1. 背景

在前面的学习中，state value 和 action value 都是以表格的形式来进行表示的，例如：

	a_1	a_2	a_3	a_4	a_5
s_1	$q_\pi(s_1, a_1)$	$q_\pi(s_1, a_2)$	$q_\pi(s_1, a_3)$	$q_\pi(s_1, a_4)$	$q_\pi(s_1, a_5)$
⋮	⋮	⋮	⋮	⋮	⋮
s_9	$q_\pi(s_9, a_1)$	$q_\pi(s_9, a_2)$	$q_\pi(s_9, a_3)$	$q_\pi(s_9, a_4)$	$q_\pi(s_9, a_5)$

每个 action value 有两个索引 s 和 a，对应着表格的行和列

- 使用表格的优点在于：表格非常直观且易于分析
- 使用表格的缺点在于：难以处理较大或连续的 state/action space，表现在两个方面：（1）存储 （2）泛化能力

因此我们引入函数近似，目的在于用函数来近似模拟 state 与 state value 的映射关系，如下所示：

- Suppose there are one-dimensional states $s_1, \ldots, s_{|\mathcal{S}|}$.
- Their state values are $v_\pi(s_1), \ldots, v_\pi(s_{|\mathcal{S}|})$, where π is a given policy.
- Suppose $|\mathcal{S}|$ is very large and we hope to use a simple curve to approximate these dots to save storage.

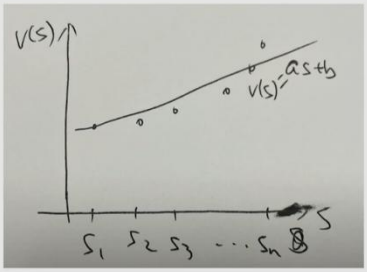


Figure: An illustration of function approximation of samples.

最简单的情况是：使用一条直线来拟合这些点，表示为：

$$\hat{v}(s, w) = as + b = [s, 1]\begin{bmatrix} a \\ b \end{bmatrix} = \phi^T(s)w$$

- w是参数向量
- $\phi(s)$是 s 的特征向量

使用这种方式存储的好处在于：

- 使用表格形式存储，我们需要存储$|S|$个 state value；现在我们只需要存储两个参数 a 和 b
- 我们可以随时用状态 s 的值，计算$\phi^T(s)w$，得到 state value 的近似值
- 但这样做会造成不能精准表示 state value，这也是为什么它被称为值近似

采用更高阶的曲线来进行拟合（例如使用二阶），表示为：

$$\hat{v}(s,w) = as^2 + bs + c = [s^2, s, 1]\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \phi^T(s)w$$

随着$\phi(s)$和w的维度升高，精度也会提高，但是我们要存储的值也会增多；$\hat{v}(s,w)$关于$\phi(s)$是非线性的，但是它关于w是线性的，非线性包含在$\phi(s)$中

值函数近似的核心思想在于：使用带参函数来近似拟合 state value 或 action value

它的好处在于：

（1） 存储：参数向量w的维度可能会远小于状态数$|S|$

（2） 泛化性：当访问到某个状态 s 时，对其 state value 的更新会造成参数向量w的更新，从而导致其他状态的 state value 的近似值发生变化

2. 目标函数介绍

我们的目标是找到最优的参数向量w，使得对每个状态 s，$\hat{v}(s,w)$可以很好地拟合$v_\pi(s)$，定义目标函数为：

$$\min_w J(w) = E[(v_\pi(S) - \hat{v}(S,w))^2]$$

其中 S 为一个随机变量，取值集合为所有状态，涉及到计算期望，那么 S 的概率分布会影响到期望的计算，怎么选择 S 的概率分布呢？

（1） 最简单、最直接的分布：均匀分布，即每个 state 都是同样重要的，则有：

$$J(w) = E[(v_\pi(S) - \hat{v}(S,w))^2] = \frac{1}{|S|}\sum_{s\in S}(v_\pi(s) - \hat{v}(s,w))^2$$

缺点：可能并不是所有状态都同等重要，在给定策略下，某些状态可能很少会被访问

（2） stationary distribution：稳态分布，它描述了马尔科夫过程的 long-run behavior

我们使用$\{d_\pi(s)\}_{s\in S}$来表示在策略π下状态 s 的权重，则有$d_\pi(s) \geq 0\ and\ \sum_{s\in S} d_\pi(s) = 1$

