强化学习扫盲贴：从Q-learning到DQN

夕小瑶的卖萌屋 2019-09-28

本文转载自知乎专栏「机器学习笔记」，原文作者「余帅」，链接
https://zhuanlan.zhihu.com/p/35882937

1 本文学习目标

1. 复习Q-Learning；
2. 理解什么是值函数近似（Function Approximation）；
3. 理解什么是DQN，弄清它和Q-Learning的区别是什么。

2 用Q-Learning解决经典迷宫问题

现有一个5房间的房子，如图1所示，房间与房间之间通过门连接，编号0到4,5号是房子外边，即我们的终点。我们将agent随机放在任一房间内，每打开一个房门返回一个reward。图2为房间之间的抽象关系图，箭头表示agent可以从该房间转移到与之相连的房间，箭头上的数字代表reward值。

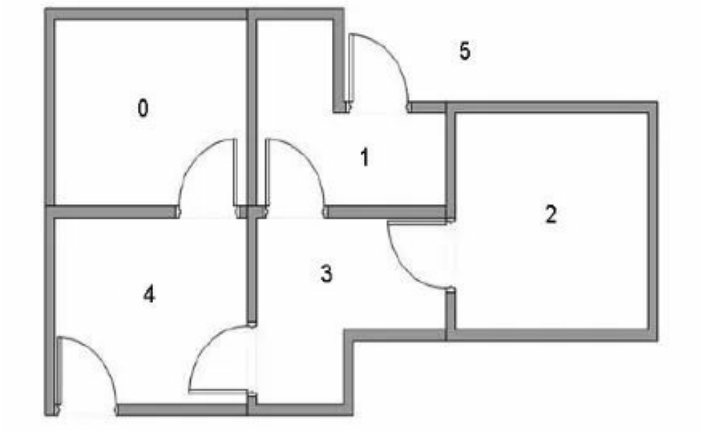


图1 房子原型图

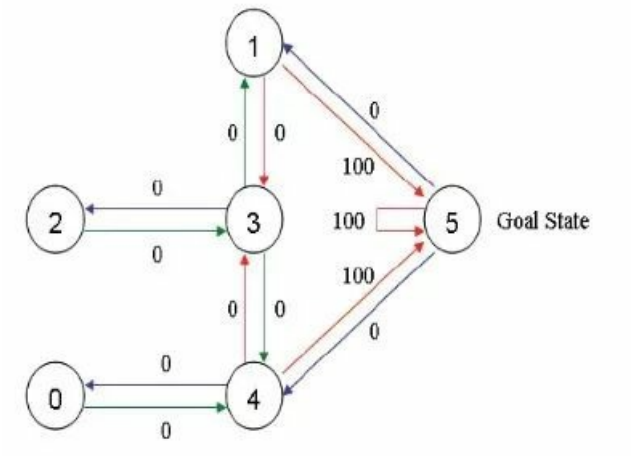


图2 抽象关系图

根据此关系，可以得到reward矩阵为

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Q-Learning是一种off-policy TD方法，伪代码如图所示

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

Q-Learning伪代码

我们首先会初始化一个Q表，用于记录状态-动作对的值，每个episode中的每一步都会根据下列公式更新一次Q表

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right]$$

这里的迷宫问题，每一次episode的结束指的是到达终点状态5。为了简单起见，这里将学习率设为1，更新公式变为

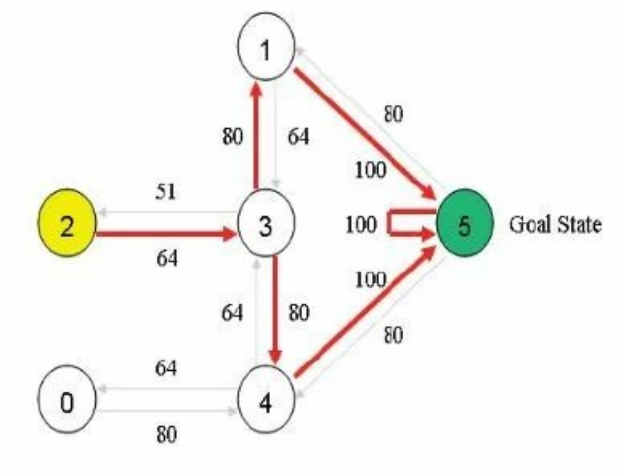
$$Q\left(S_t, A_t\right) \leftarrow R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right)$$

