20337270_钟海财_实验11

中山大学计算机学院

本科生实验报告

(2021学年春季学期)

课程名称：Artificial Intelligence

教学班级	**20级软工+网安**	**专业（方向）**	**软件工程**
学号	**20337270**	**姓名**	**钟海财**

一、 实验题目

1. **(coding) 在 CartPole-v0 环境中实现DQN算法.**

   最终算法性能的评判标准：环境最终的reward至少收敛至180.0.
2. **(optional coding) 在 CartPole-v0 环境中实现A2C算法.（选做）**

   最终算法性能的评判标准：环境最终的 reward至少收敛至180.0.

二、 实验内容

1. 算法原理

1.1 DQN简介

Q-learning算法采用一个Q-tabel来记录每个状态下的动作值，当状态空间或动作空间较大时，需要的存储空间也会较大。如果状态空间或动作空间连续，则该算法无法使用。因此，Q-learning算法只能用于解决离散低维状态空间和动作空间类问题。DQN算法的核心就是用一个人工神经网络q(s,a;w)来代替Q-tabel，即动作价值函数。网络的输入为状态信息，输出为每个动作的价值，因此DQN算法可以用来解决连续状态空间和离散动作空间问题，无法解决连续动作空间类问题。

1.2 DQN算法原理

DQN算法是一种off-policy算法，当同时出现异策、自益和函数近似时，无法保证收敛性，容易出现训练不稳定或训练困难等问题。针对这些问题，研究人员主要从以下两个方面进行了改进。

（1）经验回放：将经验（当前状态s_t、动作a_t、即时奖励r_{t+1}、下个状态s_{t+1}、回合状done)存放在经验池中，并按照一定的规则采样。

（2）目标网络：修改网络的更新方式，例如不把刚学习到的网络权重马上用于后续的自益过程。

1.3 经验回放

经验回放就是一种让经验概率分布变得稳定的技术，可以提高训练的稳定性。经验回放主要有“存储”和“回放”两大关键步骤：

存储：将经验以$(s_t, a_t, r_{t+1}, s_{t+1}, done)$形式存储在经验池中。

回放：按照某种规则从经验池中采样一条或多条经验数据。

从存储的角度来看，经验回放可以分为集中式回放和分布式回放：

集中式回放：智能体在一个环境中运行，把经验统一存储在经验池中。

分布式回放：多个智能体同时在多个环境中运行，并将经验统一存储在经验池中。由于多个智能体同时生成经验，所以能够使用更多资源的同时更快地收集经验。

从采样的角度来看，经验回放可以分为均匀回放和优先回放：

均匀回放：等概率从经验池中采样经验。

优先回放：为经验池中每条经验指定一个优先级，在采样经验时更倾向于选择优先级更高的经验。

1.4 目标网络

对于基于自益的Q学习，动作价值估计和权重w有关。当权重变化时，动作价值的估计也会发生变化。在学习的过程中，动作价值试图追逐一个变化的回报，容易出现不稳定的情况。

目标网络是在原有的神经网络之外重新搭建一个结构完全相同的网络。原先的网络称为**评估网络**，新构建的网络称为**目标网络**。在学习过程中，使用目标网络进行自益得到回报的评估值，作为学习目标。在更新过程中，只更新评估网络的权重，而不更新目标网络的权重。这样，更新权重时针对的目标不会在每次迭代都发生变化，是一个固定的目标。在更新一定次数后，再将评估网络的权重复制给目标网络，进而进行下一批更新，这样目标网络也能得到更新。由于在目标网络没有变化的一段时间内回报的估计是相对固定的，因此目标网络的引入增加了学习的稳定性。

目标网络的更新方式：

上述在一段时间内固定目标网络，一定次数后将评估网络权重复制给目标网络的更新方式为**硬更新(hard update)**，即

$$w_t \leftarrow w_e$$

其中w_t表示目标网络权重，w_e表示评估网络权重。

另外一种常用的更新方式为**软更新(soft update)**，即引入一个学习率τ，将旧的目标网络参数和新的评估网络参数直接做加权平均后的值赋值给目标网络

$$w_t \leftarrow \tau w_e + (1-\tau)\, w_t$$

学习率$\tau \in (0,1)$

2. 伪代码

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

3. 关键代码展示（带注释）

QNetwork类

定义Net类，本质是两个全连接层神经网络构成输入和输出

其中，模型的输入是 states，维度input_size可以由 `env.observation_space.n` 获得。模型的输出是每个 state 的所有 action 的 Q-value，维度output_size由 `env.action_space.n` 获得。

```
class QNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        # nn.Module的子类函数必须在构造函数中执行父类的构造函数
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        pass

    def forward(self, inputs):
        inputs = torch.Tensor(inputs)
        inputs = F.relu(self.fc1(inputs))
        action_v = self.out(inputs)
        return action_v
```

