元函数的梯度概念和计算方法可以直接推广至 n 元函数 $y = f(x_1, x_2, \ldots, x_n)$ 中，只要该函数在其定义空间内可微，这里就不在展开了。

利用沿梯度方向函数值增加速度最快，沿梯度反方向函数值减少速度最快这个特点，通过设计合理的目标函数 J, 可以找到目标函数取极小或极大值时对应的自变量 w 的取值。

对于一个类似上例的可微二元函数来说，可以直接令其对应各参数的偏导数为 0，直接求出当函数值取极小值是的参数值。不过对于大规模强化学习问题来说。J 是根据采样得到的状态转换来计算的，因此直接对其求梯度来并不能得到最优参数。这种情况下需要通过迭代、使用梯度下降来求解。具体过程如下：

1. 初始条件下随机设置参数 $w = (w_1, w_2, \ldots, w_n)$ 值；

2. 获取一个状态转换，代入目标函数 J，并计算 J 对各参数 w 的梯度:

$$\nabla_w J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix} \tag{6.7}$$

3. 设置一个正的较小的学习率 α，将原参数 w 朝着梯度的反方向作一定的更新：

$$\begin{aligned} \Delta w &= -\alpha\ \nabla_w J(w) \\ w &\leftarrow w + \Delta w \end{aligned} \tag{6.8}$$

4. 重复过程 2,3，直到参数 w 的更新小于一个设定范围或者达到一定的更新次数。

以上述的二元函数 $J(w_1, w_2) = 5(w_1 - 3)^2 + 4(w_2 - 1)^2$ 为例，我们希望找到能使 $J(w_1, w_2)$ 最小的 (w_1, w_2) 的值。很明显当 $w_1 = 3, w_2 = 1$ 时，J 取得最小值 0。这是直接令梯度为 0 计算得到的。现在我们使用梯度下降法来求解：

第一步：随机初始化 (w_1, w_2)，假设令 $w_1 = -1, w_2 = -3$，该值对应图 6.4 右侧最左下角的一个点；

第二步：计算目标函数 J 对应当前参数的梯度为 (-40,-32)；

第三步：设置学习率 $\alpha = 0.04$，更新参数值：$w_1 = -1 - 0.04 * (-40) = 0.6$，$w_2 = -3 - 0.04 * (-32) = -1.72$。该参数在图中对应的点比之前朝着图中央的空心圆圈迈了一大步。

重复第二、三步，我们发现更新的参数值对应的点离中心代表最终参数的点逐渐接近，直到第 6 次更新后已经到达离中心最近的黑点的位置，该位置对应的参数值为 $(2.81, 0.60)$。随着更新次数的增加，得到的参数值将逐渐趋于真实的参数值。

如果我们在更新参数时设置较大的或较小的学习率，会改变学习的进程：较小的学习率使得更新速度变慢，需要更多的更新次数才能得到最终结果；设置较大的学习率在更新早期虽能加快速度，但在后期有可能得不到最终想要的结果。图 6.5 显示的是上述问题使用 $\alpha = 0.2$ 时的参数更新过程，可以看出参数的第一次更新已经过头了，此后的每一次更新都围绕着真实参数值两侧震荡，无法到达真实参数值。因此要小心设置学习率的大小。为了解决这个问题，也出现了许多其它的更新办法，诸如带动量的更新和 Adam 更新等，有兴趣的读者可以参考深度学习相关的知识。

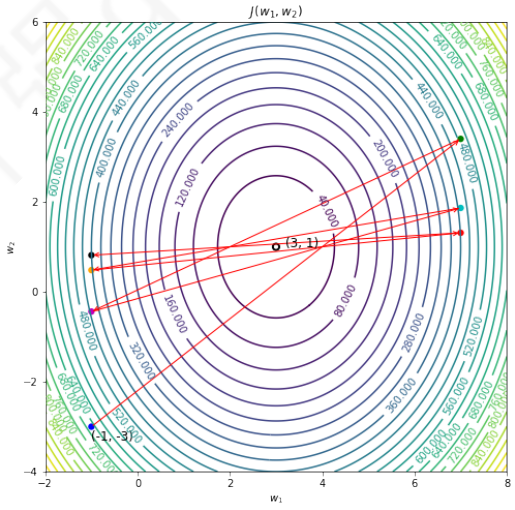


图 6.5: 过大的学习率导致梯度下降求解失败

6.3 常用的近似价值函数

理论上任何函数都可以被用作近似价值函数，实际选择何种近似函数需根据问题的特点。比较常用的近似函数有线性函数组合、神经网络、决策树、傅里叶变换等等。这些近似函数用在强化学习领域中主要的任务是对一个状态进行恰当的特征表示。近年来，深度学习技术展示了其强大的特征表示能力，被广泛应用于诸多技术领域，也包括强化学习。本节将简要介绍线性近似。随后终点介绍基于深度学习的神经网络计数进行特征表示，包括卷积神经网络。

6.3.1 线性近似

线性价值函数使用一些列特征的线性组合来近似价值函数：

$$\hat{V}(S,w) = w^T\ x(S) = \sum_{j=1}^{n} x_j(S) w_j \tag{6.9}$$

公式 (6.9) 中的 $x(S)$ 和 w 均为列向量，$x_j(S)$ 表示状态 S 的第 j 个特征分量值，w_j 表示该特征分量值的权重，也就是要求解的参数。基于公式 (6.6)，对于由线性函数 $\hat{V}(S,w)$ 近似价值函数，其对应的目标函数 $J(w)$ 为：

$$J(w) = \frac{1}{2}\ \mathbb{E}\left[V_{target}(S) - w^T\ x(S)\right]^2$$

相应的梯度 $\nabla_w J(w)$ 为：

$$\nabla_w J(w) = -\left(V_{target}(S) - w^T\ x(S)\right)\ x(S)$$

参数的更新量 Δw 为：

$$\Delta w = \alpha\left(V_{target}(S) - w^T\ x(S)\right)\ x(S) \tag{6.10}$$

上式中，使用不同的学习方法时，$V_{target}(S)$ 由不同的目标价值代替，这一点前文已经说过，需要指出的是，这些目标价值虽然仍然由近似价值函数计算得到，但在计算梯度时它们是一个数值，对参数 w 的求导为 0。

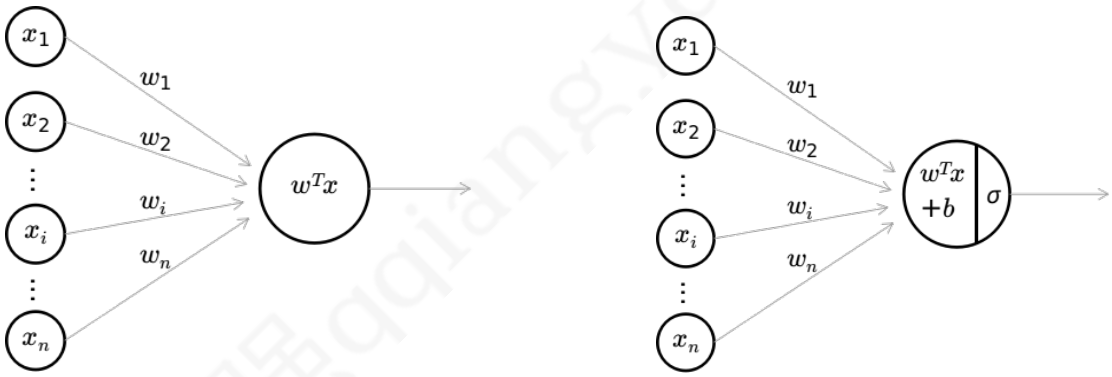
事实上，查表式的价值函数是线性近似价值函数的一个特例，这个线性近似价值函数的特征数目就是所有的状态数目 n，每一个特定的状态对应的线性价值函数的特征分量中只有一个为 1，其余 $n-1$ 个特征分量值均为 0；类似的参数也是由 n 个元素组成的向量，每一个元素实际