则目标函数可以写成：

$$J(w) = E[(v_\pi(S) - \hat{v}(S,w))^2] = \sum_{s\in S} d_\pi(s)(v_\pi(s) - \hat{v}(s,w))^2$$

$d_\pi(s)$表示达到平稳状态后 agent 访问状态 s 的概率，它表示一个状态的权重，当这个状态越被频繁地访问，它的值也就越大，权重也就越高，它的计算方式可以近似为：

$$d_\pi(s) \approx \frac{n_\pi(s)}{\sum_{s'\in S} n_\pi(s')}$$

其中$n_\pi(s)$表示在策略π下进行一条非常长的 episode，状态 s 被访问的次数

如图所示，进行一条长度为 1000 的 episode，各状态的$d_\pi(s)$变化如图所示：

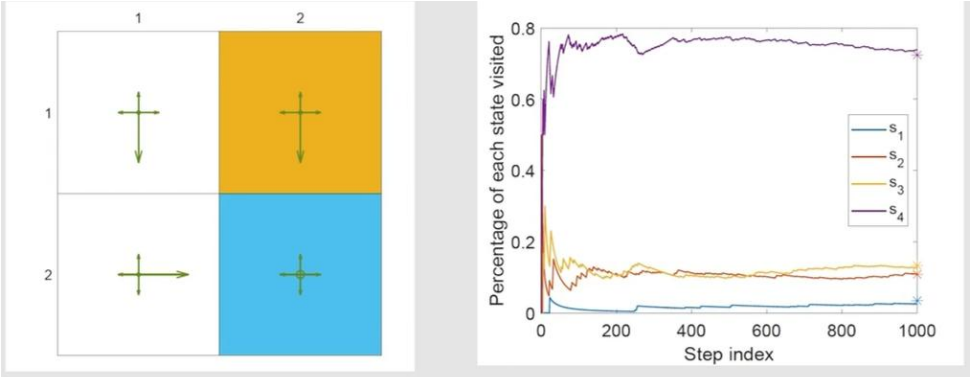


Figure: Long-run behavior of an ϵ-greedy policy with $\epsilon = 0.5$.

而$d_\pi(s)$的近似收敛值是可以计算出来的，因为其满足以下等式：

$$d_\pi^T = d_\pi^T P_\pi$$

其中P_π同贝尔曼公式中的P_π，$\{P_{ij}\}$表示从状态s_i到状态s_j的 probability

在该例子中，有P_π为

$$P_\pi = \begin{bmatrix} 0.3 & 0.1 & 0.6 & 0 \\ 0.1 & 0.3 & 0 & 0.6 \\ 0.1 & 0 & 0.3 & 0.6 \\ 0 & 0.1 & 0.1 & 0.8 \end{bmatrix}$$

计算得到

$$d_\pi = [0.0345, 0.1084, 0.1330, 0.7241]^T$$

3. 优化算法和函数选择

为了最小化目标函数$J(w)$，我们采用梯度下降法进行求解：

$$w_{k+1} = w_k - \alpha_k \nabla_w J(w_k)$$

又有

$$\begin{gathered} \nabla_w J(w_k) = \nabla_w E[(v_\pi(S) - \hat{v}(S,w))^2] \\ = E[\nabla_w (v_\pi(S) - \hat{v}(S,w))^2] \\ = 2E[(v_\pi(S) - \hat{v}(S,w))(-\nabla_w \hat{v}(S,w))] \\ = -2E[(v_\pi(S) - \hat{v}(S,w))\nabla_w \hat{v}(S,w)] \end{gathered}$$

而 true gradient 的计算涉及到了期望的计算，因此我们使用 stochastic gradient 替换 true gradient：

$$w_{t+1} = w_t + \alpha_t (v_\pi(s_t) - \hat{v}(s_t, w_t)) \nabla_w \hat{v}(s_t, w_t)$$

在该式子中，将$2\alpha_t$替换为α_t，使式子更加简洁

但是这个式子无法应用，因为实际上我们是不知道$v_\pi(s_t)$的，我们可以用近似估计值来代替它：

（1） MC 算法：g_t表示从状态s_t出发得到的一条 episode 的 discounted return，用g_t来近似估计$v_\pi(s_t)$，得到：

$$w_{t+1} = w_t + \alpha_t (g_t - \hat{v}(s_t, w_t)) \nabla_w \hat{v}(s_t, w_t)$$

（2） TD 算法：使用 TD target $r_{t+1}+\gamma\hat{v}(s_{t+1},w_t)$来近似估计$v_\pi(s_t)$，得到：

$$w_{t+1}=w_t+\alpha_t[r_{t+1}+\gamma\hat{v}(s_{t+1},w_t)-\hat{v}(s_t,w_t)]\nabla_w\hat{v}(s_t,w_t)$$

该算法的伪代码为：

Initialization: A function $\hat{v}(s,w)$ that is a differentiable in w. Initial parameter w_0.