ReplayBuffer类

DQN一个很重要的功能就是要能存储数据，然后在训练的时候minibatch出来。所以，我们需要构造一个存储机制，这里使用类ReplayBuffer来实现。

push函数实现实现经验的存储管理。sample函数实现数据的抽样。

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity

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

    def push(self, *transition):
        if len(self.buffer) == self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, n):
        index = np.random.choice(len(self.buffer), n)
        batch = [self.buffer[i] for i in index]
        return zip(*batch)

    def clean(self):
        self.buffer.clear()
```

AgentDQN类

定义DQN类，主线思路是围绕eval_net和target_net两个网络展开，定义make_action函数实现部分贪婪策略选择action，ReplayBuffer类中的push()函数实现经验的存储管理。

最核心的函数是train函数，当经验池中存放满了经验之后，智能体开始在池中随机抽取经验进行学习，每次抽取BATCH_SIZE的行数，经过整理后得到 obs, actions, rewards, next_obs四个可以被送入神经网络的张量。接下来，将obs送入eval_net得到输出q_eval;将obs_送入target_net得到输出q_next（注意，这次送入神经网络只是为了得到q_next，我们并不关心此刻target_net网络权重参数是否优化，因此我们采用了.detach()方法来阻断反向传播），最后是的常用套路，将q_value和q_target放入loss_function中，然后反向传播，采用Adam优化算法来更新评估网络参数。

run函数则是调用上述函数进行训练神经网络，并将训练过程中的回报值reward进行存储并绘图。

```
class AgentDQN:
    def __init__(self, env, args):
        hidden_size = 128
        input_size = env.observation_space.shape[0]
        output_size = env.action_space.n
        # AgentDQN(env, o_dim, 64, a_dim)
        self.env = env
        # 两个网具有相同的结构
        self.eval_net = QNetwork(input_size, hidden_size, output_size)
        # eval_net:实时更新的Q的network
        self.target_net = QNetwork(input_size, hidden_size, output_size)
        # target_net:每TARGET_REPLACE_ITER轮更新一次的target network
        self.optim = optim.Adam(self.eval_net.parameters(), lr=1e-3)
        self.buffer = ReplayBuffer(10000)
```

```
        # relay buffer 计数器
        self.loss_fn = nn.MSELoss()
        self.learn_step = 0
        self.n_episodes = 1500
        self.gamma = 0.99
        self.print_every = 20
        self.update_target = 100
        self.capacity = 5000
        self.eps = 100
        pass

    def init_game_setting(self):
        pass

    def train(self):
        if self.eps > 0.05:
            self.eps *= 0.99
        if self.learn_step % self.update_target == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step += 1
        batch_size = 128
        obs, actions, rewards, next_obs, dones = self.buffer.sample(batch_size)
        # 从replay buffer中提取出这一个batch的记录
        actions = torch.LongTensor(actions)  # LongTensor to use gather latter
        dones = torch.IntTensor(dones)
        rewards = torch.FloatTensor(rewards)
        # 分别对应了这一个batch中的原始状态，该状态采取的动作，该状态的reward
        q_eval = self.eval_net(obs).gather(-1, actions.unsqueeze(-1)).squeeze(-1)
        # 这里gather的操作是根据squeeze是0还是1选择每一行的0还是1，也就是选择maxQ(s,a)
        q_next = self.target_net(next_obs).detach()
        # detach的作用就是不反向传播去更新
        q_target = rewards + self.gamma * (1 - dones) * torch.max(q_next, dim=-1)[0]
        # Q_target = r + gamma * q_next
        loss = self.loss_fn(q_eval, q_target)
        # pytorch深度学习
        self.optim.zero_grad()
        loss.backward()
        self.optim.step()
        pass