上就存储着对应的价值。如公式所示。

$$\hat{V}(S,w) = \begin{pmatrix} 1(S=s_1) \\ 1(S=s_2) \\ \vdots \\ 1(S=s_n) \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \tag{6.11}$$

6.3.2 神经网络

神经网络近似是一种非线性近似，它的基本单位是一个可以进行非线性变换的神经元。通过多个这样的神经元多层排列、层间互连，最终实现复杂的非线性近似。线性近似可以用图 6.6(a) 表示，一个基本的神经元可以用图 6.6(b) 表示。



(a) 线性近似　　(b) 非线性近似的神经元

图 6.6: 线性近似与单个神经元的非线性近似

单个神经元在线性近似的基础上增加了一个偏置项 (b) 和一个非线性整合函数 (σ)，偏置项 b 可以被认为是一个额外的数据为 1 的输入项的权重。单个神经元最终的输出 $\hat{y}$ 为

$$\hat{y} = \sigma(z) = \sigma(w^T x + b) \tag{6.12}$$

非线性函数 σ 被称为神经元的激活函数 (activate function)，目前最常用的两个激活函数是 relu 和 tanh，它们的函数图像如图 6.7 所示。

神经网络则是由众多能够进行简单非线性近似的神经元按照一定的层次连接组成。图 6.8 显示的是一个 2 层神经网络，接受 n 个特征的输入数据，得到具有两个特征的输出。其接受输入数

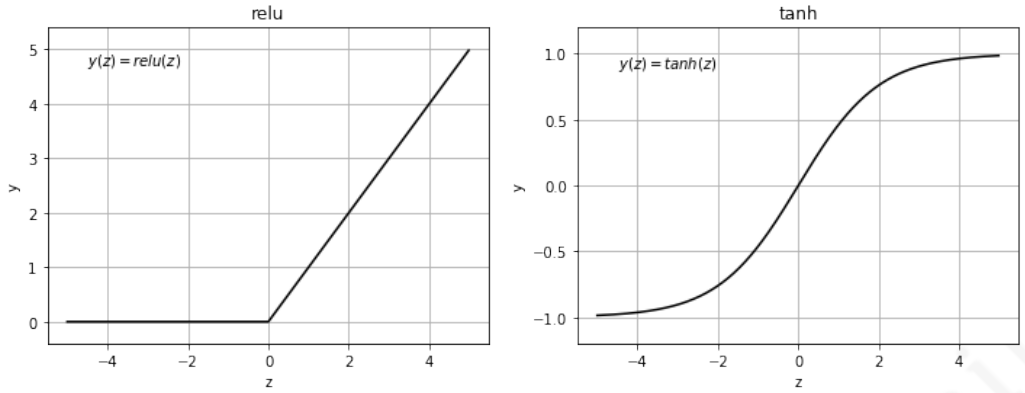


图 6.7: 两个常用的激活函数

据的第一层有 16 个神经元。输入数据可以被认为一个神经元，不过神经元的输出为给定的输入数据。图中除输入数据外的每一个神经元都和前一层的所有神经元以一定的权重 w 链接。这种连接方式称为全连接，对应的神经网络为分层全连接神经网络 (full connect neural network)。神经网络在无特别说明时一般均指全连接神经网络。

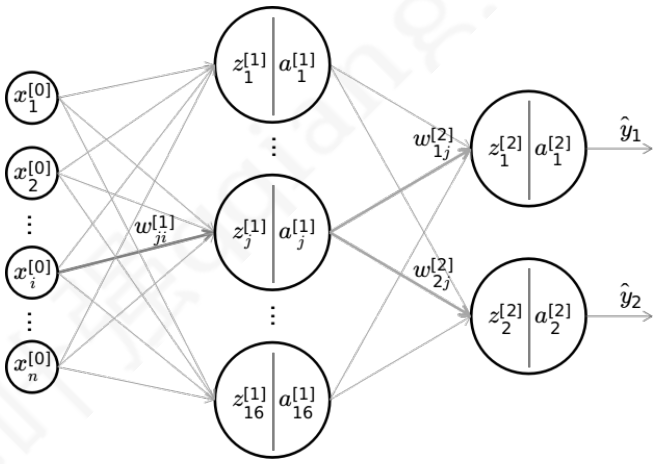


图 6.8: 一个简单的 2 层神经网络架构

分层全连接神经网络中除输入数据和输出层以外的层都叫隐藏层 (hidden layer)。图 6.8 所示的是有一个隐藏层的神经网络，该隐藏层有 16 个神经元。如果设置不同的隐藏层同时隐藏层设置不同数目的神经元，那么就形成了不同设置的神经网络。一般来说，隐藏层越多网络越深，相应的其对训练数据的非线性近似能力越强。隐藏层达到多少可以称之为深度神经网络 (deep neural network, DNN) 并没有明确的说法。

对于一个多层神经网络来说，每一层的每一个神经元都有多个代表权重的参数 w 和一个偏

置项 b。如果用 $n^{[l]}$ 表示第 l 层的神经元数量，$a_i^{[l]}$ 表示第 l 层的第 i 个神经元的输出，$w_{ji}^{[l]}$ 表示第 l 层第 j 个神经元与第 $l-1$ 层第 i 个神经元之间的链接权重，$b_j^{[l]}$ 表示第 l 层第 j 个神经元的偏置项，那么一个 L 层全连接神经网络，其参数由：

$$\langle W^{[1]}, b^{[1]} \rangle, \langle W^{[2]}, b^{[2]} \rangle, \ldots, \langle W^{[L]}, b^{[L]} \rangle$$

构成。其中 $W^{[l]}$ 是一个 $n^{[l]} \times n^{[l-1])}$ 的二维矩阵，$b^{[l]}$ 是由 $n^{[l]}$ 个元素构成的一维列向量。网络中第 l 层第 j 个神经元的输出 $a_j^{[l]}$ 为:

$$a_j^{[l]} = \sigma(z_j^{[l]}) = \sigma(\sum_{i=1}^{n^{[l-1]}} w_{ji}^{[l]} + b_j^{[l]}) \tag{6.13}$$

