DQN改进算法

DQN 算法敲开了深度强化学习的大门，但是作为先驱性的工作，其本身存在着一些问题以及一些可以改进的地方。于是，在 DQN 之后，学术界涌现出了非常多的改进算法。本章将介绍其中两个非常著名的算法：Double DQN 和 Dueling DQN，这两个算法的实现非常简单，只需要在 DQN 的基础上稍加修改，它们能在一定程度上改善 DQN 的效果。

Double DQN

在DQN中有一个臭名昭著的问题，这个问题还在未使用深度学习之前都被诟病，这就是**Q函数的过高估计**

因为在Q-learning中的目标值，在更新时，每次都是选择基于当前的Q值的最大值进行更新

$$y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$$

max 操作使得 Q 函数的值越来越大，甚至高于真实值

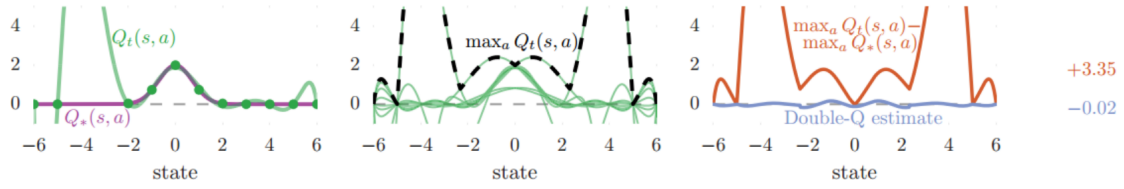
所以如果对于我们现在的Q值如果具有一定的偏差，那么它对应的下一个状态的Q值是在此基础上是偏大的，所以我们的目标值在估计时是不准确的，随着每一步的更新，这个误差累积会越来越大，对于动作空间较大的任务，DQN中的过高估计问题会非常严重，甚至导致DQN无法有效工作

所以为了解决该问题，首先分析一下**max**操作

这里**max**操作可以分为两部分：

- 选取s_{t+1}状态下的最优动作**a***，$\text{argmax}_a Q$来计算a*
- 然后根据a* 计算该动作价值对应的$Q(s_{t+1}, \mathbf{a^*})$

当这两部分采用同一套Q网络进行计算时，就会使得在不稳定的基础上更不稳定，假设一套网络的训练如下图所示



其中，x轴为状态，10个候选行动(绿色的点)；紫线是真实价值函数，绿点是训练数据点，绿线是拟合的价值函数

在-4~-2之间，拟合的数据线（绿线）与真实的数据线（紫线）有很大的偏差，所以在这种情况下去**$\text{argmax}_a Q$**，如在二张图的虚线所示（各个状态下价值最大的动作函数），Q值取得最大值，得到Q最大值的动作。这是max操作的第一部分

如果我们拟合的不好，就如最后一张图中的gap差距是非常非常大了，所以如果用这种Q函数去做目标值，去用于学习Q的参数是非常不靠谱的。在去计算Q值，max操作的第二部分

所以**Double DQN**被提出来的目的就是去解决Q-learning的过高估计，这种过高估计的原因也就是Q函数学的不是特别稳定，所以在某些位置某些时刻过多的偏差于真实的Q函数，使得在$\text{argmax}_a Q$时经常去选择偏的特别厉害，过高估计的Q值

所以针对这个问题提出了，我们有**两套Q函数**，一种Q函数的参数为**θ**，另一种Q函数的参数为**θ'**

所以将max操作的两部分对应两套网络：

- 选取s_{t+1}状态下的最优动作**a***，使用**$\text{argmax}_a Q_\theta(s_{t+1},a')$**来评估选取价值最大的动作
- 在计算时，使用另一套参数为**θ'** 网络更新Q值

$$\text{Double DQN} \quad y_t = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a'))$$

在传统的 DQN 算法中，本来就存在两套函数的神经网络——目标网络和训练网络，只不过Q值的计算只用到了其中的目标网络，那么我们恰好可以直接将训练网络作为 Double DQN 算法中的第一套神经网络来选取动作，将目标网络作为第二套神经网络计算值，这便是 Double DQN 的主要思想

代码实践：

```python
class DQN:
    ''' DQN算法,包括Double DQN '''
    def __init__(self,
                 state_dim,
                 hidden_dim,
                 action_dim,
                 learning_rate,
                 gamma,
                 epsilon,
                 target_update,
                 device,
                 dqn_type='VanillaDQN'):
        self.action_dim = action_dim
        self.q_net = Qnet(state_dim, hidden_dim, self.action_dim).to(device)
        self.target_q_net = Qnet(state_dim, hidden_dim,
                                 self.action_dim).to(device)
        self.optimizer = torch.optim.Adam(self.q_net.parameters(),
                                          lr=learning_rate)
        self.gamma = gamma
        self.epsilon = epsilon
        self.target_update = target_update
        self.count = 0
        self.dqn_type = dqn_type
        self.device = device

    def take_action(self, state):
        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 max_q_value(self, state):
        state = torch.tensor([state], dtype=torch.float).to(self.device)
```

```
        return self.q_net(state).max().item()

    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
        if self.dqn_type == 'DoubleDQN': # DQN与Double DQN的区别
            max_action = self.q_net(next_states).max(1)[1].view(-1, 1)
            max_next_q_values = self.target_q_net(next_states).gather(1,
max_action)
        else: # DQN的情况
            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
```