Aim: Approximate the true state values of a given policy π.

For each episode generated following the policy π, do

For each step (s_t,r_{t+1},s_{t+1}), do

In the general case,

$$w_{t+1}=w_t+\alpha_t\left[r_{t+1}+\gamma\hat{v}(s_{t+1},w_t)-\hat{v}(s_t,w_t)\right]\nabla_w\hat{v}(s_t,w_t)$$

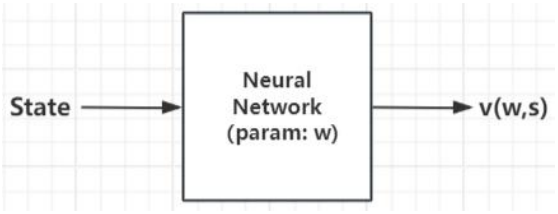
如何去选取函数$\hat{v}(s,w)$来进行近似拟合？

- 方法一：以前经常使用的，关于w为线性函数：

$$\hat{v}(s,w)=\phi^T(s)w$$

如何选取特征向量$\phi(s)$：基于多项式、基于傅里叶...

- 方法二：目前广泛使用的，神经网络，来模拟非线性函数近似



如果为线性函数拟合，即

$$\hat{v}(s,w)=\phi^T(s)w$$

则有：

$$\nabla_w\hat{v}(s,w)=\phi(s)$$

则更新过程为：

$$w_{t+1}=w_t+\alpha_t[r_{t+1}+\gamma\phi^T(s_{t+1})w_t-\hat{v}(s_t,w_t)]\phi(s_t)$$

线性函数拟合最大的缺点在于：难以选择合适的特征向量$\phi(s)$

考虑这样一种特征向量：$\phi(s)$只在其索引位置上的元素为 1，其它位置元素均为 0，例如$\phi(s_1) = [1, 0, 0, \ldots, 0]$，$\phi(s_2) = [0, 1, 0, \ldots, 0]$，表示为$\phi(s) = e_s \in R^{|S|}$，则有：

$$\hat{v}(s, w) = \phi^T(s)w = w(s)$$

其中$w(s)$是参数向量w中，s 对应索引位置上的元素值，参数向量$w = [w(s_1), w(s_2), w(s_3), \ldots, w(s_{|S|})]$

则 TD-Linear 算法可以表示为：

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma\phi^T(s_{t+1})w_t - \phi^T(s_t)w_t]\phi(s_t)$$

而$\phi(s_t) = e_{s_t}$，则有：

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma w_t(s_{t+1}) - w_t(s_t)]e_{s_t}$$

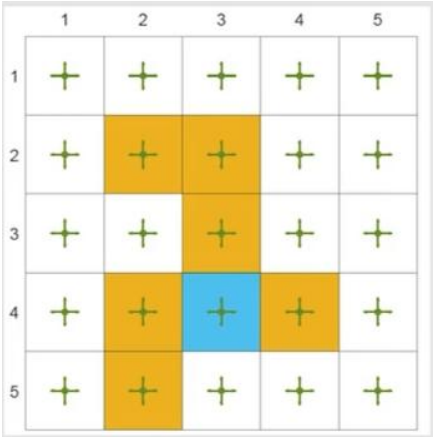
所以参数向量w中只有对应位置s_t的参数被更新，其它的参数没有被更新，单独将被更新的那个参数拿出来，可以写为：