如果使用矩阵的形式，那么第 l 层神经元的输出 $a^{(l)}$ 为:

$$a^{[l]} = \sigma(z^{[l]}) = \sigma(W^{[l]}\ a^{[l-1]} + b^{[l]}) \tag{6.14}$$

神经网络第 L 层的输出 $a^{[L]}$ 也就是该网络的最终输出 $\hat{y}$。当神经网络的参数确定时，给以神经网络的输入层一定的数据，依据公式 (6.14) 可以得到第一个隐藏层的输出，第一个隐藏层的输出作为输入数据可以计算第二个隐藏层的输出，直至计算得到输出层一个确定的输出。这种由输入数据依次经过网络的各层得到输出数据的过程称为前向传播 (forward propagation)。通过设计合理的目标 (代价) 函数，也可以利用前文介绍的梯度及梯度下降方法来求解符合任务需求的参数。只不过在计算梯度时需要从神经网络的输出层开始逐层计算值第一个隐藏层甚至是输入层。这种梯度从输出端向输入端计算传播的过程称为反向传播 (backward propagation, BP)。目标函数计算神经网络根据输入数据得到的输出 $\hat{y}$ 与实际期望的输出 y 之间的误差，计算对各参数的梯度，朝着误差减少的方向更新参数值。均方差 (mean square error, MSE) 和交叉熵 (cross_entropy) 是神经网络常用的两大目标函数:

$$J(W,b)_{MSE} = \frac{1}{2M}\sum_{k=1}^{M}\left[y^{(k)} - \hat{y}^{(k)}\right]^2 \tag{6.15}$$

$$J(W,b)_{cross\_entropy} = -\frac{1}{M}\sum_{k=1}^{M}\left[y^{(k)}\ln\hat{y}^{(k)} + (1-y^{(k)})\ln(1-\hat{y}^{(k)})\right] \tag{6.16}$$

公式 (6.15) 和 (6.16) 中的 M 指更新一次参数对应的的训练样本的数量，$y^{(k)}$ 和 $\hat{y}^{(k)}$ 分别表示第 k 个训练样本的真实输出和神经网络计算得到的输出。均方差目标函数多数常用于输出

值的绝对值大于 1 等较大数值范围内；而交叉熵目标函数由于其设计特点一般多用于输出值在 0 和 1 之间的情况。

在使用由 M 个训练样本组成的训练集训练一个神经网络时，如果每训练一个样本就计算一次目标函数并使用梯度下降算法更新网络参数，这种训练方法称为随机梯度下降 (stochastic gradient descent)；如果一次目标函数计算了训练集中的所有 M 个样本，在此基础上更新参数则称为块梯度下降 (batch gradient descent)；如果每训练 $m(m \ll M)$ 个样本更新一次参数值，这种方法称为小块梯度下降 (mini-batch gradient descent)。随机梯度下降参数更新快，但有时候会朝着错误的方向更新；块梯度梯度下降一般总能找到对应训练集的最优参数，但收敛速度慢。小块梯度下降方法结合了两者的优点，是目前主流的算法。由于 M 一般在数千、数万乃至更高级别范围，通常 m 则可以选择 64, 128, 256 等较小的 2 的整数次方。

用全连接神经网络作为强化学习中的价值近似能够很好的解决一些规模较大的实际问题，这得益于全连接神经网络有强大的各级特征学习能力和非线性整合能力。当面对图像数据作为表示状态的主要数据时，可以使用另一种强大的神经网络：卷积神经网络来进行价值近似。

6.3.3 卷积神经网络近似

卷积神经网络 (convolutional neural network, CNN) 是神经网络的一种，神经网络的许多概念和方法，例如单个神经元的工作机制、激活函数、连接权重、目标函数、反向传播、训练机制等都适用于卷积神经网络。卷积神经网络与全连接神经网络最大的差别在于网络的架构和参数的设置上，本节将对此进行基本的讲解。

卷积神经网络也是分层的，也可以有很多隐藏层，但是其主体部分每个隐藏层由许多通道 (channel) 组成，每一个通道内有许多神经元，其中每一个神经元仅与前一层所有通道内的局部位置的神经元连接，接受它们的输出信号，这种设计思想是受到高等哺乳动物的视觉神经系统中“感受野”概念的启发。此外，同一个通道内的神经元共享参数，这种设计能够比较有效的提取静态二维空间的特征。先从一个简单的例子开始讲解什么是卷积。

大多数人都玩过五子棋游戏 (图 6.9)，游戏双方分别使用黑白两色的棋子，轮流下在棋盘水平横线与垂直竖线的交叉点上，先在横线、竖线或斜对角线上形成 5 子连线者获胜。

下五子棋时经常要观察对手是否出现了 3 个棋子连在一起的情况，如果出现这种情况，那么就要考虑及时围堵对手了。我们可以使用一个过滤器来检测水平、垂直方向是否存在某玩家 3 子相连的情况。以图 6.9 中标号为 5 的那一行棋 (图 6.10(a)) 为例，为了方便计算，我们使用数字来标记棋盘上的每一个位置信息，0 表示该位置没有棋，1 表示该位置被黑棋占据；-1 表示该位置被白棋占据。随后设计如图 6.10(b) 所示三个水平连续的 1 作为过滤器来检测水平方向一方 3 子连线的情况。得到如图 6.10(c) 所示的过滤结果。

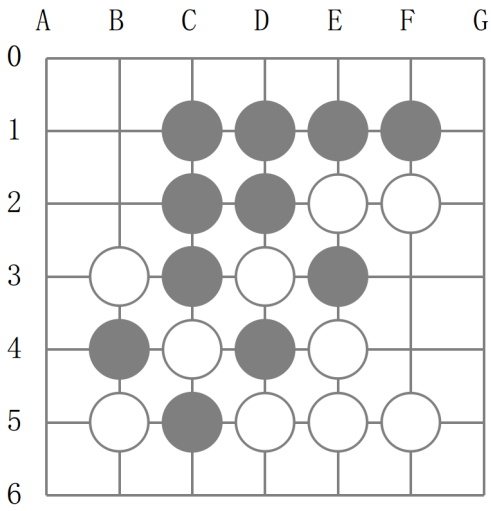
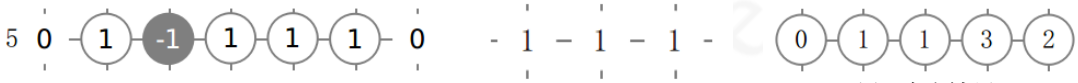