另外，将衰减系数γ设为0.8。Q表初始化为一个5×5的全0矩阵。每次这样更新，最终Q表会收敛到一个矩阵。

最终Q表收敛为

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

因此，也可以得到最优路径如下红色箭头所示



Python代码：

```
import numpy as np
GAMMA = 0.8
Q = np.zeros((6,6))
R=np.asarray([[-1,-1,-1,-1,0,-1],
  [-1,-1,-1,0,-1,100],
  [-1,-1,-1,0,-1,-1],
  [-1,0, 0, -1,0,-1],
  [0,-1,-1,0,-1,100],
  [-1,0,-1,-1,0,100]])
def getMaxQ(state):
   return max(Q[state, :])
def QLearning(state):
   curAction = None
   for action in range(6):
      if(R[state][action] == -1):
         Q[state, action]=0
      else:
         curAction = action
         Q[state,action]=R[state][action]+GAMMA * getMaxQ(curAction)
count=0
while count<1000:
   for i in range(6):
      QLearning(i)
   count+=1
print(Q/5)
```

Q-Learning方法很好的解决了这个迷宫问题，但是这终究只是一个小问题（状态空间和动作空间都很小），**实际情况下，大部分问题都是有巨大的状态空间或者动作空间，想建立一个Q表，内存是绝对不允许的，而且数据量和时间开销也是个问题。**

3 值函数近似与DQN

值函数近似(Function　Approximation)的方法就是为了解决状态空间过大，也称为“维度灾难”的问题。通过用　　函数而不是Q表来表示

，这个函数可以是线性的也可以使非线性的。

$$\hat{v}\left(s, \boldsymbol{w}\right) \approx v_{\pi}\left(s\right) \ or \ \hat{q}\left(s, a, \boldsymbol{w}\right) \approx q_{\pi}\left(s, a\right)$$

其中ω称为“权重”。那怎么把这个权重求出来，即拟合出这样一个合适的函数呢？**这里就要结合机器学习算法里的一些有监督学习算法，对输入的状态提取特征作为输入，通过MC/TD计算出值函数作为输出，然后对函数参数**

行训练，直到收敛。这里主要说的是回归算法，比如线性回归、决策树、**神经网络**等。

这里，就可以引入DQN（Deep Q-Network）了，**实际上它就是Q-Learning和神经网络的结合，将Q-Learning的Q表变成了Q-Network。**

好，现在关键问题来了。这么去训练这个网络呢？换句话说，怎么去确定网络参ω呢？**第一，我们需要一个Loss Function；第二，我们需要足够的训练样本。**

训练样本好说，通过epsilon-greedy策略去生成就好。回忆一下Q-Learning，我们更新Q表是利用每步的reward和当前Q表来迭代的。那么我们可以用这个计算出来的Q值作为监督学习的“标签”来设计Loss Function，我们采用如下形式，即近似值和真实值的均方差

$$J(\boldsymbol{w}) = \mathbb{E}_{\boldsymbol{\pi}}\left[\left(q_{\pi}(s,a) - \hat{q}(s,a,\boldsymbol{w})\right)^2\right]$$

