DQN算法

简介

问题引入：

前面引入的一些强化学习算法存在一个问题，无论是V(s)还是Q(s,a)都是默认创建一个“**表格**”，然后基于表格去更新查询，这种表格对于悬崖漫步简单的问题非常适用，对于具有离散有限的状态数是可控的，但当MDP的状态是连续的，或者是大规模的离散的情况下，如：

- 围棋博弈（10^{170}）
- 直升机、自动驾驶的汽车（连续状态空间）

可见对于这种具有大规模状态的MDP，简单的表格是无法保存对于状态价值函数以及动作价值函数，所以针对这种问题需要解决，现在主要有两个解决思路

- 对状态/动作进行离散化或分桶
- 构建参数化值函数进行估计

对状态/动作进行离散化或分桶，就是将连续的状态离散化，或者大规模离散化的状态部分聚合成新的类，从而降低问题的规模，使用该方法解决问题操作简洁直观、高效、处理许多问题时能够达到较好的效果。但是该问题过于简单地表示价值函数V，可能为每个离散区间假设一个常数值，还有一点需要注意的是可能会产生维度灾难

本章的主要解决方法还是**构建参数化值函数进行估计**

参数化价值函数：

对于MDP的状态价值函数以及动作价值函数，我们可以通过机器学习构建参数化（可学习的）函数来近似函数

$$V_\theta(s) \simeq V^\pi(s)$$
$$Q_\theta(s,a) \simeq Q^\pi(s,a)$$

上式θ是近似函数的参数，可以通过强化学习进行更新，参数化的方法将现有可见的状态泛化到没有见过的状态上，这样得到的参数化函数可以”知其一推其二”

机器学习生来就是根据训练集的数据去拟合建立函数模型的，所以根据现有的V(s)和Q(s,a)来训练出对应的拟合函数使用机器学习来解决非常合适

值函数近似函数的主要形式：

一些函数近似：

- （一般的）线性模型
- 神经网络
- 决策树
- 最近邻
- 傅立叶/小波基底

可微函数（常用）：

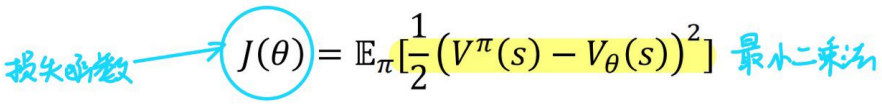
- （一般的）线性模型

- 神经网络

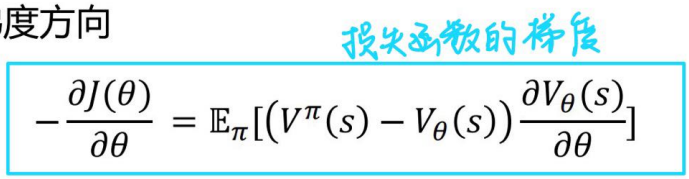
可微意味着可以计算梯度，就可以通过最速下降法来优化构建的参数函数，无论使用什么样的近似函数来估计建模，都希望模型适合非静态、非独立同分布的数据上训练，因为我们的策略目标在变，动作价值函数也在变，所以强化学习相比有监督的学习更不容易收敛，就是因为策略目标是动态的，而有监督学习目标数据标签本身是不会发生改变

下面是基于随机梯度下降（SGD）的值函数近似

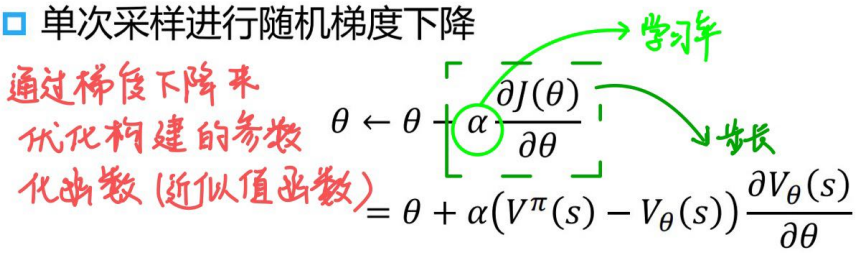
- 目标：找到参数向量θ最小化值函数近似值与真实值之间的均方误差



- 误差减小的梯度方向



- 单次采样进行随机梯度下降



当然这种函数拟合的方法存在一定的精度损失，所以是近似的方法，本章节的**DQN算法可以用来解决连续状态下离散动作的问题**

车杆环境

车杆环境为例，状态值为连续的，动作值是离散的

车杆环境就是一辆小车，智能体的任务是通过左右移动保持车上的杆竖直，若杆的倾斜度数过大，或者车子离初始位置左右的偏离程度过大，或者坚持时间到达 200 帧，则游戏结束。