图 6.9: 一个 7×7 的五子棋局



(a) 第 5 行棋局位置信息 (b) 过滤器 (c) 过滤结果

图 6.10: 使用过滤器检测五子棋一方 3 子连线

过滤器的目标是发现感兴趣的特征信息而过滤掉无关信息，过滤结果会提示棋局中是否存在过滤器感兴趣的信息。其具体的工作机制如图 6.11所示。

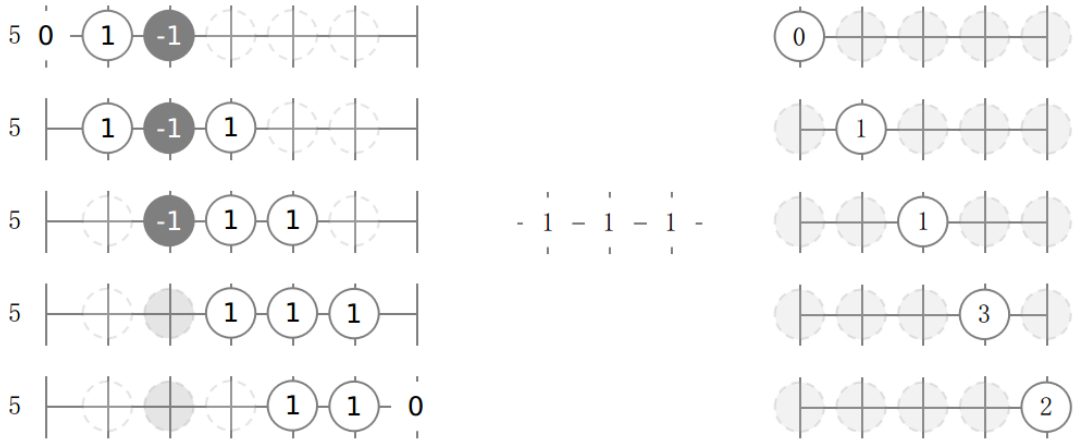


图 6.11: 检测五子棋水平方向 3 子连线的过滤器工作机制

过滤器将从该行棋局最左侧的三个棋盘位置信息开始，分别用自身的三个 1 乘以棋盘上相

应位置数字形式表示的状态，并把结果加在一起，得到最后的过滤结果。每得到一个结果后过滤器往右移动一步，再次计算新位置的过滤结果，如此反复直到过滤器来到该行棋的最右方。通过分析过滤器的计算方式，我们可以发现，当棋盘中存在 3 个黑棋或 3 个白棋连在一起的时候，过滤器得到的结果将分别是 3 和 -3，而其它情况下过滤器的结果都将在 -3 和 3 之间。所以通过观察过滤结果中有没有-3 或 3 以及出现的位置，就可以得到棋盘中有没有一方 3 子连在一起的情况以及该情况出现的位置了。垂直方向也可以采用类似的方法来判断。

过滤器在卷积神经网络中大量存在，只不过我们使用一个新的名字：卷积核。过滤器通过移步进行过滤操作的过程称为卷积操作。过滤器需要有一定数量的参数来定义，例如上例中过滤器是由一个长度为 3 的一维单位向量组成。这个长度称为卷积核的大小或尺寸。这些参数则形成了卷积核的参数。卷积操作保留了卷积核感兴趣的信息，同时对原始信息进行了一定程度的概括和压缩。例如上例中，本来一行棋的状态需要用 7 个数字来描述，但是卷积操作的结果仅有 5 个数字组成。通过这 5 个数字可以大致分析出对应位置黑棋和白棋的力量对比。有的时候我们希望卷积结果的规模与原始的数据规模一样，那么可以在原始信息的左右两侧添加一定数量的无用数据，例如我们在第五行棋左右再各添加一个 0，这样得到的卷积结果也将是 7 个数字组成的了。卷积结果与原始信息结果保持相同的规模在五子棋问题中意义不大，但在解决其它一些问题中还是有意义的。同样在这个例子中，卷积核每一步仅往右移动一格。在一些任务中，卷积核每一次可以移动超过一格的位置。

一般来说，如果一个原始信息的大小为 n，卷积核 (kernal) 大小为 f，原始信息左右各扩展 (padding)p 个数据，卷积核每次移动的步长 (stride) 为 s，那么卷积结果的数据大小 n' 可由下式计算得到：

$$n' = \frac{n + 2\times p - f}{s} + 1 \tag{6.17}$$

以上介绍的卷积操作都是针对一维空间的，判断水平或垂直方向一方 3 子连线没有问题，但是判断对角线方向 3 子连线就做不到了。由于对角线涉及到水平和垂直两个维度，我们需要把之前一维卷积操作扩展到二维空间中来。如图 6.12 所示。卷积核由二维单位矩阵构成，相应的卷积结果显示有 2 处白棋 3 子连线和 1 处黑棋 3 子连线的情况出现。二维卷积操作的基本原理与一维卷积操作类似，不同的是在计算单个卷积操作结果时将所有元素与卷积核对应位置的值相乘在求和；在移步时注意不要遗漏位置就可以了。而且卷积结果的大小在水平和垂直方向上依然可以使用公式 (6.17) 来计算。卷积操作还可以推广至三维甚至更高维的情况，这里就不展开了。

卷积核以及对应的操作是理解卷积神经网络的关键。在实际应用中，一个卷积核通常不能解决实际问题，即使在五子棋中，检测一方 3 子连线也至少需要 4 个卷积核，分别负责水平、垂直和两个对角线方向的检测，可以认为，每一个卷积核负责检测对应的一个特征。而每一个卷积操作的结果一般称为一个通道 (channel) ，也称为一个映射 (map)。在卷积神经网络中，多个通道构成一个类似于全连接神经网络的隐藏层，同时卷积核的参数并不像我们应用在五子棋例子中

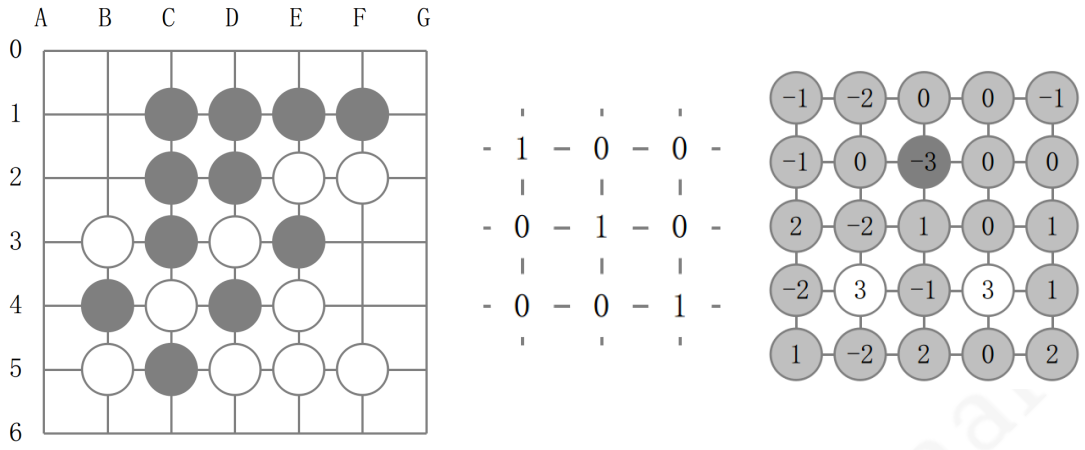
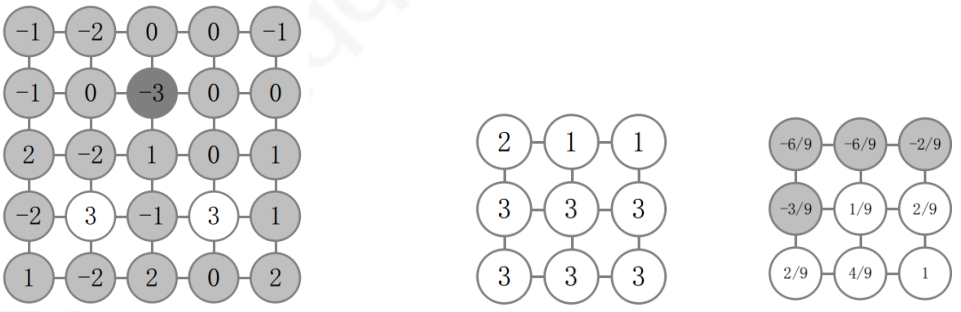


