深度学习和强化学习

题目：强化学习在众包任务推荐中的应用

成员：51255903072 陈志强

51255903042 张博宇

51255903055 何泽伟

52265903005 卢官宇

51255903082 周晨沂

院系：数据科学与工程学院

强化学习在众包任务推荐中的应用

一、背景知识

众包（Crowdsourcing）是一种通过将任务或问题外包给广大的普通大众，以获取集体智慧和劳动力的模式。它利用互联网和通信技术，将任务分解成小的子任务，并将这些子任务分发给全球范围内的人们，他们可以自愿参与并为解决问题或完成任务提供贡献。

众包的概念源自于记者 Jeff Howe 于 2006 年提出的一篇《连线》（Wired）杂志的文章，他将其定义为：“一个公司或机构把过去由员工执行的工作任务，以自由自愿的形式外包给非特定的（而且通常是大型的）大众网络的做法。众包的任务通常由个人来承担，但如果涉及到需要多人协作完成的任务，也有可能以依靠开源的个体生产的形式出现。”众包通过将任务交给大量的人群，利用他们的智慧、创造力、时间和资源，有效地解决了传统上由专业人士或有限的团队完成的任务。

二、问题描述

在众包平台中，涉及到两个主要角色：请求者和参与者。请求者在平台上发布任务需求，而参与者则通过该平台选择并完成任务。在众包系统中，当参与者进入平台时，涉及到任务排列的问题，即如何根据参与者选择展示的任务。为简化问题，假设系统只向参与者推荐一个任务。由于系统是盈利性的，因此众包系统需要同时满足参与者和请求者两方的利益。具体任务如下：

（1） 最大化参与者的利益，参与者可以找到更多相关的、感兴趣的任务，以赚取更多的报酬；

（2） 最大化请求者的利益，请求者发布的任务可以得到更多、更高质量的回答。

与此同时，由于参与者和发布者是动态变化的，因此，系统在分配任务时应能考虑并处理这种动态性。

三、数据处理

3.1 原始数据

原始数据包括 worker_quality.csv、project_list.csv 以及 project 和 entry 两个文件夹。project 和 entry 文件夹分别包含了历史的众包工作中 project 和 worker 的

具体信息，worker_quality.csv 和 project_list.csv 中包含的字段及含义如表 1 所示：

表 1-原始数据文件（worker_quality.csv、project_list.csv）字段及含义

文件	字段	描述
worker_quality.csv	worker_id	参与者 id
	work_quality	回答质量
project_list.csv	project_id	项目 id
	project_answer_num	回答次数

3.2 特征提取和数据预处理

从 entry，project 文件夹以及 worker_quanlity.csv，project_list.csv 数据表中读取数据并进行数据的合并和预处理。我们对一些字段进行了筛选，主要选择的字段如表 2 所示：

表 2-数据预处理选择字段及描述

字段	描述
project_id	问题的 id
project_category	问题的类别
project_sub_category	问题的子类别
project_industry	问题的行业
project_entry_count	问题的回答总数
project_start_date	问题的开始时间
project_deadline	问题的结束时间
project_average_score	问题的平均回答分数
project_client_feedback	问题的反馈分数
project_total_awards_and_tips	问题的奖金和小费总额
worker_id	参与者的 id
worker_quanlity	参与者的回答质量
entry_created_at	回答的时间
entry_finalist	回答是否进了问题的 final list
entry_winner	回答是否是问题的 winner
entry_award_value	回答获得的奖金
entry_tip_value	回答获得的小费

其中，对于 project_average、project_client_feedback、project_total_awards_and_tips 等连续特征的字段，我们进行了归一化处理。

此外，针对数据的分析，我们发现数据中存在着同一个参与者多次回答同一个问题的数据，考虑到在后续问题场景的建模和模型训练中，这些重复回答的数据并没有发挥额外的作用，所以我们对这些重复的回答进行去重处理，只记录同一个参与者对同一个问题的 1 个回答，这个回答的 entry_created_at 为重复回答

中的时间最早的值，entry_finalist 和 entry_winner 为重复回答对应字段的或运算结果，entry_award_value 和 entry_tip_value 为对应字段的和。考虑到如果参与者对于一个问题有多次回答，那么该参与者对该问题应该是更感兴趣的，所以我们额外记录了参与者对一个问题的回答次数，用于之后奖励值的构建。根据上述处理，就将 project-entry-worker 的三元关系简化为了 project-worker 的二元关系，从而利于后续问题的建模和模型构建。