就是有辆小车，车上一个杆，保持杆尽量不倒，坚持到一定时间

智能体的状态是一个维数为 4 的向量（车的位置、车的速度、杆的角度、杆尖端的速度），每一维都是连续的，其动作是离散的，动作空间大小为 2（向左移动、向右移动）。在游戏中每坚持一帧，智能体能获得分数为 1 的奖励，坚持时间越长，则最后的分数越高，坚持 200 帧即可获得最高的分数。

环境状态空间：

维度	意义	最小值	最大值
0	车的位置	-2.4	2.4
1	车的速度	-Inf	Inf
2	杆的角度	~ -41.8°	~ 41.8°
3	杆尖端的速度	-Inf	Inf

环境动作空间：

标号	动作
0	向左移动小车
1	向右移动小车

DQN

DQN是深度强化学习第一个模型，对于DQN介绍有3个版本，在DQN上进行改进可得**Double DQN**，**Dueling DQN**

在介绍DQN前，回顾一下Q-learning

根据数据更新Q值函数，它不是直接更新策略，而是基于值来选择策略

Q-learning算法本质就是在更新一个参数化的函数$Q_\theta(s,a)$，现在时代（主流）所使用的就是参数化的函数更新，而不再是用一个表格来保存Q值对表格更新

它优化目标是TD，TD误差越小说明Q值越准确，越收敛

所以Q-learining算法的一个关键思想就是构建$Q_\theta(s,a)$函数，直观地想法利用神经网络来构建，但使用深度神经网络会给参数化带来不确定性、不稳定性。这是因为深度神经网络很难被训练，更不用说该函数是在一个动态的数据系统中进行建模，这就使最后的目标值经常会根据现在更新的Q函数发生改变，使之训练不稳定，主要体现在以下方面

不稳定性的体现：

- 不断地采样训练数据$\{(s_t,a_t,s_{t+1},r_t)\}$，是不满足独立同分布的，往往上一个transition和对应的下一个transition是具有相关性的
- $Q_\theta(s,a)$值频繁的更新

所以针对这一问题，需要相应的解决办法进行解决，第一种问题的解决办法是**经验回放**，第二种解决办法是使用双网络结构，评估网络和**目标网络**

经验回放：

在原来的 Q-learning 算法中，每一个数据只会用来更新一次Q值。为了更好地将 Q-learning 和深度神经网络结合，DQN 算法采用了**经验回放**（experience replay）方法，具体做法为维护一个**回放缓冲区**，将每次从环境中采样得到的四元组数据（状态、动作、奖励、下一状态）存储到回放缓冲区中，训练 Q 网络的时候再从回放缓冲区中随机采样若干数据来进行训练。

概括的说，将训练过程的每一步存放在回放缓冲区中，在训练Q网络时再从回放缓冲区中随机采样若干数据进行训练，采样一般服从均匀分布

使用经验回放的好处：

- 使样本满足独立假设。在 MDP 中交互采样得到的数据本身不满足独立假设，因为这一时刻的状态和上一时刻的状态有关。采用经验回放可以打破样本之间的相关性，让其满足独立假设。
- 提高样本效率。每一个样本可以被使用多次，十分适合深度神经网络的梯度学习。

经验池：

回放缓冲区又称经验池，经验池用来存放transition，一个transition为(s_t,a_t,s_{t+1},r_t)

回放容器（replay buffer）为：存储n个transition

如果超过n个transition时，删除最早进入容器的transition（所以容器是n个transition数据类型的队列）

容器容量（buffer capacity）n为一个超参数：这个n一般设置为较大的数如10^5~10^6，具体大小取决于任务

实现：

经验回放池的类，主要包括加入数据、采样数据两大函数

```python
class ReplayBuffer:
    ''' 经验回放池 '''
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)  # 队列,先进先出

    def add(self, state, action, reward, next_state, done):  # 将数据加入buffer
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):  # 从buffer中采样数据,数量为batch_size
        transitions = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*transitions)
        return np.array(state), action, reward, np.array(next_state), done

    def size(self):  # 目前buffer中数据的数量
        return len(self.buffer)
```

数据训练在从经验池采样时一般服从均匀分布，但是使用均匀分布时会遇到这种情况，

如上图所示，马里奥游戏中有普通行走探索的经验和boss战的经验，相比这两种经验哪个比较重要的？当然是打boss的经验，因为行走探索的经验在经验池是非常多的，而打boss时的经验就非常稀少，如果按均匀抽样的话，我们绝大数会抽到行走探索的经验进行学习，这会使得我们闯到boss关时一马平川，但在doss时确很容易失误，因为我们打boss时训练的比较少，这样会陷入恶性的循环，降低学习效率。所以我们更偏向于去学习经验池中很少的关键经验，所以从经验池中抽样时就要非均匀的抽象，去学习更关键的transition，所以下面介绍经验回放中的一个改进——优先经验回放