图 6.12: 使用卷积操作检测五子棋 '\' 向对角线方向 3 子连线

那样事先设计好的，而是网络通过学习得到的。卷积神经网络中还有一个概念是池化 (pooling)。池化操作主要分为两种：最大池化 (max pooling) 和平均池化 (average pooling)。池化操作过程与卷积操作类似，也存在着大小、步长、扩展等概念，池化结果的大小也可以使用公式 (6.17) 计算得到。池化操作不需要核，最大池化操作就是把当前操作区域中的最大数据作为池化结果；而平均池化则是把操作区域的所有值的平均值作为池化结果。图 6.13 展示了对五子棋的一个卷积结果分别进行大小为 $f = 3\times3$，步长 $s = 1$，扩展 $p = 0$ 的最大池化和相同参数的平均池化的结果。



(a) 原始信息 (b) 最大池化结果 (c) 平均池化结果

图 6.13: 最大池化和平均池化

在了解了卷积神经网络的一些基本操作后，再来从宏观上理解卷积神经网络的架构就容易些了。图 6.14 展示的是卷积神经网络的鼻祖——1998 年 LeCun 提出的可以用来进行手写数字识别的 LeNet-5 的架构。该网络接受黑白手写字符图片作为输入，随后的第一个隐藏层设计了 6 个通道的 $f = 5\times5, s = 1, p = 0$ 的卷积核，得到的卷积结果规模为 $6\times28\times28$，随后使用了

$f=2\times2, s=2, p=0$ 的池化操作得到第二个隐藏层，同时数据规模缩小为 6×14×14；第三个隐藏层的数据又是通过卷积操作得到，这次设计了 16 个设置与之前相同的卷积核，得到的结果规模为 16×10×10；随后的一个池化操作将数据规模进一步缩小为 16×5×5。此后使用 3 个全连接最终到达长度为 10 的输出层。输出层的每一个数据分别代表输入图像是手写数字 0-9 的概率。

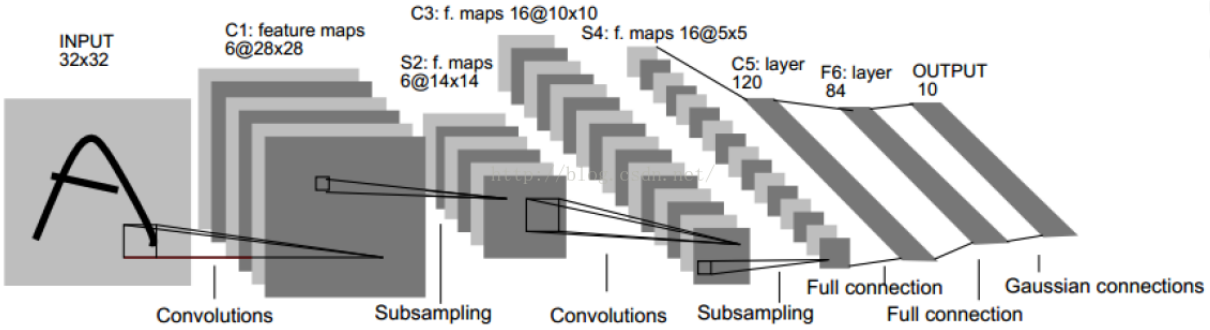


图 6.14: LeNet-5 卷积神经网络

相比较全连接神经网络，卷积神经网络具备出色的空间特征检测能力，而且参数规模小。如果一个隐藏层设计了 n 个通道，每一个通道的卷积核大小为 $f\times f$，同时前一个隐藏层有 m 个通道的话，不考虑偏置项，该隐藏层一共仅有参数的个数 c 为：

$$c=n\times f\times f\times m \tag{6.18}$$

池化操作不需要任何参数，这大大减少了参数的个数。卷积隐藏层通道的个数反映了网络在这一层的特征检测能力。通常卷积神经网络越靠近输出层其通道数越多，但是每一个通道的数据规模越来越小。此外卷积核的大小也影响网络的架构，早期人们习惯使用诸如 $7\times7, 9\times9$ 的大卷积核，近来研究发现大的卷积核可以由多个 3×3 的卷积核来替代，同时总体参数也减少了。因而目前主流卷积神经网络都倾向于使用 3×3 大小的卷积核。池化和全连接在卷积神经网络中的作用也被越来越淡化，有些卷积神经网络甚至抛弃了全连接层。

强化学习领域有许多图像信息作为状态表示的情况，例如训练一个个体像人类玩游戏的场景一样，通过直接观察游戏屏幕的图像来进行策略优化，这种情况下结合卷积神经网络作为价值函数的近似多会带来效率的提高。在棋类游戏领域也是如此，卷积神经网络可以出色的对当前棋盘状态进行评估，指导价值及策略的优化。本书将在最后一章详细介绍卷积神经网络应用于棋类游戏领域。对初学者来说，卷积神经网络比较难理解，由于本书旨在应用卷积神经网络来进行强化学习，故而不可能花大量的篇幅来讲解卷积神经网络本身。读者可以通过本章的编程实践和本书后续的一些讲解来加深对卷积神经网络的理解。对卷积神经网络感兴趣的读者也可以查阅专门的书籍和文献。

6.4 DQN 算法

本节将结合价值函数近似与神经网络技术，介绍基于神经网络 (深度学习) 的 Q 学习算法：深度 Q 学习 (deep Q-learning, DQN) 算法。DQN 算法主要使用经历回放 (experience replay) 来实现价值函数的收敛。其具体做法为：个体能记住既往的状态转换经历，对于每一个完整状态序列里的每一次状态转换，依据当前状态的 s_t 价值以 ϵ-贪婪策略选择一个行为 a_t，执行该行为得到奖励 r_{t+1} 和下一个状态 s_{t+1}，将得到的状态转换存储至记忆中，当记忆中存储的容量足够大时，随机从记忆力提取一定数量的状态转换，用状态转换中下一状态来计算当前状态的目标价值，使用公式 (6.4) 计算目标价值与网络输出价值之间的均方差代价，使用小块梯度下降算法更新网络的参数。具体的算法流程如算法 4 所示。

算法 4: 基于经历回放的 DQN 算法

输入: episodes, α, γ
输出: optimized action-value function $Q(\theta)$
initialize:experience $\mathbb{D}$, action-value function $Q(\theta)$ with random weights
repeat for each episode in episodes
  get features ϕ of start state S of current episode
  repeat for each step of episode
    $A = \text{policy}(Q, \phi(S); \theta)$ (e.g. ϵ-greedy policy)
    $R, \phi(S'), is\_end = \text{perform\_action}(\phi(S), A)$
    store transition $(\phi(S), A, R, \phi(S'), is\_end)$ in $\mathbb{D}$
    $\phi(S) \leftarrow \phi(S')$
    sample random minibatch(m) of transitions $(\phi(S_j), A_j, R_j, \phi(S'_j), ie\_end_j)$ from $\mathbb{D}$
    set

$$y_j = \begin{cases} r_j & if\ is\_end_j\ is\ True \\ r_j + \gamma \max_{a'} Q(\phi(S'_j), a'; \theta) & otherwise \end{cases}$$

    perform mini-batch gradient descent on $(y_j - Q(\phi(S_j), A_j; \theta))^2/m$
  until S is terminal state;
until all episodes are visited;