3.3 数据集划分

对于所有的回答（entry）数据，先按照参与者（worker）的 id 进行分组，然后对每个参与者回答的问题（project）按照时间进行排序，最终得到每个参与者对应的回答历史。数据集的划分方式为对每个参与者的回答历史分别进行数据集的划分。由于该问题场景和参与者的点击历史有着十分重要的关系，所以为了加快模型收敛以及缓解推荐时的冷启动问题，我们决定划分 20%的数据作为先验数据，60%的数据作为训练集，10%作为验证集，10%作为测试集，即对于每个参与者的历史回答，选择时间排序后前 20%的回答历史作为已知的先验数据，20%-80%的回答历史作为训练集，80%-90%的回答历史作为验证集，90%-100%的回答历史作为测试集。各数据的具体数量如表 3 所示，其中验证集和测试集虽然都是按 10%进行划分，但由于不同参与者的回答历史长度的不同的取整，导致数量略有差别。

表 3-数据集划分描述

数据集	数量
先验数据集	12238
训练集	38527
验证集	6282
测试集	7495

四、概念定义

强化学习（Reinforcement Learning）[1] 属于机器学习的方法论和范式，是智能体（Agent）通过与环境（Environment）交互来实现目标的一种计算方法。智能体在环境的一个状态下做一个动作决策，把这个动作作用到环境中，这个环境发生相应的改变并且将相应的奖励反馈和下一轮的状态返回给智能体。通过这种迭代的交互，智能体的目标是最大化在多轮交互过程中获得的累计奖励的期望。

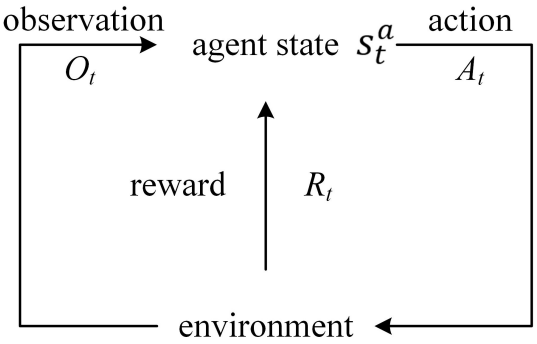

图 1-强化学习架构

图 1 展示了对强化学习整体结构的理解，从当前 Agent 状态s_t^a出发，当完成一个行为A_t后，对环境产生一定的影响，然后它给 Agent 返回一个奖励信号R_t，Agent 可以从中观察到信息O_t，然后进入新的状态，完成一个循环迭代过程。

强化学习中，环境是根据问题场景制定的，而智能体是由具体的算法决定的，不同的算法中，智能体的结构、与环境的交互方式等都各不相同。因此，在本章中，只介绍众包问题场景的环境的构建，而不同算法对应的不同智能体的结构，在第六章中再分别详细介绍。

该众包问题本质上是一个推荐的问题，参与者（worker）就相当于推荐场景中的用户（user），而问题（project）就相当于是推荐场景中的物品（item）。平台要为每个到来的参与者推荐其感兴趣的问题。

根据两个子问题的目标的不同，设计的环境略有不同，我们将每个参与者分开进行处理，对于一个参与者，先用先验数据集中的历史回答数据来初始化该参与者的点击历史。训练过程中，按照时间顺序一步步遍历参与者回答的时间，然后找出该时间下可以回答的所有问题作为候选推荐问题，将当前的回答历史和候选推荐问题作为状态输出给智能体，智能体根据具体的算法做出决策，从候选推荐问题中选择一个问题作为动作返回给环境进行推荐，然后将这个问题加入到参与者的回答历史后进入下一个时间点。如果一个参与者的回答的时间遍历结束，那么就继续下一个参与者的遍历过程，我们将一个参与者的一段完整的遍历过程作为一个回合。通过对数据的分析，我们发现每个问题的 final list 和 winner 对应的参与者数量都比较稀少，并且只有进入 final list 和 winner 的参与者才会获得这个问题的奖励值和小费，所以对于进入问题的 final list 和 winner 的参与者，我们给予额外的奖励值。

具体的环境定义如下：

最大化参与者的利益：

- 状态（state）：当前参与者的历史推荐（包括正确推荐的问题和错误推荐的问题），当前该参与者可以回答的候选问题列表。
- 动作（action）：选择一个问题推荐给当前参与者。
- 奖励（reward）：如果推荐的问题是参与者真实回答过的问题，那么奖励值为该参与者真实回答这个问题的次数。此外，如果参与者进入了这个问题的 final list，那么奖励值额外+5；如果参与者是这个问题的 winner，那么奖励值额外+10。