    def make_action(self, observation, test=False):  # test=False
        if np.random.uniform() <= self.eps:
            # 贪婪策略
            action = np.random.randint(0, self.env.action_space.n)
        else:
            action_value = self.eval_net(observation)
            # actions_value 是每个action的Q值 (即Q(s,a))
            action = torch.max(action_value, dim=-1)[1].numpy()
            # 表示Q值最大的action的index
        return int(action)

    def run(self):
        self.init_game_setting()
        scores_deque = deque(maxlen=100)
        max_score = 0
        times = 20
        scores = []
        print('observation space:', self.env.observation_space)
        print('action space:', self.env.action_space)
        for i_episode in range(self.n_episodes):
            obs = self.env.reset()
```

```
            # 重置当前环境状态
            episode_reward = 0
            # 每个episode中的reward
            done = False
            while not done:
                # env.render()
                # 渲染环境
                action = self.make_action(obs)
                # 根据当前的状态s，选择当前回合合适的action
                next_obs, reward, done, info = self.env.step(action)
                self.buffer.push(obs, action, reward, next_obs, done)
                # 状态，该状态使用的a，该状态用a之后的r，之后的状态这些信息存储起来，放入
replay buffer中
                episode_reward += reward
                obs = next_obs
                if self.buffer.len() >= self.capacity:
                    self.train()
            scores_deque.append(episode_reward)
            # 维护可视化得分变化队列
            scores.append(episode_reward)
            if i_episode % self.print_every == 0:
                print('Episode {}\tAverage Score: {:.2f}'.format(i_episode,
np.mean(scores_deque)))
            if np.mean(scores_deque) > max_score:
                max_score = np.mean(scores_deque)
            if np.mean(scores_deque) == 200.0:
                times -= 1
                print('Environment solved in {:d} episodes!\tAverage Score: {:.2f}'
                      .format(i_episode, max_score))
            if times == 0:
                break
        print("max_score is %.2f" % max_score)
        plt.figure().add_subplot(111)
        plt.plot(np.arange(1, len(scores) + 1), scores)
        plt.ylabel('Score')
        plt.xlabel('Episode')
        plt.show()
```

主程序

由于我只实现了DQN，所以直接将主程序写在agent_dqn.py里：

定义环境env为CartPole-v0；

使用该环境定义AgentDQN类型的agent并进行对其训练、输出、绘图。