$$w_{t+1}(s_t) = w_t(s_t) + \alpha_t[r_{t+1} + \gamma w_t(s_{t+1}) - w_t(s_t)]$$

可以看到它与之前的 tabular TD 算法是一样的

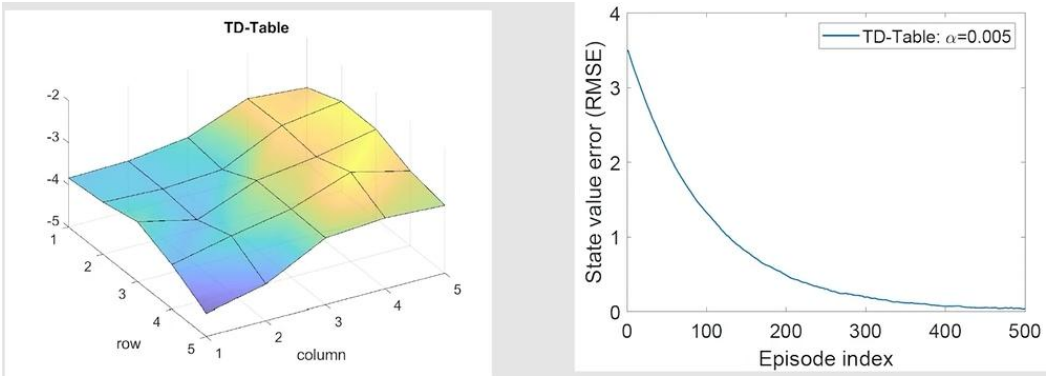
例子：

给定一个 5x5 的网格世界，给定策略为在每个 state 进行任意一个 action 的概率相同，均为 0.2，$r_{boundary} = r_{forbidden} = -1, r_{target} = 1, \gamma = 0.9$



根据贝尔曼公式，计算出真实的 state value，并画成 3D 图，结果如下：

使用 tabular TD 算法估计 state value 值，结果如下：

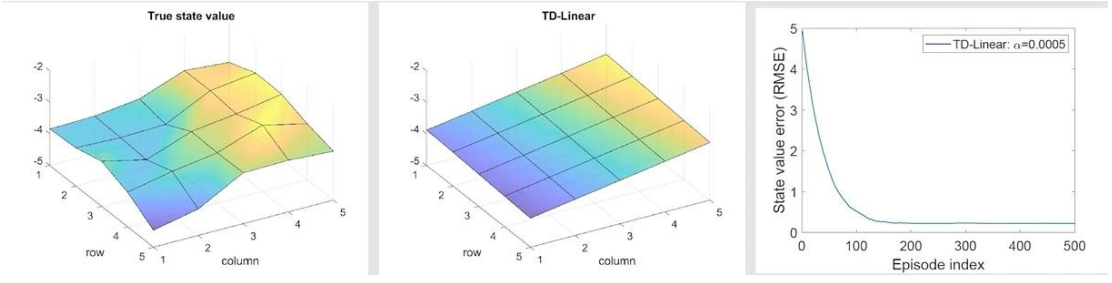


使用 TD-Linear 算法进行估计：

取特征向量为：$\phi(s) = [1, x, y]^T \in R^3$，则有：

$$\hat{v}(s, w) = \phi^T(s)w = [1, x, y]\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = w_1 + w_2 x + w_3 y$$

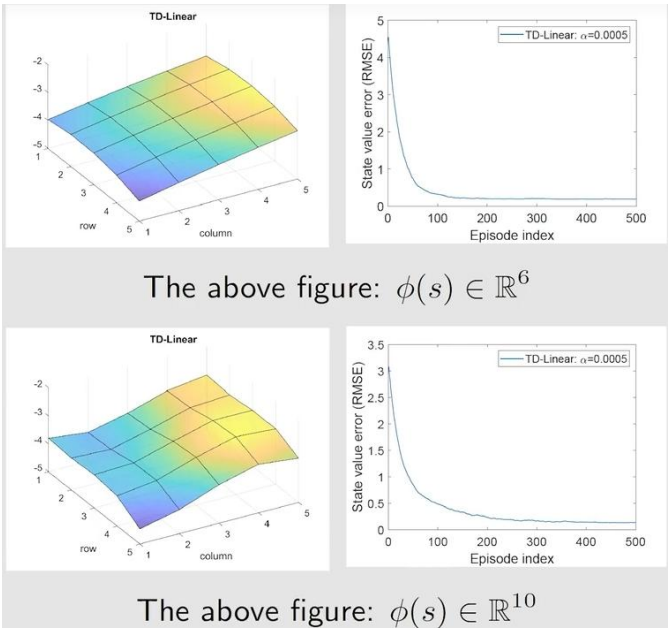
可以看到，它是一个平面，它的近似估计结果如下：



可以看到，趋势是对的，但是不能很好地拟合，如果我们使用更高阶的特征向量进行拟合，取 $\phi(s) = [1, x, y, x^2, y^2, xy]^T \in R^6$，则有：

$$\hat{v}(s, w) = \phi^T(s)w = w_1 + w_2 x + w_3 y + w_4 x^2 + w_5 y^2 + w_6 xy$$

