# 中山大学计算机学院

# 人工智能

# 本科生实验报告

**（2023 学年春季学期）**

## 一、 实验题目

**DQN**

## 二、 实验内容

### 1. 算法原理

深度 Q 网络（DQN）是一种结合了深度学习和强化学习的技术，用于解决像 CartPole 这样的控制问题。核心是经验回放和目标网络。经验回放通过存储智能体与环境交互过程中的状态转移元组（包括状态、动作、奖励、下一个状态和完成标志），并从中随机采样来训练网络，这有助于算法利用数据更加高效，同时减少训练过程中的方差。目标网络是评估网络的一个延迟副本，它定期更新以保持 Q 值目标的稳定性。在训练过程中，智能体使用 ε-greedy 策略进行探索，即大多数时候选择当前估计最佳的行动，但以小概率 ε 随机选择动作以探索未知策略。随着训练的进行， ε 会逐渐减小，使得智能体从探索转向利用。通过这种方式，DQN 算法能够学习到在 CartPole 游戏中保持杆子平衡的有效策略。

### 2. 伪代码

```
    def learn(self):
procedure learn(self)
    if length(self.buffer) < args.batch_size
        return
    end if

    if self.learn_step % args.update_target = 0
```

```
            self.target_net.load_state_dict(self.eval_net.state_dict())
        end if
        self.learn_step := self.learn_step + 1

        (obs, actions, rewards, next_obs, dones) := self.buffer.sample(args.batch_size)
        actions := to_LongTensor(actions).unsqueeze(1)
        dones := to_FloatTensor(dones)
        rewards := to_FloatTensor(rewards)

        q_eval := self.eval_net(obs).gather(1, actions)
        q_next := self.target_net(next_obs).max(1)[0].detach()
        q_target := rewards + args.gamma * (1 - dones) * q_next

        loss := self.loss_fn(q_eval, q_target.unsqueeze(1))
        self.optim.zero_grad()
        loss.backward()
        self.optim.step()
    end procedure
```

### 3. 关键代码展示（带注释）

**神经网络定义**

```python
class QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNet, self).__init__()
        # 两个线性层
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.Tensor(np.array(x))
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

**经验库**

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0
        self.transition = namedtuple('Transition', ('state', 'action',
'reward', 'next_state', 'done'))

    def __len__(self):
        return len(self.buffer)
```

```
    def push(self, *transition):
        # append(None)是为了占位，下面可以直接按下标进行访问
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = self.transition(*transition)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        # 在 buffer 中随机取样，batch_size
        transitions = random.sample(self.buffer, batch_size)
        batch = self.transition(*zip(*transitions))
        return batch.state, batch.action, batch.reward,
batch.next_state, batch.done

    def clean(self):
        self.buffer = []
        self.position = 0
```

**智能体 DQN（Agent）**

```
class DQN:
    def __init__(self, env, input_size, hidden_size, output_size):
        self.env = env
        self.eval_net = QNet(input_size, hidden_size, output_size)
        self.target_net = QNet(input_size, hidden_size, output_size)
        self.optim = optim.Adam(self.eval_net.parameters(), lr=args.lr)
        self.eps = args.eps
        self.buffer = ReplayBuffer(args.capacity)
        self.loss_fn = nn.MSELoss()
        self.learn_step = 0

    def choose_action(self, obs):
        if np.random.rand() < self.eps:
            return self.env.action_space.sample()
        else:
            with torch.no_grad():
                return torch.argmax(self.eval_net(obs)).item()

    def store_transition(self, *transition):
        self.buffer.push(*transition)

    def learn(self):
        if len(self.buffer) < args.batch_size:
            return
```

```
        # 滞后更新 target_net
        if self.learn_step % args.update_target == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step += 1

        obs, actions, rewards, next_obs, dones =
self.buffer.sample(args.batch_size)
        actions = torch.LongTensor(actions).unsqueeze(1)
        dones = torch.FloatTensor(dones)
        rewards = torch.FloatTensor(rewards)
        # 智能体决定采取的动作对应的 Q 值
        q_eval = self.eval_net(obs).gather(1, actions)
        q_next = self.target_net(next_obs).max(1)[0].detach()
        # 根据是否完成，实现 q_target 的赋值
        q_target = rewards + args.gamma * (1 - dones) * q_next

        loss = self.loss_fn(q_eval, q_target.unsqueeze(1))
        self.optim.zero_grad()
        loss.backward()
        self.optim.step()
```

**4. 创新点&优化（如果有）**

1、在 class Qnet 中，发出警告 Creating a tensor from a list of numpy.ndarrays is extremely slow.这里将 x = torch.Tensor(x)改成 x = torch.Tensor(np.array(x))，加快速度