该算法流程图中使用 θ 代表近似价值函数的参数。相比前一章的各种学习算法，该算法中的状态 S 都由特征 $\phi(S)$ 来表示。为了表述得简便，在除算法之外的公式中，本书将仍直接使用 S 来代替 $\phi(S)$。在每一产生一个行为 A 并与环境实际交互后，个体都会进行一次学习并更新一次参数。更新参数时使用的目标价值由公式 (6.19) 产生：

$$Q_{target}(S_t, A_t) = R_t + \gamma\ max\ Q(S'_t, A'_t; \theta^-) \tag{6.19}$$

公式 (6.19) 中的 θ^- 是上一个更新周期价值网络的参数。DQN 算法在深度强化学习领域取

得了不俗的成绩，不过其并不能保证一直收敛，研究表明这种估计目标价值的算法过于乐观的高估了一些情况下的行为价值，导致算法会将次优行为价值一致认为最优行为价值，最终不能收敛至最佳价值函数。一种使用双价值网络的 DDQN(double deep Q network) 被认为较好地解决了这个问题。该算法使用两个架构相同的近似价值函数，其中一个用来根据策略生成交互行为并随时频繁参数 (θ)，另一个则用来生成目标价值, 其参数 (θ^-) 每隔一定的周期进行更新。该算法绝大多数流程与 DQN 算法一样，只是在更新目标价值时使用公式 (6.20)：

$$Q_{target}(S_t, A_t) = R_t + \gamma Q\left(S'_t, \underset{a'}{max}\ Q(S'_t, a';\ \theta);\ \theta^-\right) \tag{6.20}$$

该式表明，DDQN 在生成目标价值时使用了生成交互行为并频繁更新参数的价值网络 $Q(\theta)$，在这个价值网络中挑选状态 S' 下最大价值对应的行为 A'_t，随后再用状态行为对 (S'_t, A'_t) 代入目标价值网络 $Q(\theta^-)$ 得出目标价值。实验表明这样的更改比 DQN 算法更加稳定，更容易收敛值最优价值函数和最优策略。在编程实践环节，我们将实现 DQN 和 DDQN。

在使用神经网络等深度学习技术来进行价值函数近似时，有可能会碰到无法得到预期结果的情况。造成这种现象的原因很多，其中包括基于 TD 学习的算法使用引导 (bootstrapping) 数据，非线性近似随机梯度下降落入局部最优值等，也和 ϵ-贪婪策略的 ϵ 的设置有关。此外深度神经网络本身也有许多训练技巧，包括学习率的设置、网络架构的设置等。这些设置参数有别于近似价值函数本身的参数，一般称为超参数 (super parameters)。如何设置和调优超参数目前仍没有一套成熟的理论来指导，得到一套完美的网络参数有时需要多次的实践的并对训练结果进行有效的观察分析。

6.5 编程实践：基于 PyTorch 实现 DQN 求解 PuckWorld 问题

在构建近似价值函数的实践中，PyTorch 框架提供了非常方便的一整套解决方案。PyTorch 既可以像 numpy 那样进行张量计算，也可以很轻易地搭建复杂的神经网络。从本章开始将会较多的使用 PyTorch 提供的功能进行深度强化学习的编程实践。PyTorch 的官方网站将其描述为：优先基于 Python 的深度学习框架，可以进行张量和动态神经网络的运算。深度学习领域的张量 (tensor) 这个概念其实并不难理解，我们都知道标量 (scalar) 这个概念，标量指的是一个没有方向、只有大小的数据，数学上通常就用一个数值来表示，它没有维度或者说是 0 维的；我们熟悉的矢量又称向量 (vector)，它既有大小又有方向，在数学计算时通常用一组排列在一行或者一列的数值表示，它是 1 维的；此外矩阵 (matrix) 也是大家较为熟悉的一个数据表现或转换形式，它是有多行多列的数据组合在一起的，它是 2 维的；而张量则是 3 维或者 3 维以上的数据表现或转换形式。在深度学习和强化学习领域，我们要处理的数据通常很少是 0 维的标量，经常是 1

维的向量和 2 维的矩阵，例如描述一个状态的特征值组合在一起就是一个向量；描述一个五子棋盘当前状态则需要一个 2 维的矩阵来完整描述；描述一个彩色图片则要描述图片上每一个像素点的红绿蓝三种颜色的具体数值，此时就需要用一个 3 维张量来描述了；如果要记录用来处理二维图像的卷积神经网络的一个隐藏层所有权重参数，则需要一个 4 维的张量，因为一个隐藏层包括多个通道，每一个通道需要记录一个 3 维的卷积核矩阵，这是因为每一个神经元需要与前一个隐藏层所有通道的对应神经元相连接。张量的运算规则与矢量以及矩阵运算规则类似，在 PyTorch 看来，矩阵、向量甚至标量都被认为张量来对待，所不同的就是实现规定好它们的维度和尺寸信息，这里的维度和尺寸信息可以称为是形态 (shape) 或尺寸 (size)。在编程实践中，养成经常查看张量形态的习惯对于编写正确有效的代码是非常有帮助的。

PyTorch 提供的张量运算方法和索引方法与 numpy 非常接近，然而 PyTorch 的强大之处并不仅体现在张量运算上，它还提供了自动计算梯度的功能。梯度是一个比较难懂的概念，梯度的计算也是一个较复杂的运算过程。在最新的版本 (0.4) 中，张量对象本身就可以具备自动计算梯度的功能。有了这个功能，在构建基于梯度运算的神经网络或其它模型时误差的反向传播和参数更新都可以自动的完成，读者可以把精力集中在模型的构建和训练方法的设计上。在构建神经网络模型时 PyTorch 支持动态图，意味着可以在实际数据运算时根据前一个节点的结果来动态调整网络中张量流转的路线。PyTorch 官网提供了非常好的教程帮助使用者理解这些设计理念并快速上手，建议对 PyTorch 不熟悉的读者在阅读本书后续内容前先登录其官网，对 PyTorch 有一个基本的认识。

本节的编程实践将使用 PyTorch 库构建一个简单的 2 层神经网络，将其作为近似价值函数应用于 DQN 和 DDQN 算法中解决本章一开始提到的 PuckWorld 问题。在 DQN 中状态由特征组成的一维向量表示，价值由一个接受状态特征为输入的函数来表示，在实现基于 DQN 的个体前，先使用 PyTorch 提供的功能快速实现 2 层神经网络表示的近似价值函数，再实现 DQN 和 DDQN 算法并用它来解决本章一开始提到的 PuckWorld 问题。