我们还可以取更高维的特征向量 $\Phi(s) = [1, x, y, x^2, y^2, xy, x^3, y^3, x^2y, xy^2]^T \in R^{10}$，它们的结果如下所示：



4. 使用 Sarsa 值函数近似估计 action value

该算法的求解步骤为：

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)]\nabla_w\hat{q}(s_t, a_t, w_t)$$

和刚才的 TD 算法相似，只是将 state value 的部分都替换成了 action value

该算法的伪代码为：

Aim: Search a policy that can lead the agent to the target from an initial state-action pair (s_0, a_0).

For each episode, do

If the current s_t is not the target state, do

Take action a_t following $\pi_t(s_t)$, generate r_{t+1}, s_{t+1}, and then take action a_{t+1} following $\pi_t(s_{t+1})$

Value update (parameter update):

$$w_{t+1} = w_t + \alpha_t\left[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)\right]\nabla_w\hat{q}(s_t, a_t, w_t)$$

Policy update:

$$\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1) \quad \text{if } a = \arg\max_{a\in\mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$$

$$\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|} \quad \text{otherwise}$$

5. 使用 Q-learning 值函数近似估计 action value

该算法的求解步骤为：

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma \max_{a \in A(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t)]\nabla_w \hat{q}(s_t, a_t, w_t)$$

和刚才的 Sarsa 算法相似，只是 TD target 由 Sarsa 算法中的$r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t)$

转为了$r_{t+1} + \gamma \max_{a \in A(s_{t+1})} \hat{q}(s_{t+1}, a, w_t)$

该算法的伪代码为（on-policy 版本）：

Initialization: Initial parameter vector w_0. Initial policy π_0. Small $\varepsilon > 0$.

Aim: Search a good policy that can lead the agent to the target from an initial state-action pair (s_0, a_0).

For each episode, do

If the current s_t is not the target state, do

Take action a_t following $\pi_t(s_t)$, and generate r_{t+1}, s_{t+1}

Value update (parameter update):

$$w_{t+1} = w_t + \alpha_t\Big[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} \hat{q}(s_{t+1}, a, w_t) - \hat{q}(s_t, a_t, w_t)\Big]\nabla_w \hat{q}(s_t, a_t, w_t)$$

Policy update:

$$\pi_{t+1}(a|s_t) = 1 - \frac{\varepsilon}{|\mathcal{A}(s)|}(|\mathcal{A}(s)| - 1) \quad \text{if} \quad a = \arg\max_{a \in \mathcal{A}(s_t)} \hat{q}(s_t, a, w_{t+1})$$

$$\pi_{t+1}(a|s_t) = \frac{\varepsilon}{|\mathcal{A}(s)|} \quad \text{otherwise}$$

6. Deep Q-learning or deep Q-network（DQN）

该算法的目标函数为：

$$J(w) = E[\left(R + \gamma \max_{a \in A(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)\right)^2]$$

该算法要做的是通过优化参数向量w，使得$J(w)$取得最小值

这个式子实际上就是 Bellman optimal error，因为 Q-learning 算法实际上是在求解贝尔曼最优公式：

$$q(s, a) = E\left[R_{t+1} + \gamma \max_{a \in A(S_{t+1})} q(S_{t+1}, a) \,|S_t = s, A_t = a\right], \forall s, a$$

要取得目标函数$J(w)$的最小值，我们采用 gradient descent 的方法，但是要对参数向量w求导，函数$J(w)$中总共有两个部分包含w，我们固定前面一个w，将它视为常量，只对后面一个w进行求导，令：

$$y = R + \gamma \max_{a \in A(S_{t+1})} \hat{q}(S', a, w)$$

为了实现这一想法，我们引入两个 network：

- main network：$\hat{q}(s', a, w)$
- target network：$\hat{q}(s', a, w_T)$

使用这两个 network 把目标函数中的两个$\hat{q}$区分开，得到：

$$J = E[\left(R + \gamma \max_{a \in A(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right)^2]$$

