中山大学计算机学院

人工智能

本科生实验报告

（2025 学年春季学期）

课程名称：Artificial Intelligence

教学班级		专业（方向）	
学号		姓名	

一、 实验题目

实现 DQN 算法

在 CartPole-v0 环境中实现 DQN 算法

最终算法性能的评判标准：以算法收敛的 reward 大小、收敛所需的样本数量给分。reward 越高（至少是 180，最大是 200）、收敛所需样本数量越少，分数越高

二、 实验内容

1. 算法原理

1.1 $DQN(Deep\ Q-network)$

DQN 可以看作$Q-learning$算法与深度神经网络的结合.

传统的$Q-learning$算法是通过下面的 TD 方式来更新并维护Q表，通过查表的方式来选择当前状态下能够获得最大收益的动作.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[R_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right] \quad (1)$$

但是$Q-learning$算法有一定的局限性，他必须要求状态和动作必须是离散的，对于连续的状态和动作，我们是无法建立表格的.

而DQN在$Q-learning$的基础上解决这个问题，如果状态和动作是连续的，那么$Q(s,a)$也是连续的，结合神经网络强大的表达能力，我们可以使用一个神经网络去拟合函数$Q(s,a)$.这就是DQN算法的思想.拟合$Q(s,a)$函数的神经网络称为Q网络.

观察（1）式子可以发现式子的第二项越小，那么说明当前状态提升的值就越小，就越说明当前策略就越接近最优策略.

所以我们这样定义Q网络的损失函数：

$$Loss(w) = \frac{1}{2N}\sum_{i=1}^{N}\left[Q_w(s_i,a_i) - (r_i + \gamma \max_a Q_w(s_i^{'},a_i^{'}))\right]^2 \quad (2)$$

这样我们DQN所要优化的目标就是：

$$w^* = \arg\min_w Loss(w) \quad (3)$$

其中w为Q网络的网络参数，w^*为Q网络最优网络参数

为了实现神经网络拟合$Q(s,a)$函数，DQN还引入了经验回放和目标网络两个机制.

1.2 目标网络

观察（2）式子可以发现，TD 误差目标本身也就是神经网络中的标签，已经包含神经网络的输出，因此在更新网络参数的同时目标也在不断地改变，这非常容易造成神经网络训练的不稳定性.

所以我们引入了目标网络机制：我们用$Q_{w^-}(s,a)$ 来表示目标网络，这个网络不进行梯度更新，每隔一段时间就将当前网络中的参数复制到目标网络进行更新.这样的做法能够使得网络的训练更加的稳定.

在引入目标网络之后，我们的损失函数就变为：

$$\mathcal{L}oss(w)=\frac{1}{2N}\sum_{i=1}^{N}\left[Q_w(s_i,a_i)-(r_i+\gamma\max_a Q_{w^-}(s_i^{'},a_i^{'}))\right]^2 \quad (4)$$

其中，Q_{w^-}表示目标网络，Q_w表示Q网络

1.3 经验回放

由于我们Q网络得到的样本前后是由关系的，不符合监督学习要求的数据独立同分布，为了是我们的网络和算法更加的鲁棒，DQN引入了经验回放机制：就是将每次探索所得到的数据都放入到一个缓冲区，每次训练时都从缓冲区进行采样来训练和更新我们的Q网络.

这样的优点有：打破了训练数据的前后关联性，同时统一数据可以重复利用，数据利用率，有利于神经网络的学习.

1.4 $\varepsilon-Greedy$策略

在动作选择策略中如果一直采用贪心的策略，可能会导致一些状态永远的不会出现这样会导致可能会错过更优的动作和状态.

所以这里对贪心策略进行了优化，引入概率参数ε，表示有ε的概率随机选择其他非最优动作，有 1-ε的额外概率选择使得$Q(s,a)$最大的动作.