最大化请求者的利益：

- 状态（state）：当前时间可以回答的所有问题，当前参与者的历史推荐（包括正确推荐的问题和错误推荐的问题）。
- 动作（action）：对于当前的问题列表，选择问题推荐给参与者。
- 奖励（reward）：如果被推荐的参与者是这个问题的真实回答者，那么奖励值为该参与者真实回答这个问题的次数。同样，如果参与者进入了这个问题的 final list，那么奖励值额外+5；如果参与者是这个问题的 winner，那么奖励值额外+10。最终的奖励值再乘上被推荐的参与者归一化后的质量。

五、算法介绍

5.1 DQN 算法（Value-based method）

Q-learning 算法：

Q-learning 算法是一种 Value-based 的强化学习算法，Q 函数$Q(s,a)$表示在状态s下采取行动a所得到的累积回报，即能获得的 Q 值是多少。算法的目标是最大化 Q 值，通过在给定状态下选择最优动作来实现最大化预期奖励。该算法使用 Q-Table 来记录不同状态下不同动作的预估 Q 值。在探索环境之前，Q-Table 会被随机初始化。当 Agent 在环境中进行探索时，它会使用贝尔曼方程（Bellman equation）来迭代更新$Q(s,a)$。随着迭代次数的增多，Agent 会对环境越来越了解，Q 函数也能够被逐渐拟合得更好，直到收敛或达到设定的迭代结束次数。算法伪代码如下：

Algorithm Q-learning

Initialize Q-table with zeros for all state-action pairs

Repeat the following for each episode:

Initialize the current state

Repeat the following until the goal state is reached:

Choose an action (exploration or exploitation) based on the current state and the Q-table

Take the chosen action and observe the next state and the associated reward

Update the Q-value of the current state-action pair using the Q-learning update equation:

$$Q(s,a) = Q(s,a) + \alpha * [R + \gamma * \max_{a'} Q(s',a') - Q(s,a)]$$

Update the current state to the next state

End episode

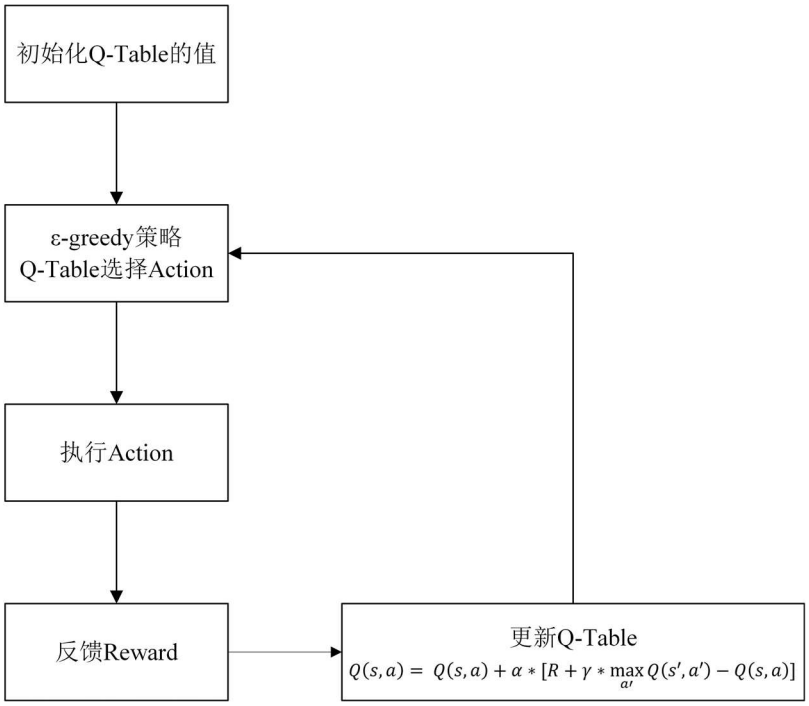

图 2-Q-learning 算法流程

传统的 Q-learning 算法使用一个 Q-Table 来存储所有状态和动作的 Q 值，但是对于大型状态空间的问题，需要大量内存来存储 Q-Table。而 DQN 使用神经网络来近似 Q 函数，将状态作为输入，输出对应的动作 Q 值。

Deep Q-network 算法：

DQN 算法的主要做法是 Experience Replay，其将系统探索环境得到的数据储存起来，然后随机采样样本更新深度神经网络的参数。

DQN 的改进主要有以下几点：

（1）使用神经网络逼近 Q 函数：DQN 使用神经网络来逼近 Q 函数，将状态作为输入，输出对应的动作 Q 值。这样可以处理大型状态空间的问题，并且

能够泛化到未见过的状态。

（2）使用经验回放：DQN 使用经验回放存储 Agent 与环境的交互过程，包括状态、动作、奖励、下一个状态等信息。然后在训练过程中，从回放缓冲区中随机抽取样本进行训练，可以减小样本间的相关性，提高样本的利用率。