采用随机梯度下降法（SGD）来迭代求解，得到我们想要的，具体公式和过程还请看参考资料，这里不展开了，其实就是求导啦。值得一提的是，上述公式中的q(·)根据不同方法算出来，其形式不一样，比如利用MC，则为（回报）；利用TD(0)，则为

$$R_{t+1} + \gamma \hat{q}\left(s_{t+1}, a_{t+1}, \boldsymbol{w}\right)$$

Q-Learning呢，就是

$$R_{t+1} + \gamma \max \hat{q}\left(s_{t+1}, a_{t+1}, \boldsymbol{w}\right)$$

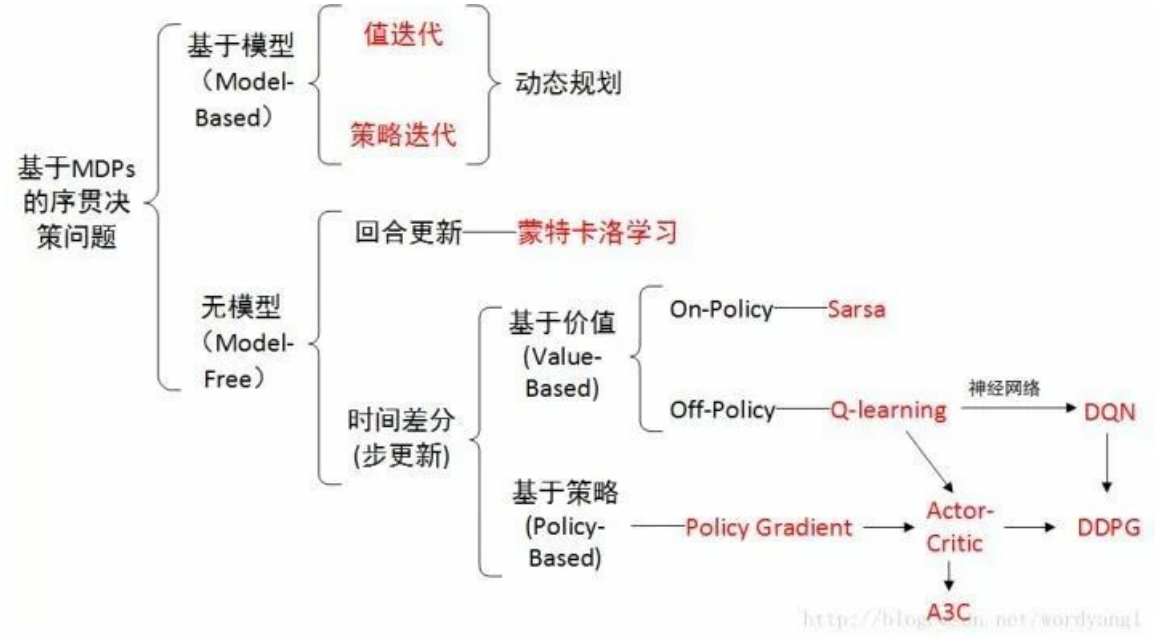
在David Silver的课里，他根据每次更新所参与样本量的不同把更新方法分为增量法（Incremental Methods）和批处理法（Batch Methods）。前者是来一个数据就更新一次，后者是先攒一堆样本，再从中采样一部分拿来更新Q网络，称之为“**经验回放**”，实际上DeepMind提出的DQN就是采用了经验回放的方法。为什么要采用经验回放的方法？因为对神经网络进行训练时，**假设样本是独立同分布的。**而通过强化学习采集到的数据之间存在着关联性，利用这些数据进行顺序训练，神经网络当然不稳定。经验回放可以打破数据间的关联。

最后附上DQN的伪代码

Algorithm 1 Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode 1, M **do** Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in the emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

学到这里，其实可以做一个阶段性总结了，强化学习算法的基本框架可以用下图概括



参考文献

[1]Reinforcement Learning: An Introduction - Chapter 9: On-policy Prediction with Approximation
[2]Reinforcement Learning: An Introduction - Chapter 10: On-policy Control with Approximation
[3]David Silver's RL Course Lecture 6 - Value Function Approximation (video, slides)
[4]DQN从入门到放弃5 深度解读DQN算法
[5]一条咸鱼的强化学习之路6之值函数近似（Value Function Approximation）和DQN

文章已于修改