中山大学计算机学院

人工智能

本科生实验报告

课程名称：Artificial Intelligence

学号	22336259	姓名	谢宇桐

一、 实验题目

实现 DQN 算法

1. 算法原理

本次实验要求我们在`CartPole-v0`环境中实现 DQN 算法

DQN

DQN（Deep Q Network）是深度神经网络和 Q-Learning 算法相结合的一种基于价值的深度强化学习算法。

Q-Learning 算法构建了一个状态-动作值的 Q 表，其维度为 (s,a)，其中 s 是状态的数量，a 是动作的数量，根本上是 Q 表将状态和动作映射到 Q 值。此算法适用于状态数量能够计算的场景。但是在实际场景中，状态的数量可能很大，这使得构建 Q 表难以解决。为破除这一限制，我们使用 Q 函数来代替 Q 表的作用，后者将状态和动作映射到 Q 值的结果相同。

由于神经网络擅长对复杂函数进行建模，因此我们用其当作函数近似器来估计此 Q 函数，这就是 Deep Q Networks。此网络将状态映射到可从该状态执行的所有动作的 Q 值。即只要输入一个状态，网络就会输出当前可执行的所有动作分别对应的 Q 值。如下图所示，它学习网络的权重，以此输出最佳 Q 值。

DQN 体系结构主要包含：Q 网络、目标网络，以及经验回放组件：

Q 网络是一个深度神经网络经过训练以生成最佳状态-动作值的 agent。它将环境状态作为输入，并输出每个可能动作的 Q 值（即期望回报）。这些 Q 值表示在给定状态下采取每个动作的预期效用。

目标网络是 Q 网络的一个副本，其参数在训练过程中定期更新。这有助于稳定训练过程，因为它减少了目标值的波动。

经验回放单元的作用是与环境交互，生成数据以训练 Q 网络。目标网络与 Q 网络在初始时是完全相同的。

关键代码展示（可选）

```python
class QNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        # 第一层全连接层，从输入状态到隐藏层
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        # 第二层全连接层，从隐藏层到另一个隐藏层
        self.fc3 = nn.Linear(hidden_size, output_size)
        # 第三层全连接层，从隐藏层到输出层，输出层的大小等于动作空间的大小

    def forward(self, inputs):
        # 使用ReLU激活函数对输入进行非线性变换
        out = torch.relu(self.fc1(inputs))
        out = torch.relu(self.fc2(out))
        # 输出层不使用激活函数，因为Q值是连续值
        out = self.fc3(out)
        return out
```

```python
class ReplayBuffer:
    # 经验回放缓冲区，用于存储过去的转换（状态，动作，奖励，下一个状态，完成标志）
    def __init__(self, buffer_size):
        self.buffer_size = buffer_size
        self.buffer = []
        self.position = 0
        self.priorities = np.zeros( shape: (buffer_size,), dtype=np.float32)
        # 用于优先级经验回放的优先级数组
        self.max_priority = 1.0
        return

    def __len__(self):
        # 返回缓冲区中的经验数量
        return len(self.buffer)
```

```
    1 个用法
    def push(self, *transition):
        # 将新的经验转换添加到缓冲区
        if len(self.buffer) < self.buffer_size:
            self.buffer.append(None)
        self.buffer[self.position] = transition
        # 初始优先级设为最大
        self.priorities[self.position] = self.max_priority
        # 循环位置索引
        self.position = (self.position + 1) % self.buffer_size
        return

    2 个用法 (1 个动态)
    def sample(self, batch_size):
        # 根据优先级概率采样经验
        priorities = self.priorities[:len(self.buffer)]
        probs = priorities ** 0.6
        probs /= probs.sum()
        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        batch = [self.buffer[idx] for idx in indices]
        return batch, indices
```

```
def clean(self):
    # 清空缓冲区
    self.buffer = []
    self.position = 0
    return

1 个用法
def update_priorities(self, indices, priorities):
    # 更新给定索引的优先级
    self.priorities[indices] = priorities
    # 更新最大优先级
    self.max_priority = max(self.max_priority, np.max(priorities))
    return
```

```python
class AgentDQN(Agent):
    def __init__(self, env, args):
        super(AgentDQN, self).__init__(env)
        # 初始化环境和参数
        self.env = env
        self.args = args
        # 选择设备，优先使用GPU
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        # 初始化Q网络
        self.q_network = QNetwork(env.observation_space.shape[0], hidden_size: 256, env.action_space.n).to(self.device)
        # 目标网络，用于稳定训练
        self.target_network = copy.deepcopy(self.q_network)
        # 初始化优化器
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=5e-4)
        # 初始化经验回放缓冲区
        self.replay_buffer = ReplayBuffer(8000)
        # 其他超参数
        self.minimal_size = 200
        self.batch_size = 64
        self.gamma = 0.98  # 折扣因子
        self.epsilon = 0.9  # 初始探索率
        self.epsilon_decay = 0.95  # 探索率衰减率
        self.epsilon_min = 5e-4  # 最小探索率
        self.update_target_every = 10  # 目标网络更新频率
        self.steps = 0
```

```python
def train(self):
    # 训练Q网络
    if len(self.replay_buffer) < self.minimal_size:
        return
    # 从缓冲区中采样一批经验
    transitions, indices = self.replay_buffer.sample(self.batch_size)
    batch = list(zip(*transitions))

    states, actions, rewards, next_states, dones = batch[0], batch[1], batch[2], batch[3], batch[4]
    # 转换数据类型并移动到设备
    states = torch.tensor(np.array(states), dtype=torch.float32).to(self.device)
    actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
    rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
    next_states = torch.tensor(np.array(next_states), dtype=torch.float32).to(self.device)
    dones = torch.tensor(dones, dtype=torch.float32).to(self.device)

    # 计算当前状态的Q值
    q_values = self.q_network(states).gather(1, actions.unsqueeze(1)).squeeze(1)
    # 使用目标网络计算下一个状态的最大Q值
    next_actions = self.target_network(next_states).max(1)[1].unsqueeze(-1)  # DDQN
    next_q_values = self.target_network(next_states).gather(1, next_actions).squeeze(1)  # DDQN
    # 计算目标Q值
    expected_q_values = rewards + (self.gamma * next_q_values * (1 - dones))
    # 计算损失
    loss = nn.MSELoss()(q_values, expected_q_values.detach())
```