Dueling DQN

假设我们的动作价值函数服从某一个分布：

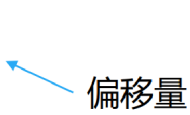
$$Q(s,a) \sim \mathcal{N}(\mu, \sigma)$$

那这个分布的均值就是状态价值函数，即：

$$V(s) = \mathbb{E}[Q(s,a)] = \mu$$

因为Q值是在某个确定状态下的动作价值，那么在该状态下所有动作价值的均值就是该状态的价值，所以我们可以转换的看作我们Q值是状态价值加上某个偏移量：

$$Q(s,a) = \mu + \varepsilon(s,a)$$



针对这个偏移量我们定义优势函数这一概念：

$$\varepsilon(s,a) = Q(s,a) - V(s)$$

所以如果这个偏移量是正值的话，说明在当前状态下采取该动作的价值是高于平均值的，所以可以考虑去采取该动作，而偏移量为负值的话，就建议不要去采取这个动作

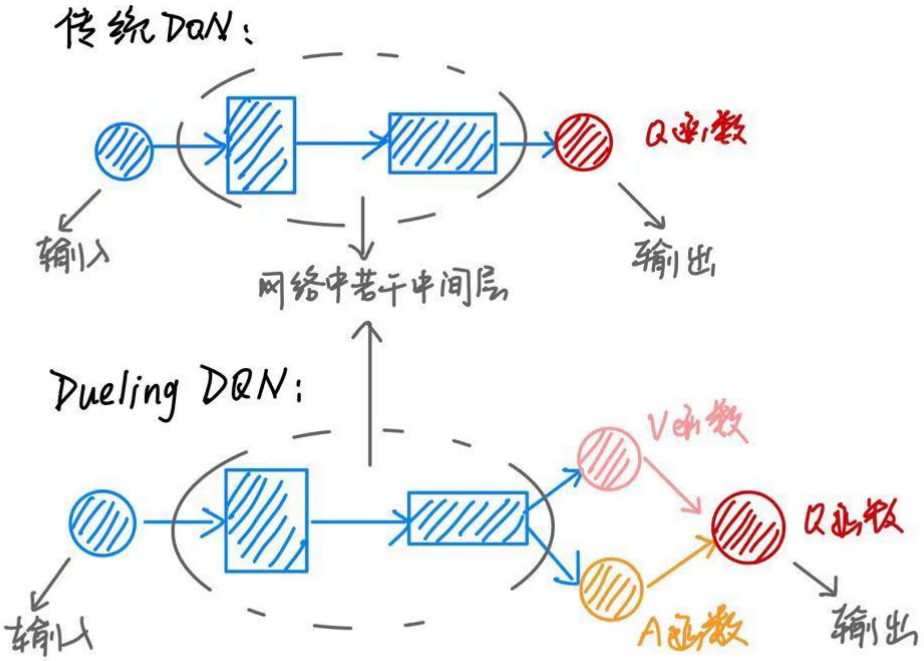
所以这个优势函数我们可以把它直接定义出来：

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

$$Q^{\pi}(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}[Q^{\pi}(s,a)]$$

所以在引入这个函数的概念之后，我们对网络的建模进行一定的改动，如下图所示

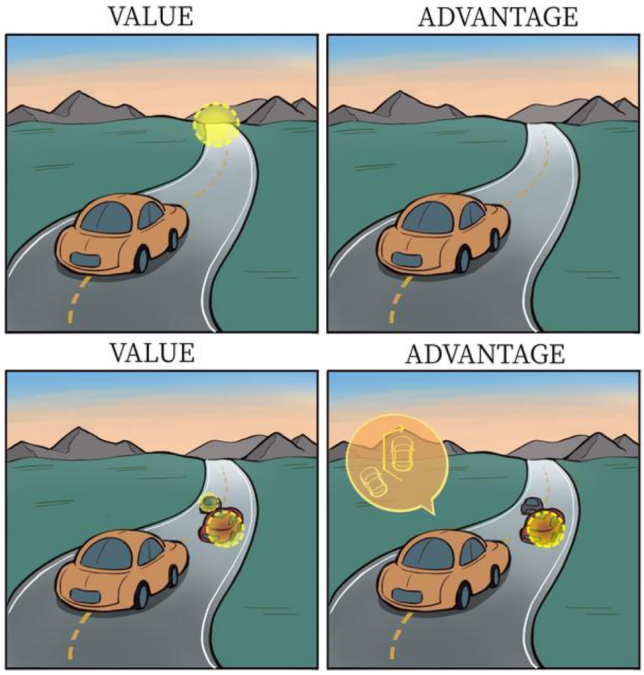


Dueling DQN与传统的DQN在前面的网络搭建都是一样的，重点在于在最后根据数据建模时将状态函数和优势函数分开建模，然后最后将两个建模好的函数合并作为Q函数：

$$Q_{\eta,\alpha,\beta}(s,a) = V_{\eta,\alpha}(s) + A_{\eta,\beta}(s,a)$$

将状态价值函数和优势函数分别建模的好处在于：某些情境下智能体只会关注状态的价值，而并不关心不同动作导致的差异，此时将二者分开建模能够使智能体更好地处理与动作关联较小的状态

在下图所示的驾驶车辆游戏中，智能体注意力集中的部位被显示为黄色高光部分，当智能体前面没有车时，车辆自身动作并没有太大差异，此时智能体更关注状态价值，而当智能体前面有车时（智能体需要超车），智能体开始关注不同动作优势值的差异。



所以可以得出Dueling DQN的一些优点：

- 可以处理与动作关联的较小状态
- 状态值函数的学习较为有效：一个状态值函数对应多个优势函数

代码实践：

```
class VAnet(torch.nn.Module):
    ''' 只有一层隐藏层的A网络和V网络 '''
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(VAnet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)  # 共享网络部分
        self.fc_A = torch.nn.Linear(hidden_dim, action_dim)
        self.fc_V = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        A = self.fc_A(F.relu(self.fc1(x)))
        V = self.fc_V(F.relu(self.fc1(x)))
        Q = V + A - A.mean(1).view(-1, 1)  # Q值由V值和A值计算得到
        return Q

```

Dueling DQN 与 DQN 相比的差异只是在网络结构上，大部分代码依然可以继续沿用。上述为定义的状态价值函数和优势函数的复合神经网络`VAnet`

```
class DQN:
    ''' DQN算法,包括Double DQN和Dueling DQN '''
    def __init__(self,
                 state_dim,
                 hidden_dim,
                 action_dim,
                 learning_rate,
                 gamma,
                 epsilon,
                 target_update,

```

```
                 device,
                 dqn_type='VanillaDQN'):
        self.action_dim = action_dim
        if dqn_type == 'DuelingDQN':  # Dueling DQN采取不一样的网络框架
            self.q_net = VAnet(state_dim, hidden_dim,