2、根据论文中的伪代码，采取双网络，定义 eval_net 和 target_net，两个网络有相同的结构，并且每隔一段时间将 eval_net 中的参数复制到 target_net，通过 target_net 网络参数更新的滞后性，有助于避免过拟合问题

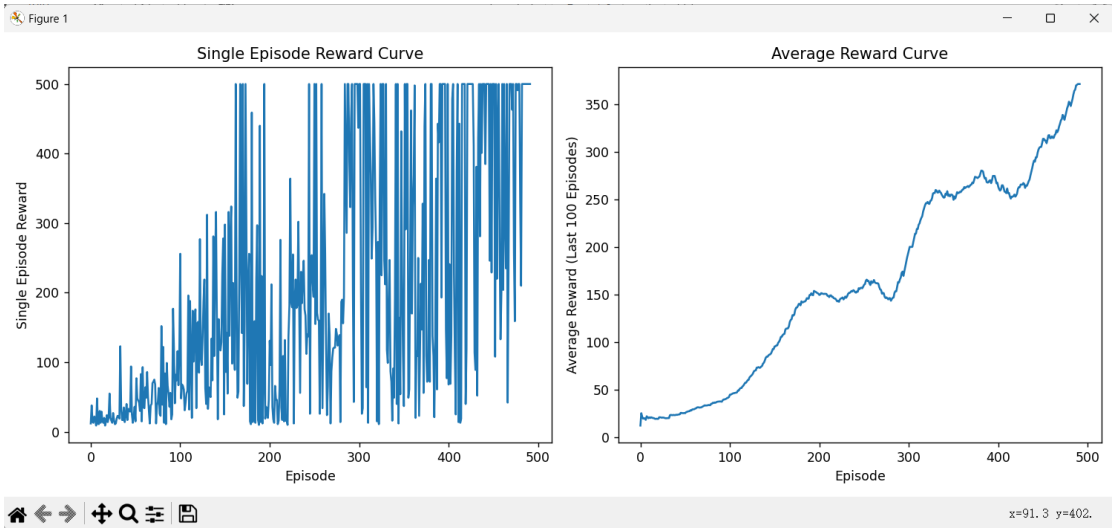
3、建立 ReplayBuffer 类，来实现经验机制，并且不断将新知识替换旧知识，稳定提高模型的处理能力

4、动态的探索率衰减，通过 np 库中的线性插值函数 np.interp 获取探索率 epsilon，并利用随机数比较的方式决定是随即探索还是利用网络生成。在训练初期多进行探索，在训练后期减少探索率。