数学公式如下：

$$\pi(a|s)=\begin{cases}\dfrac{\varepsilon}{|\mathcal{A}|}, & a\neq \underset{a\in\mathcal{A}}{argmax}\,Q(s,a)\\ 1-\varepsilon+\dfrac{\varepsilon}{|\mathcal{A}|}, & a=\underset{a\in\mathcal{A}}{argmax}\,Q(s,a)\end{cases} \quad (5)$$

2. 伪代码

Algorithm: *DQN*

Inputs: ReplayBuffer: $\mathcal{R}$, batch_size: N, target_net_update_interval: d
Return: policy: π
initialize: $\mathcal{R}$, Qnet: $Q_\phi(s,a)$, target_net: $Q_{\phi^-}(s,a)$
for ep=1,2,...,E do
    initialize env and get initial state s_1
    for t=1,2,...,T do
        a_t $Q_\phi(s,a)$ by $\varepsilon-Greedy\ policy$
        get s_{t+1}, r_t by execute a_t
        $\mathcal{R} \leftarrow (s_t, a_t, s_{t+1}, r_t)$
        sample N data in $\mathcal{R}$
        $\mathcal{L}oss(\phi) = \frac{1}{2N}\sum_{i=1}^{N}\left[Q_\phi(s_i,a_i) - (r_i + \gamma \max_{a'} Q_{\phi^-}(s_i', a_i'))\right]^2$
        update $\phi \leftarrow \phi - \alpha \cdot \frac{d\mathcal{L}oss}{d\phi}$
        if t % d == 0 then
            update $Q_{\phi^-}(s,a) \leftarrow Q_\phi(s,a)$
        end if
    end for
end for

3. 关键代码展示

3.1 Qnetwork

Qnet，在本次实验中设置了输入为状态向量，输出为动作向量，一个 128 个神经元的隐藏层，并以 ReLU 函数激活的神经网络作为本次实验的 Qnet

```
class QNetwork(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):

        super(QNetwork, self).__init__()

        # 隐藏层

        self.fc1=nn.Sequential(

            torch.nn.Linear(input_size,hidden_size),

            nn.ReLU()

        )

        # 输出层

        self.fc2=nn.Sequential(
```

```
            torch.nn.Linear(hidden_size,output_size)
        )
    def forward(self, inputs):
        inputs=self.fc1(inputs)
        inputs=self.fc2(inputs)
        return inputs
```

3.2 经验回放池 ReplayBuffer

经验回放池，采用 collections 中的双端队列 deque 来维护.因为当 buffer 满时会自动换出最早进入的元素.

```
# 经验回放池类
class ReplayBuffer:
    def __init__(self, buffer_size):
        self.buffer = collections.deque(maxlen=buffer_size)  # 双端队列,先进先出
    def __len__(self):
        return len(self.buffer)
    def push(self, *transition):
        self.buffer.append(transition)  # deque 的好处当 buffer 满时会自动换出最早进入的元素
    # 从 buffer 中采样数据,数量为 batch_size
    def sample(self, batch_size):
        # 随机抽取 batch_size 个数据
        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
    # 清空 buffer
    def clean(self):
        self.buffer.clear()
```

3.3 AgentDQN

3.3.1 AgentDQN 的初始化

初始化智能体，包括 Qnet 的网络、目标网络、优化器、学习率调度器、随机种子、损失函数、回合数、折扣因子、ε 、初始状态

```
    def __init__(self, env, args):
        """
        Initialize every things you need here.
        For example: building your model
        """
        super(AgentDQN, self).__init__(env)
        self.env=env
        # 随机种子
        random.seed(0)
        np.random.seed(0)
        env.seed(0)
        torch.manual_seed(0)
        # QNet 相关 args
        self.state_dim=env.observation_space.shape[0]   # 输入维度
        self.action_dim=env.action_space.n  # 输出层维度
        self.hidden_size=args.hidden_size   # 隐藏层维度
        self.lr=args.lr # 学习率
        self.device=torch.device("cuda" if args.use_cuda else "cpu")
        # 训练网络
        self.train_Qnet=QNetwork(self.state_dim,self.hidden_size,self.action_dim).to(self.device)
        # 目标网络
        self.target_Qnet=QNetwork(self.state_dim,self.hidden_size,self.action_dim).to(self.device)
        # 优化器
        self.optimizer=torch.optim.Adam(self.train_Qnet.parameters(),lr=self.lr)
        # 学习率调度器
        self.scheduler=torch.optim.lr_scheduler.ReduceLROnPlateau(self.optimizer, mode='m
ax', factor=0.999, patience=5, min_lr=1e-5)
```

```
        # 损失函数
        self.criterion=nn.MSELoss()
        # 回合数
        self.episodes=args.episodes
        # 经验回放池
        self.replay_buffer=ReplayBuffer(args.buffer_size)
        self.batch_size=args.batch_size # 批大小
        self.min_size=args.min_size
        # 动作选择
        self.epsilon=args.epsilon   # epsilon-贪心策略
        self.gamma=args.gamma   # 折扣因子
        self.cnt=0  # 计数器
        self.target_update_cnt=args.target_update_cnt   # 目标网络更新间隔
    def init_game_setting(self):
        """
        Testing function will call this function at the begining of new game
        Put anything you want to initialize if necessary
        """
        self.env.seed(11037)
        state = self.env.reset()  # 使用默认种子或从参数中获取
        return state
```