6.5.1 基于神经网络的近似价值函数

我们要设计的这个近似价值函数基类针对的是 gym 中具有连续状态、离散行为的环境，在近似价值函数的构架类型中选择接受状态特征作为输入，输出多个行为对应的行为价值的第三种架构 (图 6.2)。近似价值函数的核心功能就是根据状态来得到该状态下所有行为对应的价值，进而为策略提供价值依据并计算目标价值。要构建这样一个基于神经网络的近似价值函数，需要知道对应强化问题状态的特征数、行为的个数，如此这个网络的输入层和输出层的结构就确定了，此外我们还需要指定隐藏层的个数以及每一个隐藏层包含的神经元的数量。本例设计的神经网络只包含一个隐藏层，默认为 32 个神经元。首先导入一些必要的库：

```
import numpy as np
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
import copy
```

可以设计构造函数如下的 NetApproximator 类：

```
class NetApproximator(nn.Module):
    def __init__(self, input_dim = 1, output_dim = 1, hidden_dim = 32):
        '''近似价值函数
        Args:
            input_dim: 输入层的特征数 int
            output_dim: 输出层的特征数 int
        '''
        super(NetApproximator, self).__init__()
        self.linear1 = torch.nn.Linear(input_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, output_dim)
```

NetApproximator 类继承自 nn.Module 类，后者是 PyTorch 中一个非常关键的类，几乎所有的模型都继承自这个类，而重写这个类只需要重写其构造函数和相应的前向运算 (forward) 方法。NetApproximator 类的构造函数中声明了两个线性变换，并没有定义实现神经元整合功能的非线性单元，这是因为这两个线性单元都是包含需要训练的参数的。由于非线性整合功能 PyTorch 本身提供，且不需要参数，故而不需要成为类的成员变量。

NetApproximator 类另一个重写的方法 (forward) 代码如下：

```
    def forward(self, x):
        '''前向运算，根据网络输入得到网络输出
        '''
        x = self._prepare_data(x) # 需要对描述状态的输入参数x做一定的处理
        h_relu = F.relu(self.linear1(x)) # 非线性整合函数relu
        # h_relu = self.linear1(x).clamp(min=0) # 实现relu功能的另一种写法
        y_pred = self.linear2(h_relu) # 网络预测的输出
        return y_pred
```

forward 方法接受网络的输入数据，首先对其进行一定的预处理。随后将其送入第一个线性变换单元，得到线性整合结果后再进行一次非线性的 relu 处理，处理的结果送入另一个线性处理单元，其结果作为神经网络的输出，也就是网络预测的当前状态下各个行为的价值。由于神经网络的输出是针对每一个行为的价值，这个价值的范围在整个实数域，第二个线性整合恰好可以通过参数调整映射至整个实数域。在输入数据送入线性整合单元之前的预处理主要是为了数据类型的兼容和对单个采样样本的支持。因为通常从 gym 环境得到的数据类型是基于 numpy 的数组或者是一个 0 维的标量 (当状态的特征只有一个时)。而 PyTorch 神经网络模块处理的数据类型多数是基于 torch.Tensor 或 torch.Variable(在 0.4 版本中 Variable 已整合至 Tensor 中)，后者一般不接受 0 维的标量。

```
    def _prepare_data(self, x, requires_grad = False):
        '''将numpy格式的数据转化为Torch的Variable
        '''
        if isinstance(x, np.ndarray):
            x = torch.from_numpy(x) # 从numpy数组构建张量
        if isinstance(x, int): # 同时接受单个数据
            x = torch.Tensor([[x]])
        x.requires_grad_ = requires_grad
        x = x.float()   # 从from_numpy()转换过来的数据是DoubleTensor形式
        if x.data.dim() == 1: # 如果x是一维的，下一行代码将其转换为2维
            x = x.unsqueeze(0) # torch的nn接受的输入都至少是2维的
        return x
```

该方法除接受输入数据 x 外，还接受一个名为 requires_grad 的参数。如果该参数的值为 True，则将通过配置使得数据同时具备自动梯度计算功能，通常原始输入数据不需要梯度来更新数据本身，因而该参数默认值为 False。如此最简单的神经网络模块就定义好了，由于这个例子规模较小，我们将训练网络、误差计算、梯度反向传播和参数更新工作也作为这个类的一个方法。代码如下：

```
    def fit(self, x, y, criterion=None, optimizer=None,
                    epochs=1, learning_rate=1e-4):
        '''通过训练更新网络参数来拟合给定的输入x和输出y
        '''
        if criterion is None: # 损失的计算依据
            criterion = torch.nn.MSELoss(size_average = False)
        if optimizer is None: # 参数优化器
            optimizer = torch.optim.Adam(self.parameters(), lr = learning_rate)
```

```
        if epochs < 1: # 对参数给定的数据训练的次数
            epochs = 1

        y = self._prepare_data(y, requires_grad = False) # 输出数据一般也不需要
            梯度

        for t in range(epochs):
            y_pred = self.forward(x) # 前向传播
            loss = criterion(y_pred, y) # 计算损失
            optimizer.zero_grad() # 梯度重置，准备接受新梯度值
            loss.backward() # 反向传播时自动计算相应节点的梯度
            optimizer.step() # 更新权重
        return loss # 返回本次训练最后一个epoch的损失(代价)
```

此外，还编写了两个辅助函数 (___call___ 和 clone)。前者使得该类的对象可以向函数名一样接受参数直接返回结果，后者则实现了对自身的复制。代码如下：

```
    def __call__(self, x):
        y_pred = self.forward(x)
        return y_pred.data.numpy()

    def clone(self):
        '''返回当前模型的深度拷贝对象
        '''
        return copy.deepcopy(self)
```

这样一个基于神经网络的近似价值函数就完全设计好了。

6.5.2 实现 DQN 求解 PuckWorld 问题

实现了基于神经网络的近似价值函数后，设计基于 DQN 的个体类就比较简单了。DQN 类仍然继承自先前介绍的 Agent 基类，我们只需重写策略 (policy) 方法和学习方法 (learning_method) 方法就可以了。该类的构造函数代码如下：

```
class DQNAgent(Agent):
    '''使用近似的价值函数实现的Q学习个体
    '''
```

```
    def __init__(self, env: Env = None,
                       capacity = 20000,
                       hidden_dim: int = 32,
                       batch_size = 128,
                       epochs = 2):
        if env is None:
            raise "agent should have an environment"
        super(DQNAgent, self).__init__(env, capacity)
        self.input_dim = env.observation_space.shape[0] # 状态连续
        self.output_dim = env.action_space.n # 离散行为用int型数据0,1,2,..,n表示
        # print("{},{}".format(self.input_dim, self.output_dim))
        self.hidden_dim = hidden_dim
        # 行为网络，该网络用来计算产生行为，以及对应的Q值，参数频繁更新
        self.behavior_Q = NetApproximator(input_dim = self.input_dim,
                              output_dim = self.output_dim,
                              hidden_dim = self.hidden_dim)
        # 计算目标价值的网络，初始时从行为网络拷贝而来，两者参数一致，该网络参数
            不定期更新
        self.target_Q = self.behavior_Q.clone()

        self.batch_size = batch_size  # mini-batch学习一次状态转换数量
        self.epochs = epochs # 一次学习对mini-batch个状态转换训练的次数
```