# 三、 实验结果及分析

**|----------如有优化，请重复 1，2，分析优化后的算法结果----------|**

第一次运行结果，这里我设置学习率为 0.01，Qnet 有 3 个线性层，效果不佳

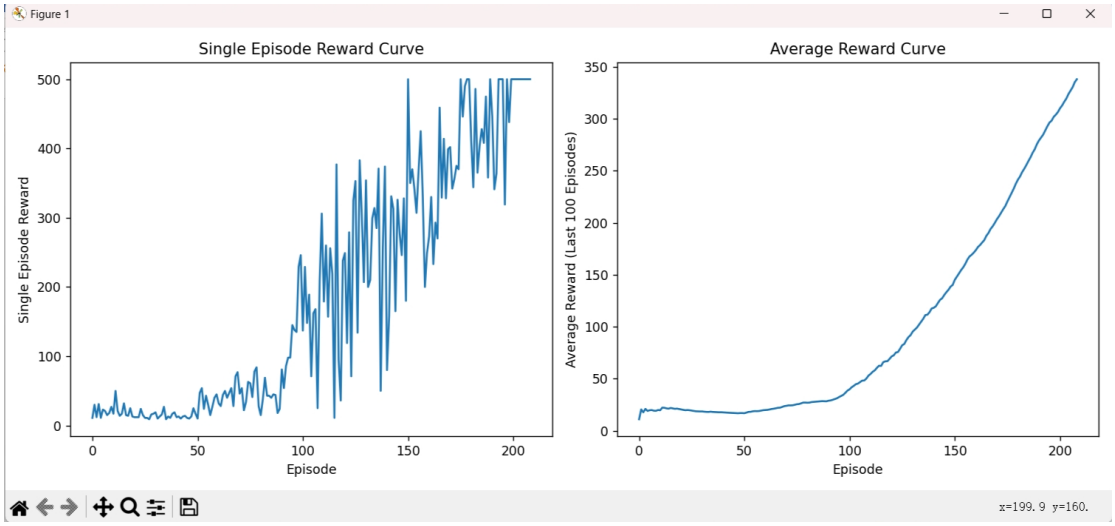



优化后的结果：设置学习率为 0.001，线性层减少一个，2 个线性层，探索率采取 np 库的线性插值方法，获取更加平滑下降的 epsilon 值

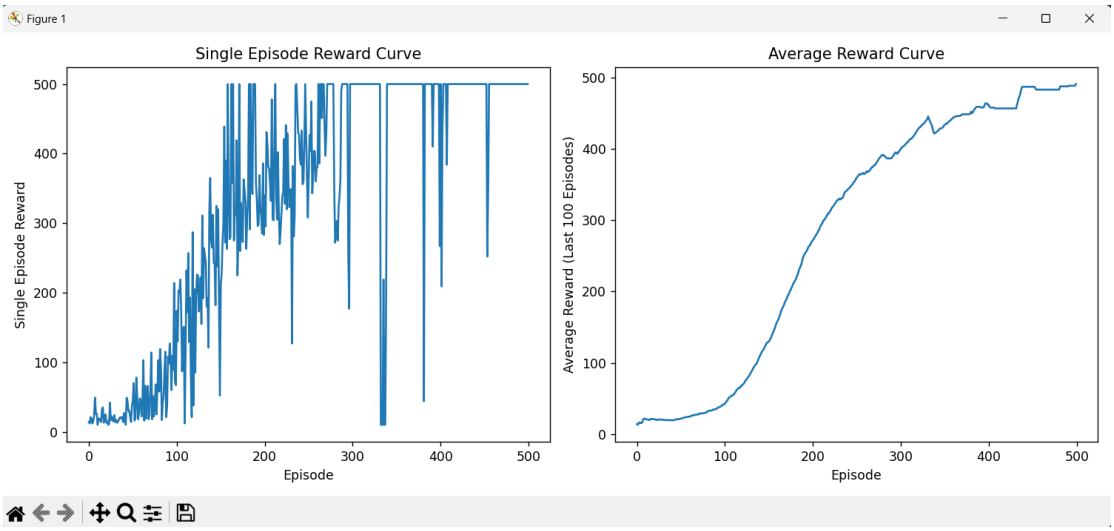
参考代码：[深度强化学习_DQN_纯白板逐行代码 Python 实现_哔哩哔哩_bilibili]

```
Episode 130, Average Reward: 95.3, Reward: 354.0
Episode 140, Average Reward: 118.23, Reward: 80.0
Episode 150, Average Reward: 145.11, Reward: 500.0
Episode 160, Average Reward: 173.67, Reward: 273.0
Episode 170, Average Reward: 203.39, Reward: 402.0
Episode 180, Average Reward: 241.49, Reward: 411.0
Episode 190, Average Reward: 279.18, Reward: 446.0
Episode 200, Average Reward: 310.14, Reward: 500.0
Solved in 208 episodes!
```

可以看到 208 局游戏，已经可以完成任务：连续 10 局 Reward 值为 500，208 局游戏的单局游戏 reward 曲线和最近 100 局游戏的平均 reward 曲线




接下来，运行 500 局游戏，观察奖励曲线




```
Solved in 473 episodes!
Solved in 474 episodes!
Solved in 475 episodes!
Solved in 476 episodes!
Solved in 477 episodes!
Solved in 478 episodes!
Solved in 479 episodes!
Episode 480, Average Reward: 483.45, Reward: 500.0
Solved in 480 episodes!
Solved in 481 episodes!
Solved in 482 episodes!
Solved in 483 episodes!
Solved in 484 episodes!
Solved in 485 episodes!
Solved in 486 episodes!
Solved in 487 episodes!
Solved in 488 episodes!
Solved in 489 episodes!
Episode 490, Average Reward: 488.01, Reward: 500.0
Solved in 490 episodes!
Solved in 491 episodes!
Solved in 492 episodes!
Solved in 493 episodes!
Solved in 494 episodes!
Solved in 495 episodes!
Solved in 496 episodes!
Solved in 497 episodes!
Solved in 498 episodes!
Solved in 499 episodes!
```

重复运行代码，**效果最佳的一次**，运行结果如下：
可以看到，在第 280 局游戏，最近百局得分便超过所要求的 475 分，收敛速度很快

```
Episode 270, Average Reward: 471.94, Reward: 253.0
Episode 280, Average Reward: 477.37, Reward: 500.0
Solved in 280 episodes!
Solved in 281 episodes!
Solved in 282 episodes!
Solved in 283 episodes!
Solved in 284 episodes!
Solved in 285 episodes!
```