3.3.2 Agent 的训练

按照伪代码中的算法流程来编写智能体训练的算法流程.这里创建了一个一个 returns 数组来记录每次训练 rewards，用于结果可视化

```
    def train(self):
        """
        Implement your training algorithm here
        """
        returns=[]
        for i in range(10):
```

```
        for t in range(int(self.episodes/10)):
            # 获得初始状态
            state=self.init_game_setting()
            episode_reward=0
            done=False
            while not done:
                # 获取下一个 action
                action=self.make_action(state)
                # 执行 action 转移到下一个状态并返回 reward
                next_state,reward,done,_=self.env.step(action)
                episode_reward+=reward
                # 放入 ReplayBuffer
                self.replay_buffer.push(state, action, reward, next_state, done)
                state=next_state
                # 如果经验回放池数量足够，那么抽样来更新网络参数
                if len(self.replay_buffer) > self.min_size:
                    # 采样
                    states,actions,rewards,next_states,dones=self.replay_buffer.sample(self.batch_size)
                    # 更新网络参数，与环境进行交互
                    self.run(states,actions,rewards,next_states,dones)
            returns.append(episode_reward)
            # 调整学习率
            self.scheduler.step(episode_reward)
            print(f'Episode {i+1},Time step {t+1}: Return {episode_reward}')
    return returns
```

3.3.3 $\varepsilon-Greedy$动作选择策略

采用$\varepsilon-Greedy$的动作选择策略，以 ε 的概率来随机选择其他非最优动作，1-ε 的概率来选择使得Q值最大的动作．ε 的概率越大表示智能体越倾向于随机探索，ε 越小智能体越趋向于奖励最大的动作．

```
# ε-贪婪策略采取动作
```

```
    def make_action(self, observation, test=True):
        """
        Return predicted action of your agent
        Input:observation
        Return:action
        """
        # 以ε的概率随机生成动作，表示随机探索
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.action_dim)
        # 以 1-ε的概率执行令 Qnet 最大的动作 action
        else:
            state = torch.tensor([observation], dtype=torch.float).to(self.device)
            action = self.train_Qnet(state).argmax().item()
        return action
```

3.3.4 Qnet 网络的更新

智能体与环境的交互，来通过更新神经网络的参数来改变我们选择动作的策略，从而实现智能体与环境的交互.从经验回放池中采样一定的数据，然后计算其 Loss 值，然后反向传播采用梯度下降法更新我们的 Q 网络参数.如果达到了一定的时间间隔，就用我们当前的网络参数取更新目标网络参数

```
    def run(self,*args):
        """
        Implement the interaction between agent and environment here
        """
        # 数据处理将数据转换成合适的类型和维度
        states,actions,rewards,next_states,dones=args
        states=torch.tensor(states,dtype=torch.float32).to(self.device)
        actions=torch.tensor(actions,dtype=torch.int64).view(-1, 1).to(self.device)
        rewards=torch.tensor(rewards,dtype=torch.float32).view(-1, 1).to(self.device)
        next_states=torch.tensor(next_states,dtype=torch.float32).to(self.device)
        dones=torch.tensor(dones,dtype=torch.float32).view(-1, 1).to(self.device)
```

```
        # Q 值
        q_values=self.train_Qnet(states).gather(1,actions)
        # 下个状态最大 Q 值
        max_next_q_values=self.target_Qnet(next_states).max(1)[0].view(-1,1)
        # TD 误差目标
        q_targets=rewards+self.gamma*max_next_q_values*(1-dones)
        # 损失值
        loss=self.criterion(q_values,q_targets)
        # 清空梯度
        self.optimizer.zero_grad()
        # 反向传播
        loss.backward()
        self.optimizer.step()
        # 当间隔 update_cnt 时更新目标网络
        if self.cnt % self.target_update_cnt == 0:
            self.target_Qnet.load_state_dict(self.train_Qnet.state_dict())
        self.cnt+=1
```