优先经验回放：

对于上述问题，我们需要一个标准来衡量什么样的数据是关键数据，是我们重点采样和学习的数据

以 Q 函数的值与 Target 值的差异来衡量学习的价值，即

$$p_t = |r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') - Q_\theta(s_t, a_t)|$$

来作为衡量标准，每次数据的学习p_t是有差距的，p_t越大，则该数据对学习越有帮助，因为差距越小说明Q值越逼近越收敛，而训练时训练过度是没必要的，甚至会造成在网络某处发生分散

所以在选择样本时尽量选择p_t大的样本，为了使各样本都有机会被采样，存储 $et = (st, at, st+1, rt, pt + \epsilon)$，其中$\epsilon$为干扰噪音项，防止在计算时概率为0，见下述公式

针对p_t的占比，样本 et 被选中的概率为：

$$P(t) = \frac{p_t^\alpha}{\sum_k p_k^\alpha}$$

（α用于平滑概率，α = 1则完全按照p_t去采样

重要性采样：

按这种方法选择的样本e_t是往往适合网络学习的，但如果以分布P(t)进行采样，在求均值时均匀分布还好，但是非均匀分布会引入很大的偏差

举个例子（摘自博客 原文链接：https://blog.csdn.net/MR_kdcon/article/details/112134708

某校男生1000人，女生5000人，且男生均比女生高，我们为了估计该校学生的平均身高，就必须进行采样（不然一个个去测量，然后求平均太费时费力了）。但是采样的时候，其中4950个女生由于某种原因不能参加，所以只能在1000个男生，50个女生中采样。如果这时候通过采样求取平均的方式，首先男生有约95%的概率抽到，女生仅仅5%，故比如说最后抽100个求平均，那么其结果肯定是偏离真实值的，也就是bias会很大，其本质原因是非均匀采样导致的。

所以**这时候就要引入修正因子：重要性采样权重。**通过加权的方式来修正：

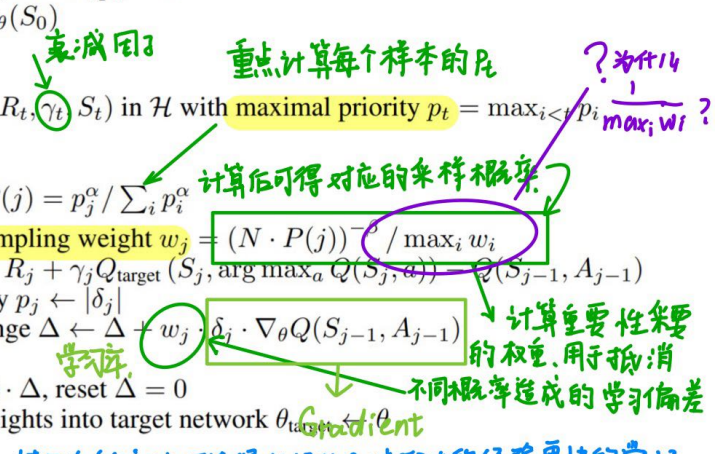
$$\text{权重为 } \omega_t = \frac{(N \times P(t))^{-\beta}}{\max_i \omega_i}$$

流程：

经验回放

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for



其中有一点疑问就是重要性采样的权重为什么要除以最大的权重?

Big $|\delta_t|$ ==> High probability ==> Small learning rate

更大的TD——>抽样的概率越大——>学习率小

从而抵消大概率抽样样本的偏差

目标网络:

经验回放用于解决样本的非独立分布问题，并且提高了训练样本的利用率，但是训练时的目标Q函数频繁的更新并未解决，所以这里引入目标网络这一概念

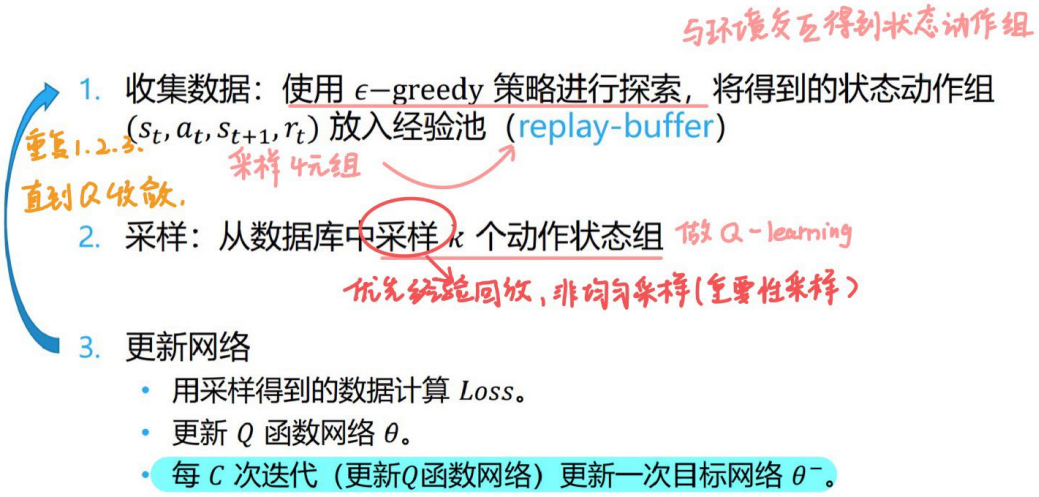
既然训练过程中 Q 网络的不断更新会导致目标不断发生改变，不如暂时先将 TD 目标中的 Q 网络固定住。为了实现这一思想，我们需要利用两套 Q 网络。

- 原来的网络$\mathbf{Q_\theta(s,a)}$，用于计算原来的损失函数，并且使用正常梯度下降法来进行更新
- 目标网络使用 $\boldsymbol{\theta^-}$ 旧的参数网络计算原来的损失函数中的目标值的项

如果两套网络的参数随时保持一致，则仍为原先不够稳定的算法。所以为了让更新目标更稳定，目标网络并不会每一步都更新。具体而言，目标网络使用训练网络的一套较旧的参数，训练网络$Q_\theta(s,a)$在训练中的每一步都会更新，而目标网络的参数每隔**C**步才会与训练网络同步一次，即 $\boldsymbol{\theta^- \leftarrow \theta}$。这样做使得目标网络相对于训练网络更加稳定。

算法流程:

算法流程



代码实践：

定义一层只有一层隐藏层的 Q 网络

```
class Qnet(torch.nn.Module):
    ''' 只有一层隐藏层的Q网络 '''
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(Qnet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))  # 隐藏层使用ReLU激活函数
        return self.fc2(x)
```

实现 DQN 算法

```
class DQN:
    ''' DQN算法 '''
    def __init__(self, state_dim, hidden_dim, action_dim, learning_rate,
gamma,
                 epsilon, target_update, device):
        self.action_dim = action_dim
        self.q_net = Qnet(state_dim, hidden_dim,
                          self.action_dim).to(device)  # Q网络
        # 目标网络
        self.target_q_net = Qnet(state_dim, hidden_dim,
                                 self.action_dim).to(device)
        # 使用Adam优化器
        self.optimizer = torch.optim.Adam(self.q_net.parameters(),
                                          lr=learning_rate)
        self.gamma = gamma  # 折扣因子
        self.epsilon = epsilon  # epsilon-贪婪策略
        self.target_update = target_update  # 目标网络更新频率
        self.count = 0  # 计数器,记录更新次数
```

```
        self.device = device

    def take_action(self, state):  # epsilon-贪婪策略采取动作
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.action_dim)
        else:
            state = torch.tensor([state], dtype=torch.float).to(self.device)
            action = self.q_net(state).argmax().item()
        return action

    def update(self, transition_dict):
        states = torch.tensor(transition_dict['states'],
                              dtype=torch.float).to(self.device)
        actions = torch.tensor(transition_dict['actions']).view(-1, 1).to(
            self.device)
        rewards = torch.tensor(transition_dict['rewards'],
                               dtype=torch.float).view(-1,
1).to(self.device)
        next_states = torch.tensor(transition_dict['next_states'],
                                   dtype=torch.float).to(self.device)
        dones = torch.tensor(transition_dict['dones'],
                             dtype=torch.float).view(-1, 1).to(self.device)

        q_values = self.q_net(states).gather(1, actions)  # Q值
        # 下个状态的最大Q值
        max_next_q_values = self.target_q_net(next_states).max(1)[0].view(
            -1, 1)
        q_targets = rewards + self.gamma * max_next_q_values * (1 - dones
                                                                ) # TD误差目
标
        dqn_loss = torch.mean(F.mse_loss(q_values, q_targets))  # 均方误差损失
函数
        self.optimizer.zero_grad()  # PyTorch中默认梯度会累积,这里需要显式将梯度置
为0
        dqn_loss.backward()  # 反向传播更新参数
        self.optimizer.step()

        if self.count % self.target_update == 0:
            self.target_q_net.load_state_dict(
                self.q_net.state_dict())  # 更新目标网络
        self.count += 1
```