                               self.action_dim).to(device)
            self.target_q_net = VAnet(state_dim, hidden_dim,
                                      self.action_dim).to(device)
        else:
            self.q_net = Qnet(state_dim, hidden_dim,
                              self.action_dim).to(device)
            self.target_q_net = Qnet(state_dim, hidden_dim,
                                     self.action_dim).to(device)
        self.optimizer = torch.optim.Adam(self.q_net.parameters(),
                                          lr=learning_rate)
        self.gamma = gamma
        self.epsilon = epsilon
        self.target_update = target_update
        self.count = 0
        self.dqn_type = dqn_type
        self.device = device

    def take_action(self, state):
        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 max_q_value(self, state):
        state = torch.tensor([state], dtype=torch.float).to(self.device)
        return self.q_net(state).max().item()

    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)
        if self.dqn_type == 'DoubleDQN':
            max_action = self.q_net(next_states).max(1)[1].view(-1, 1)
            max_next_q_values = self.target_q_net(next_states).gather(
                1, max_action)
        else:
            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)
        dqn_loss = torch.mean(F.mse_loss(q_values, q_targets))
        self.optimizer.zero_grad()
        dqn_loss.backward()
        self.optimizer.step()

        if self.count % self.target_update == 0:
            self.target_q_net.load_state_dict(self.q_net.state_dict())
        self.count += 1
```

小结：

- DQN：一次输入多个行动Q值输出、目标网络、随机采样经验

$$y_t = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_{\theta'}(s_{t+1}, a'))$$

“Human-Level Control Through Deep Reinforcement Learning”, Mnih, Kavukcuoglu, Silver et al. Nature 2015.

- Double DQN：解耦合行动选择和价值估计、解决DQN过高估计问题

$$y_t = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_{\theta}(s_{t+1}, a'))$$

“Double Reinforcement Learning with Double Q-Learning”, van Hasselt et al. AAAI 2016.

- Dueling DQN：精细捕捉价值和行动的细微关联、多种advantage函数建模

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)$$