（3）使用目标网络：DQN 还使用了两个网络，一个是主网络用于对当前状态的 Q 值估计，另一个是目标网络用于计算目标 Q 值。目标网络的参数会定期更新，以使得训练过程更加稳定。

（4）使用ε-greedy 策略：DQN 使用ε-greedy 策略来平衡探索和利用。在训练过程中，Agent 有ε的概率以随机动作探索环境，有 1-ε的概率根据当前 Q 值选择最优动作。

DQN 算法伪代码[2]如下：

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

5.2 策略梯度算法（Policy-based method）

策略梯度算法是强化学习中的一种算法，其基本思想是通过梯度上升的方式来优化策略。具体来说，策略梯度算法首先定义一个策略函数，该函数可以根据当前的状态生成一个动作。然后，通过与环境的交互，收集到一系列的状态、动作和奖励，用这些数据来估计策略函数的梯度，然后沿着梯度的方向更新策略函数，以此来提高策略的性能。

算法原理：

Algorithm policy gradient

Initialize θ arbitrarily
for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**

for $t = 1$ to $T - 1$ **do**

$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$

end for

end for

return θ

其中，关键的部分是神经网络更新梯度$\alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$，$\alpha$即为学习率，$\nabla_\theta \log \pi_\theta(s_t, a_t) v_t$是指策略在$\theta$上的梯度，$\log \pi_\theta(s_t, a_t)$是状态$s_t$下选择行动$a_t$概率的 log 值，$v_t$是在状态$s_t$下选择行动$a_t$所获得的 reward。

策略梯度算法相比于基于值的方法（DQN、Q-learning）相比，优势在于：策略梯度适用于大规模连续动作空间、高维离散动作空间的问题。然而策略梯度依赖蒙特卡洛采样获取比较稀疏的反馈。基于价值的方法可以利用丰富的环境反馈(奖励)。在反馈较丰富的问题上，基于价值的方法可能收敛更快。

5.3 Actor-Critic 算法

Actor-Critic 算法结合了策略梯度方法(Actor)与状态-动作价值函数估计(Critic)。Actor 的前身是 Policy Gradient，这样就可以在很容易地在连续动作中选取更合适的动作（但 Q-Learning 不适合此任务）；Critic 的前身是 Q-Learning 或者其他的以值为基础的学习法，能进行单步更新，而更传统的 Policy Gradient 则是回合更新，这降低了学习效率。

该算法的主要思想如图 3 所示。Actor: 用于 Parameterized 策略函数，输出动作的概率分布；Critic: 估计状态-动作值函数$V(s)$或$Q(s, a)$。

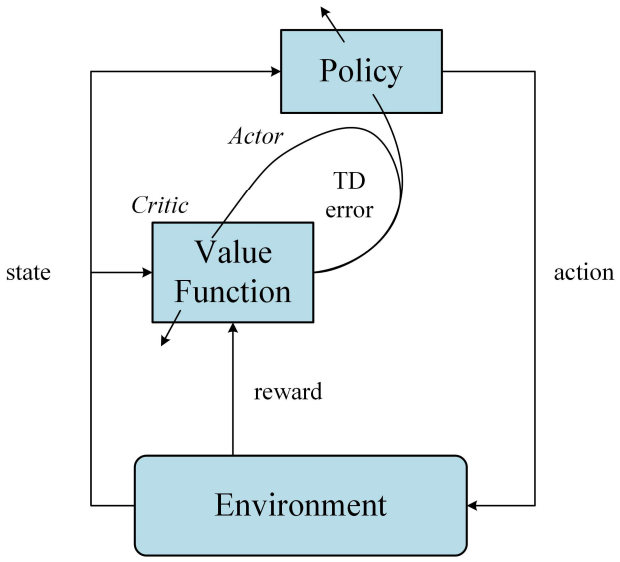

图 3-Actor-Critic 算法主要思想

算法流程：

（1）探索环境，执行动作，观察状态s和奖励r；

（2）Critic 根据当前策略评估$V(s)$或$Q(s,a)$；

（3）使用 Critic 评估结果更新 Actor 的参数,通常使用策略梯度提升算法；

（4）重复上述流程,不断提升 Actor 的策略。

Actor-Critic 算法的优势在于：（1）Actor 专注优化策略，Critic 专注估计价值；（2）Critic 提供的额外反馈可以减少方差，提高学习效率；（3）可以处理连续动作空间。缺点在于：（1）需要同时训练两个模型，稍复杂；（2）算法稳定性依赖于 Critic 的准确度。