3.4 本次实验的参数设置

```
parser.add_argument('--env_name',default="CartPole-v0", help='environment name')
parser.add_argument("--seed", default=11037,type=int)
parser.add_argument("--hidden_size",default=128, type=int)
parser.add_argument("--lr", default=0.002,type=float)
parser.add_argument("--gamma", default=0.98,type=float)
# 实现 DQN 所需要的额外参数
parser.add_argument("--batch_size", default=64,type=int)
parser.add_argument("--buffer_size",default=10000, type=int)
parser.add_argument("--epsilon", default=0.01,type=float)
parser.add_argument("--episodes", default=300,type=int)
parser.add_argument("--min_size", default=500,type=int)
parser.add_argument("--target_update_cnt",default=10, type=int)
```

4. 创新点&优化

在课堂上我们除了学习了最基本的DQN算法，同时也学习了DQN算法的优化版本，比如$DoubleDQN$、$Dueling\ DQN$，**本次实验在DQN的基础上也实现和对比了以上几种种算法.**

4.1 $Double\ DQN$

传统的DQN算法优化的目标为：

$$r+\gamma Q_{w^-}(s^{'},\underset{a^{'}}{argmax}\,Q_{w^-}(s^{'},a^{'}))$$

而$Double\ \ DQN$算法优化的目标变为了：

$$r+\gamma Q_{w^-}(s^{'},\underset{a^{'}}{argmax}\,Q_{w}(s^{'},a^{'}))$$

也就是使用训练网络的 Q 值来选择下一个动作，而不是传统DQN使用目标网络来选择

4.2 $Dueling\ DQN$

$DueliDuelingg\ DQN$在DQN的基础上引入了优势函数A，它由状态动作价值Q减去状态价值V，同一个状态下，所有动作的优势值之和为 0，因为所有动作的动作价值的期望就是这个状态的状态价值.

所以$Dueling\ DQN$将Q网络的建模换成了：

$$Q(s,a)=V(s,a)+A(s,a)$$

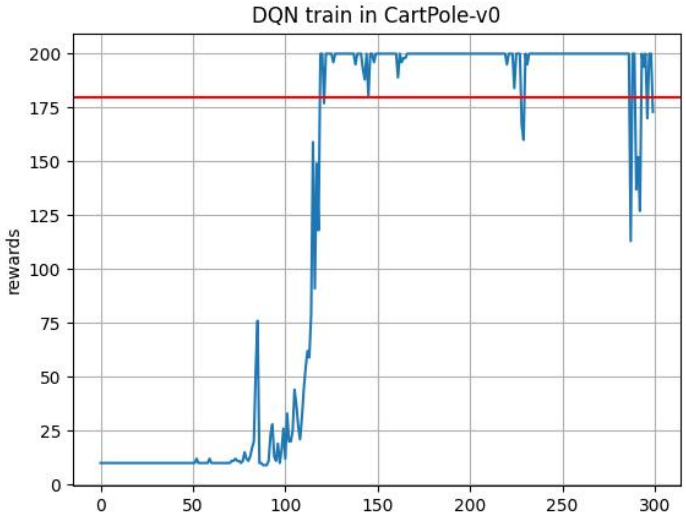
更加详细的介绍可以查阅仓库：Hands-on-RL/第 8 章-DQN 改进算法.ipynb at main • boyu-ai/Hands-on-RL