```
if __name__ == '__main__':
    env = gym.make('CartPole-v0')
    env.seed(0)
    agent = AgentDQN(env, 128)
    agent.run()
```

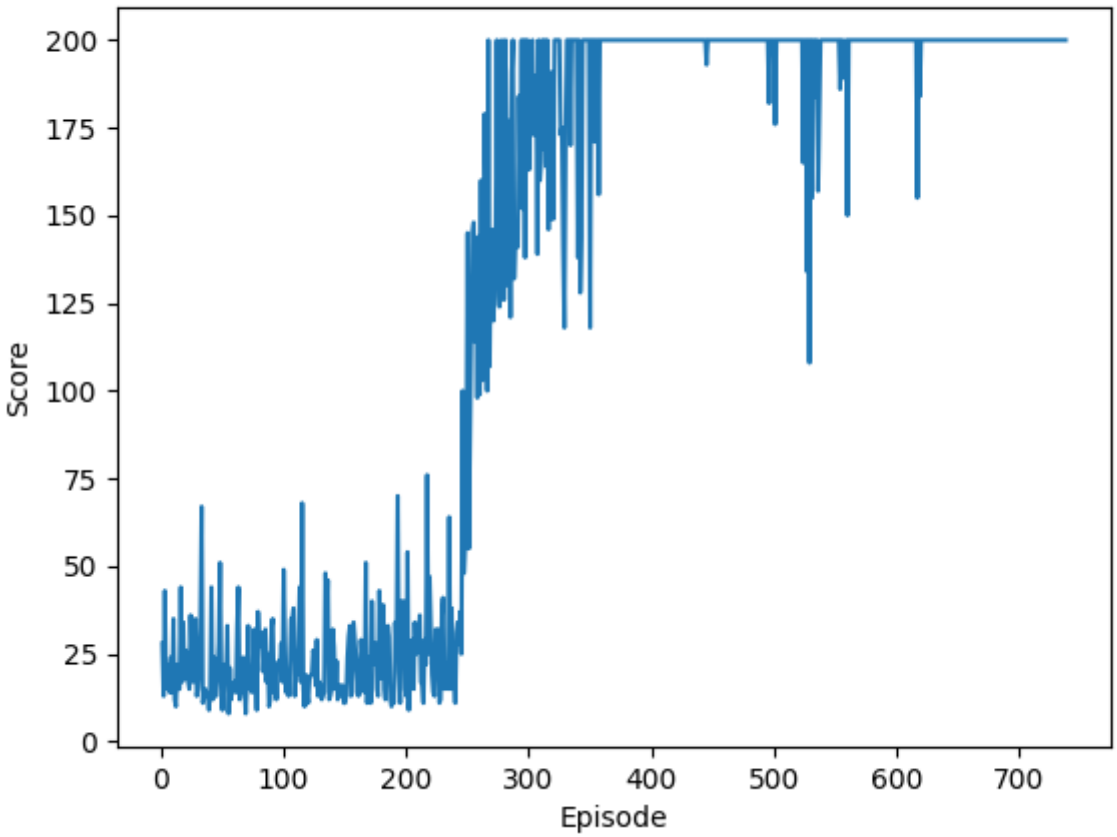
4. 创新点&优化（如果有）

无。

三、 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）

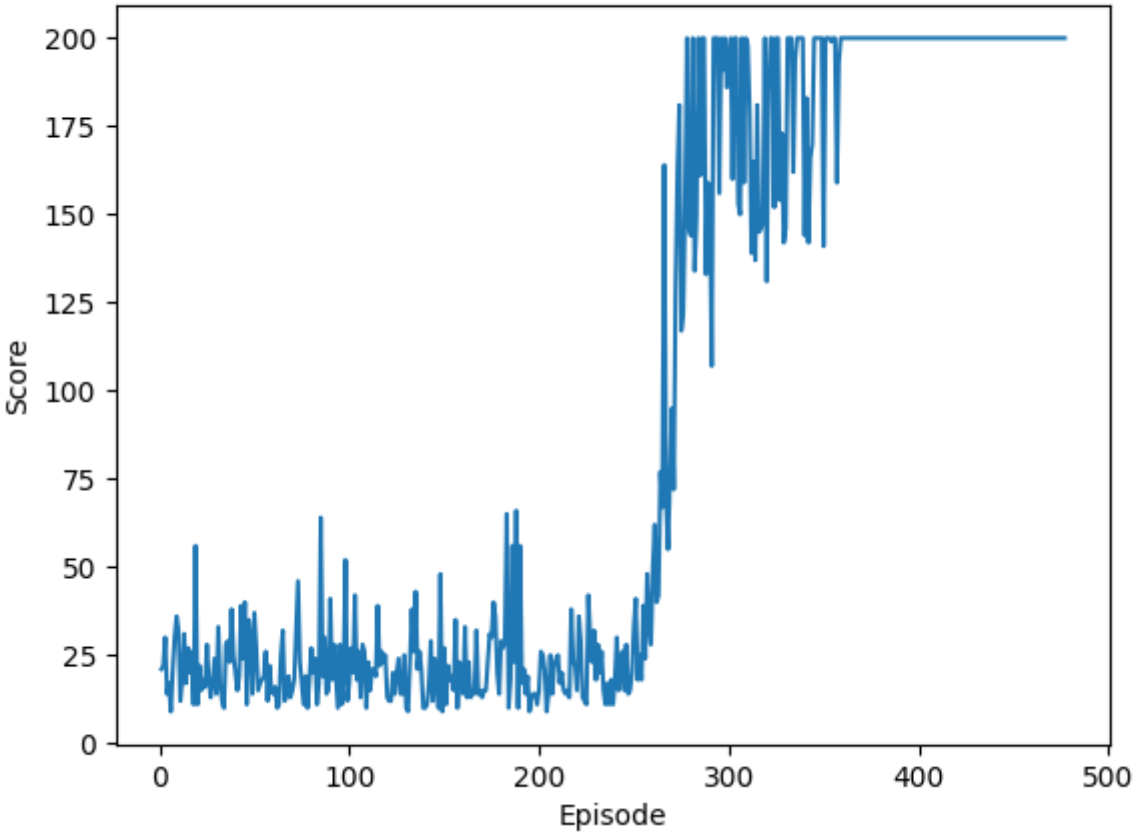
结果1