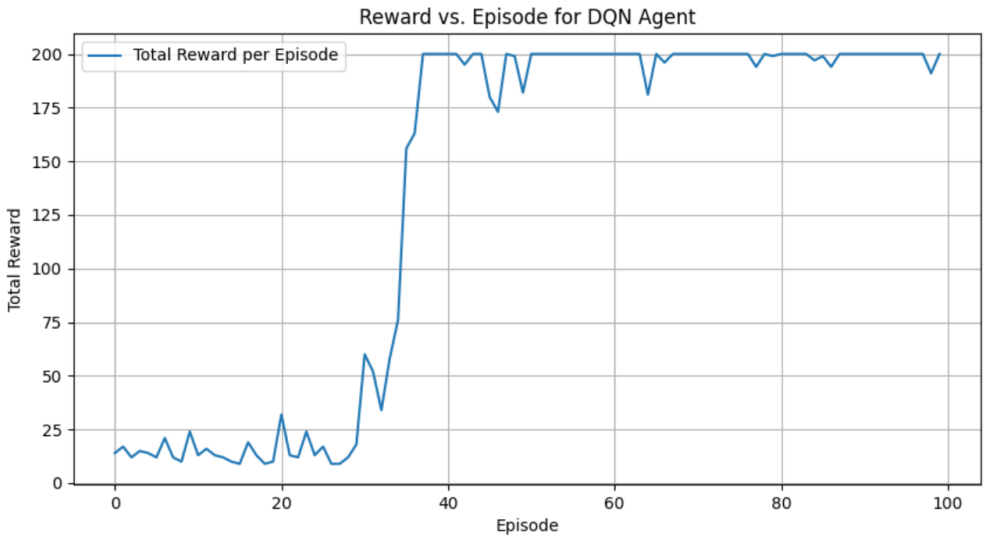
```
    # 优化
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # 更新缓冲区中的优先级
    self.replay_buffer.update_priorities(indices, (q_values - expected_q_values).abs().cpu().detach().squeeze(-1))

    # 定期更新目标网络
    if self.steps % self.update_target_every == 0:
        self.target_network.load_state_dict(self.q_network.state_dict())

1 个用法
def act(self, observation, test=False):
    # 选择动作，根据ε-greedy策略
    if test or random.random() > self.epsilon:
        observation = torch.tensor(observation, dtype=torch.float32).unsqueeze(0).to(self.device)
        with torch.no_grad():
            # 选择Q值最高的动作
            action = self.q_network(observation).argmax().item()
    else:
        # 随机选择动作进行探索
        action = self.env.action_space.sample()
    return action
```

```
def run(self):
    # 运行训练循环
    total_rewards = []
    for episode in range(100):
        state = self.env.reset()
        total_reward = 0
        done = False
        while not done:
            action = self.act(state, test=False)
            # 执行动作并获取反馈
            next_state, reward, done, _ = self.env.step(action)
            # 存储经验到回放缓冲区
            self.replay_buffer.push( *transition: state, action, reward, next_state, done)
            state = next_state
            total_reward += reward
            self.train()
            self.steps += 1

        # 衰减探索率ε
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
        total_rewards.append(total_reward)
        print(f"Episode {episode + 1:3d}, Total Reward: {total_reward}, Epsilon: {self.epsilon:.6f}")

    return total_rewards
```

```python
def draw(total_rewards):
    plt.figure(figsize=(10, 5))
    plt.plot( *args: total_rewards, label='Total Reward per Episode')
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.title('Reward vs. Episode for DQN Agent')
    plt.legend()
    plt.grid(True)
    plt.show()
    return
```

二、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）



2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

reward 是代理与环境交互的关键反馈，通过不断优化以最大化累积奖励，代理可以学习到在给定环境中的最优策略。收敛速度越快，表示学习效率越高。self.epsilon 变量控制着探索和利用的平衡。随着训练的进行，epsilon 逐渐减小，意味着代理将更少地进行随机探索，更多地利用已学到的知识来获取更高的奖励. 在代码的最后，total_rewards 列表存储了每个 episode 的累积奖励，可以用来评估训练过程的效果。最终 reward 收敛到 200 进行较小波动，可以看到为最优策略。

三、 参考资料

【深度强化学习】(1) DQN 模型解析