其中w_T是 target network 的参数，则对w求导得到：

$$\nabla_w J = E\left[\left(R + \gamma \max_{a \in A(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right) \nabla_w \hat{q}(S, A, w)\right]$$

要实现这一算法，我们在每次迭代中从 replay buffer 中取得一小块采样$\{(s, a, r, s')\}$，对参数向量w进行更新，使得向 main network 中输入(s, a)得到$\hat{q}(s, a, w)$，能更接近$y_T = r + \gamma \max_{a \in A(s')} \hat{q}(s', a, w_T)$。在经过一定次数的迭代后，使用$w$来更新$w_T$

7. experience replay：经验回放

我们在收集 experience sample 的时候，一定是有先后顺序的，但是我们使用这些 experience sample 的时候可以不按照它的先后顺序来使用，而是将其打乱放在一个集合中，这个集合被称为 replay buffer：$B = \{(s, a, r, s')\}$，从这个集合中对 experience sample 采样用来训练就被称为 experience replay，采样应该服从均匀分布，即每个 experience sample 被访问的概率都应该是相同的

为什么在 tabular Q-learning 算法中没有涉及到 experience replay 呢？

从第七章的学习中我们可以知道，tabular Q-learning 算法的更新步骤为：

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - \left[r_{t+1} + \gamma \max_{a \in A} q_t(s_{t+1}, a)\right]\right]$$

可以看到，更新过程不涉及计算期望值，因此不需要(s, a)的分布，也就不需要 experience replay

而 deep Q-learning 算法的更新步骤为：

$$w_{t+1} = w_t - \alpha_t \nabla_w J$$

$$\nabla_w J = E\left[\left(R + \gamma \max_{a \in A(S')} \hat{q}(S', a, w_T) - \hat{q}(S, A, w)\right) \nabla_w \hat{q}(S, A, w)\right]$$

更新过程涉及计算期望值，而要求期望值需要知道(s, a)的分布，所以对于 deep Q-learning 算法来说，experience replay 是必要的

deep Q-learning 算法的伪代码为（off-policy 版本）：

Aim: Learn an optimal target network to approximate the optimal action values from the experience samples generated by a behavior policy π_b.

Store the experience samples generated by π_b in a replay buffer $\mathcal{B} = \{(s, a, r, s')\}$

For each iteration, do

Uniformly draw a mini-batch of samples from $\mathcal{B}$

For each sample (s, a, r, s'), calculate the target value as $y_T = r + \gamma \max_{a \in \mathcal{A}(s')} \hat{q}(s', a, w_T)$, where w_T is the parameter of the target network

Update the main network to minimize $(y_T - \hat{q}(s, a, w))^2$ using the mini-batch $\{(s, a, y_T)\}$

Set $w_T = w$ every C iterations

- 为什么没有 policy update 步骤？

因为该算法是 off-policy 的，当我们近似求得所有(s, a)对的 action value 后，可以直接一步得到 optimal policy

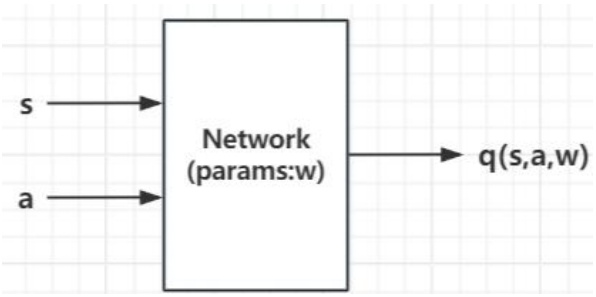
- 为什么不用我们推导出来的公式进行更新？

因为我们推导出来的公式过于底层，现在的神经网络可以批量计算损失，进行训练，在每一批次中，我们首先计算出来 label，即$y_T = r + \gamma \max_{a \in A(s')} \hat{q}(s', a, w_T)$，然

后计算其与神经网络的输出$\hat{q}(s,a,w)$的损失函数，对参数向量进行更新

- network 的输入和输出与原 DQN 论文中的输入输出是不同的

在该例子中，输入为(s,a)，输出为$\hat{q}(s,a,w)$



而在原论文中，输入为 s，输出为$\hat{q}(s,a_1,w),\hat{q}(s,a_2,w),\ldots,\hat{q}(s,a_{|A(s)|},w)$，这样便于计算$\max_{a\in A(S')}\hat{q}(S',a,w_T)$