三、 实验结果及分析

1. 实验结果展示

1.1 DQN 的 reward 曲线

可以看到 rewards 曲线在 125 轮左右就收敛到了 200 分（满分为 200 分），这样的结果是比较优秀的.红色的线表示及格线 180 分



2. 评测指标展示及分析

2.1 不同参数对*DQN*模型性能的影响

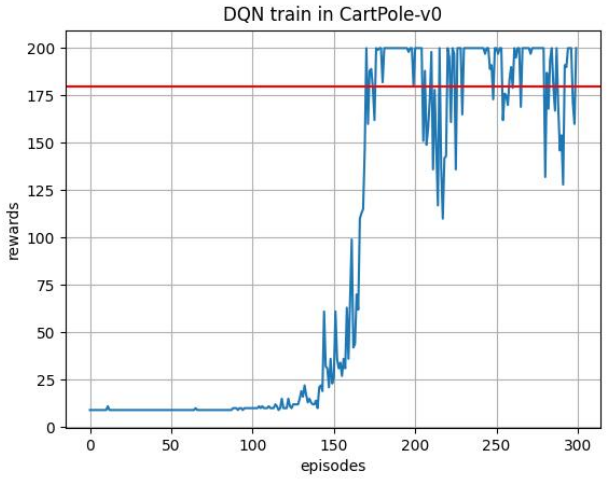
Q 网络结构	γ	ε	学习率	收敛 ep 数	Rewards
单隐藏层 128	0.98	0.01	0.002	120	200
单隐藏层 128	0.7	0.01	0.002	150	190(抖动大)
单隐藏层 128	0.98	0.1	0.002	340	200
单隐藏层 128	0.98	0.01	0.01	70	115(过拟合)

通过表格可以看出，**衰减因子 γ 大小决定着智能体的选择是更加倾向于及时回报还是长期回报**，上表的情况就是 γ 过小，导致智能体缺乏长远效益，导致 rewards 抖动大，而且可能当前局部的最优解中.

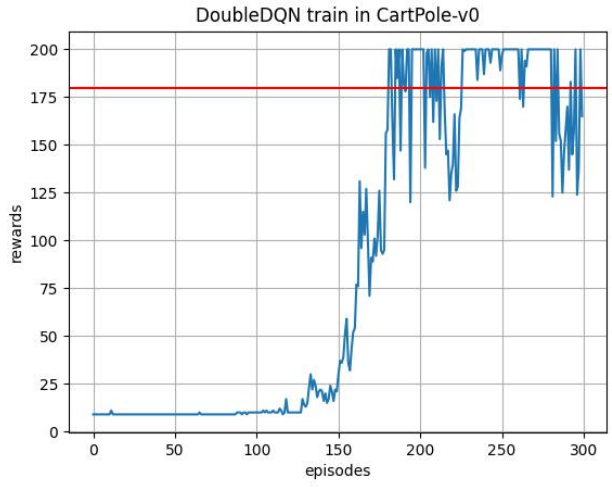
ε 决定智能体选择动作时是更加倾向于随机探索还是选择最大收益的动作选择，可以看到当 ε 变大时，智能体更加倾向于随机探索，rewards 收敛的回合数明显变大了.

学习率 lr 和之前的神经网络一样影响函数的拟合效果，学习率如果过小会导致收敛速度慢，过大会导致过拟合的现象，上表就是学习率过大导致了过拟合的现象，在后期 rewards 明显变小.

2.2 *DQN*、*Double DQN*、*Dueling DQN* 算法的比较

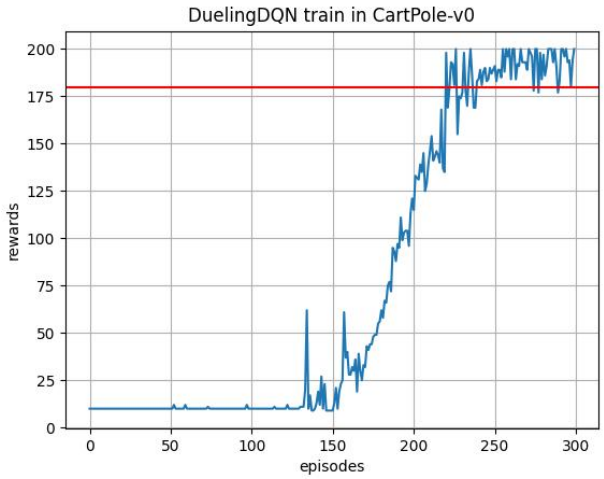


这是使用 *DQN* 算法训练后的 Rewards 曲线



这是使用*Double DQN*算法在相同参数情况下的结果.

可以看到*Double DQN*和*DQN*的曲线大致一样，但是*Double DQN*产生的抖动的极差更加的小，这是因为改进的算法*Double DQN*能够极大的缓解*Q*值过高估计的问题.这一点可以对比训练过程中的*Q*值的曲线有更加明显的结果.



这是使用$Dueling\ DQN$算法在相同参数情况下的结果.

可以看到$Dueling\ DQN$算法在收敛时产生的抖动更小，相比于传统的DQN,$Dueling\ DQN$ 在多个动作选择下的学习更加稳定.随着动作空间的增大这种稳定优势会更加的明显