该类定义了两个基于神经网络的近似价值函数，其中一个行为网络是策略产生实际交互行为的依据，另一个目标价值网络用来根据状态和行为得到目标价值，是计算代价的依据。在一定次数的训练后，要将目标价值网络的参数更新为行为价值网络的参数，下面的代码实现这个功能：

```
    def _update_target_Q(self):
        '''将更新策略的Q网络(连带其参数)复制给输出目标Q值的网络
        '''
        self.target_Q = self.behavior_Q.clone() # 更新计算价值目标的Q网络
```

DQN 生成行为的策略依然是 ϵ-贪婪策略，相应的策略方法代码如下：

```
    def policy(self, A, s, Q = None, epsilon = None):
        '''依据更新策略的价值函数(网络)产生一个行为
```

```
        '''
        Q_s = self.behavior_Q(s) # 基于NetApproximator实现了__call__方法
        rand_value = random() # 生成0与1之间的随机数
        if epsilon is not None and rand_value < epsilon:
            return self.env.action_space.sample()
        else:
            return int(np.argmax(Q_s))
```

DQN 的学习方法 (learning_method) 方法比较简单，其核心就是根据当前状态特征 S_0 依据行为策略生成一个与环境交互的行为 A_0，交互后观察环境，得到奖励 R_1，下一状态的特征 S_1，以及状态序列是否结束。随后将得到的状态转换纳入记忆中。在每一个时间步内，只要记忆中的状态转换够多，都随机从中提取一定量的状态转换进行基于记忆的学习，实现网络参数的更新：

```
    def learning_method(self, gamma = 0.9, alpha = 0.1, epsilon = 1e-5,
                        display = False, lambda_ = None):
        self.state = self.env.reset()
        s0 = self.state # 当前状态特征
        if display:
            self.env.render()
        time_in_episode, total_reward = 0, 0
        is_done = False
        loss = 0
        while not is_done:
            s0 = self.state # 获取当前状态
            a0 = self.perform_policy(s0, epsilon) # 基于行为策略产生行为
            s1, r1, is_done, info, total_reward = self.act(a0) # 与环境交互
            if display:
                self.env.render()

            if self.total_trans > self.batch_size: # 记忆中的状态转换足够
                loss += self._learn_from_memory(gamma, alpha) # 从记忆学习
            time_in_episode += 1

        loss /= time_in_episode
        if display:
            print("epsilon:{:3.2f},loss:{:3.2f},{}".format(epsilon,loss,self.
                experience.last_episode))
        return time_in_episode, total_reward
```

从上面的代码可以看出，对于 DQN 来说，整个学习方法最关键的地方就是从记忆学习 (_learn_from_memory) 这个方法。该方法的实现如下：

```
    def _learn_from_memory(self, gamma, learning_rate):
        trans_pieces = self.sample(self.batch_size)  # 随机获取记忆里的状态转换
        states_0 = np.vstack([x.s0 for x in trans_pieces])
        actions_0 = np.array([x.a0 for x in trans_pieces])
        reward_1 = np.array([x.reward for x in trans_pieces])
        is_done = np.array([x.is_done for x in trans_pieces])
        states_1 = np.vstack([x.s1 for x in trans_pieces])

        # 准备训练数据
        X_batch = states_0
        y_batch = self.target_Q(states_0)  # 得到numpy格式的结果

        Q_target = reward_1 + gamma * np.max(self.target_Q(states_1), axis=1)*\
            (~ is_done) # is_done则Q_target==reward_1

        # 取消下列代码行的注释则变为DDQN算法
        # 行为a'从行为价值网络中得到
        # a_prime = np.argmax(self.behavior_Q(states_1), axis=1).reshape(-1)
        # (s',a')的价值从目标价值网络中得到
        # Q_states_1 = self.target_Q(states_1)
        # temp_Q = Q_states_1[np.arange(len(Q_states_1)), a_prime]
        # (s,a)的目标价值根据贝尔曼方程得到
        # Q_target = reward_1 + gamma * temp_Q * (~ is_done)
        # is_done则Q_target==reward_1
        ## DDQN算法部分尾部

        y_batch[np.arange(len(X_batch)), actions_0] = Q_target

        # 训练行为价值网络，更新其参数
        loss = self.behavior_Q.fit(x = X_batch,
                                   y = y_batch,
                                   learning_rate = learning_rate,
                                   epochs = self.epochs)

```

```
        mean_loss = loss.sum().data[0] / self.batch_size
        # 可根据需要设定一定的目标价值网络参数的更新频率
        self._update_target_Q()
        return mean_loss
```

该方法顺带实现了 DDQN，根据注释应该不难理解。这样一个 DQN 和 DDQN 算法就完成了，可以将 DQNAgent 类与前一章实现的其它 Agent 类一起放在文件 agents.py 中。现在使用下面的代码来观察其在 PuckWorld 环境中的表现如何。

```
import gym
from puckworld import PuckWorldEnv
from agents import DQNAgent
from utils import learning_curve

env = PuckWorldEnv()
agent = DQNAgent(env)

data = agent.learning(gamma=0.99,
                epsilon = 1,
                decaying_epsilon = True,
                alpha = 1e-3,
                max_episode_num = 100,
                display = False)

learning_curve(data, 2, 1, title="DQNAgent performance on PuckWorld",
                x_name="episodes", y_name="rewards of episode")
```

在运行上述代码后将得到类似如图 6.15 所示的结果。可以看出使用 DQN 算法较为成功的解决了 PuckWorld 问题，个体仅通过为数不多次的完整状态序列就可以较为迅速的跟踪靠近目标小球，并稳定在一个较高的水平上。读者可以在训练一定次数后，通过下面的代码来观察交互界面下拥有 DQN 算法的个体的表现：

```
data = agent.learning(gamma=0.99, #衰减引子
                epsilon = 1e-5, # 近似完全贪婪
                decaying_epsilon = False,
                alpha = 1e-5, # 不学习
                max_episode_num = 20, # 观察次数
```

```
6                 display=True) # 显示交互界面
```

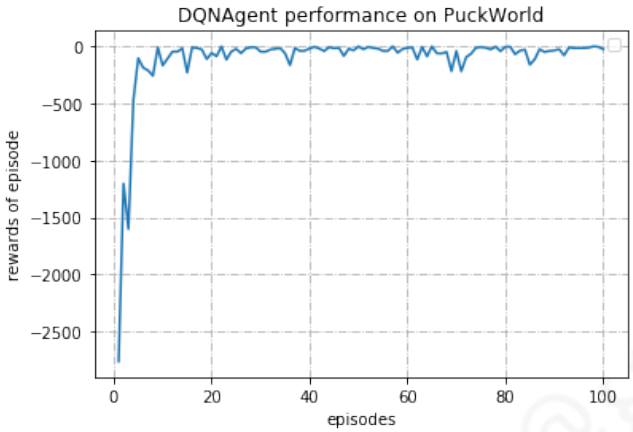


图 6.15: DQN 算法在 PuckWorld 环境中的表现

读者也可以通过设置不同的超参数，例如神经网络隐藏层的神经元个数、学习率、衰减因子、甚至目标价至网络参数更新的频率等值来观察个体的表现差异，从中体会基于深度学习的强化学习如何进行模型调优。