```
Episode 620 Average Score: 195.42
Episode 640 Average Score: 198.61
Episode 660 Average Score: 199.39
Episode 680 Average Score: 199.39
Episode 700 Average Score: 199.39
Environment solved in 718 episodes! Average Score: 200.00
Environment solved in 719 episodes! Average Score: 200.00
Episode 720 Average Score: 200.00
Environment solved in 720 episodes! Average Score: 200.00
Environment solved in 721 episodes! Average Score: 200.00
Environment solved in 722 episodes! Average Score: 200.00
Environment solved in 723 episodes! Average Score: 200.00
Environment solved in 724 episodes! Average Score: 200.00
Environment solved in 725 episodes! Average Score: 200.00
Environment solved in 726 episodes! Average Score: 200.00
Environment solved in 727 episodes! Average Score: 200.00
Environment solved in 728 episodes! Average Score: 200.00
Environment solved in 729 episodes! Average Score: 200.00
Environment solved in 730 episodes! Average Score: 200.00
Environment solved in 731 episodes! Average Score: 200.00
Environment solved in 732 episodes! Average Score: 200.00
Environment solved in 733 episodes! Average Score: 200.00
Environment solved in 734 episodes! Average Score: 200.00
Environment solved in 735 episodes! Average Score: 200.00
Environment solved in 736 episodes! Average Score: 200.00
Environment solved in 737 episodes! Average Score: 200.00
max_score is 200.00
```

说明：在Episode为280时reward就收敛到了190多，并在Episode为718时reward成功收敛到200.

结果2




```
Episode 380 Average Score: 183.19
Episode 400 Average Score: 187.96
Episode 420 Average Score: 194.20
Episode 440 Average Score: 197.70
Environment solved in 457 episodes! Average Score: 200.00
Environment solved in 458 episodes! Average Score: 200.00
Environment solved in 459 episodes! Average Score: 200.00
Episode 460 Average Score: 200.00
Environment solved in 460 episodes! Average Score: 200.00
Environment solved in 461 episodes! Average Score: 200.00
Environment solved in 462 episodes! Average Score: 200.00
Environment solved in 463 episodes! Average Score: 200.00
Environment solved in 464 episodes! Average Score: 200.00
Environment solved in 465 episodes! Average Score: 200.00
Environment solved in 466 episodes! Average Score: 200.00
Environment solved in 467 episodes! Average Score: 200.00
Environment solved in 468 episodes! Average Score: 200.00
Environment solved in 469 episodes! Average Score: 200.00
Environment solved in 470 episodes! Average Score: 200.00
Environment solved in 471 episodes! Average Score: 200.00
Environment solved in 472 episodes! Average Score: 200.00
Environment solved in 473 episodes! Average Score: 200.00
Environment solved in 474 episodes! Average Score: 200.00
Environment solved in 475 episodes! Average Score: 200.00
Environment solved in 476 episodes! Average Score: 200.00
max_score is 200.00

进程已结束,退出代码0
```

说明：在Episode为290时reward就收敛到了190多，并在Episode为457时reward成功收敛到200.

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）

评测指标：

reward的收敛速度 及 reward最终收敛的值。

分析：

从reward的收敛速度来说我的DQN算法在Episode在300之前就收敛到了190多，并且收敛后的波动较小，可见实现的DQN算法的收敛效果较好。

从reward最终收敛的值来说我的DQN算法最终能够收敛到200，可见我实现的DQN算法的可行性和有效性较好。

四、 思考题

本次实验无思考题。

五、 参考资料

1. 实验文档：17-BaseRL.pdf
2. 课本：《人工智能》(第三版) 清华大学出版社
3. 参考网址：

   Gym中的CartPole环境 - 知乎 (zhihu.com)

   深度强化学习-DQN算法原理与代码 - 代码天地 (codetd.com)

   强化学习算法 DQN 解决 CartPole 问题，代码逐条详解 - 代码天地 (codetd.com)

   DQN(Deep Q Network)及其代码实现 - 知乎 (zhihu.com)