六、模型设计及效果评估

在本节，我们描述了基于 DQN 算法、策略梯度算法以及 Actor-Critic 算法的模型详细设计过程，以及和随机策略的结果比较。

6.1 基于 DQN 算法的模型设计（Value-based method）

在本问题环境中，状态（state）是参与者的历史回答和候选问题，动作（action）是要在候选问题中选择一个问题进行推荐，状态的数量空间十分庞大，且不同的状态有着不同的候选问题，显然使用传统的 Q-learning 算法用一个 Q-Table 来存储所有状态和动作的 Q 值不现实的。所以我们决定使用 DQN 算法，用神经网络来拟合逼近 Q 值。

由于状态中的参与者历史回答和候选问题都是与问题相关的，所以我们先设计了一个 ProjectEncoder 用于对问题的编码，具体模型结构如图 4 所示。

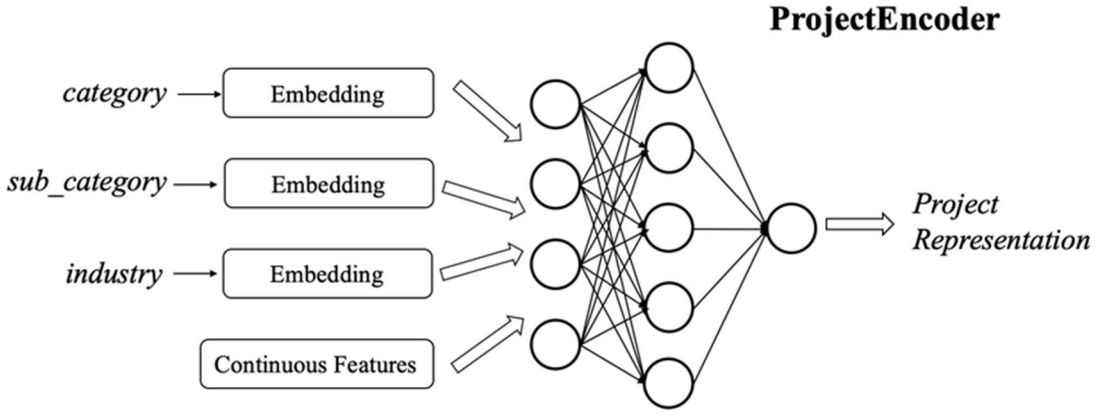

图 4-ProjectEncoder 模型结构

DQN 需要有一个 Q 网络输出$Q(s,a)$，即输入当前的状态（state）和动作

（action），网络要输出一个值表示对应的 Q 值，Q 网络的模型结构如图 5 所示。

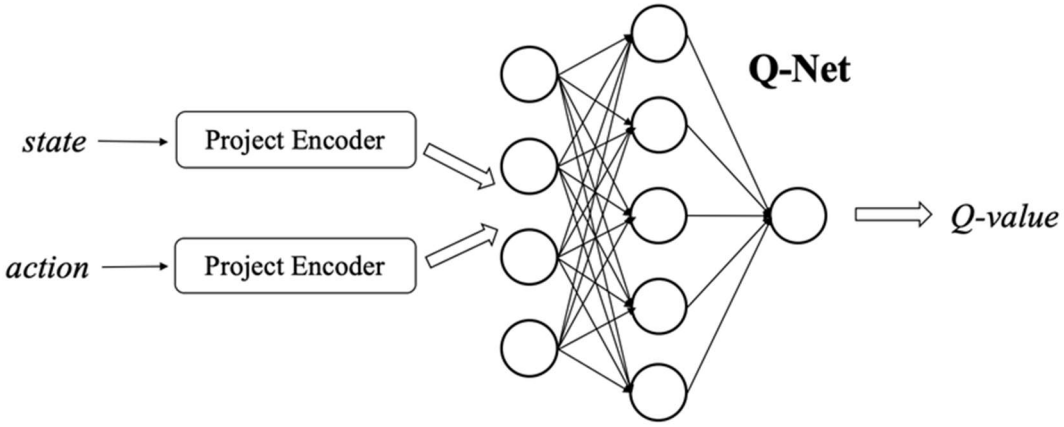

图 5-Q 网络模型结构

Q 网络为智能体（agent）的主要结构，在智能体进行训练的过程中，我们使用了 ε-greedy 策略来平衡探索和利用。在训练过程中，Agent 有 ε 的概率以随机动作探索环境，有 1-ε 的概率根据当前 Q 值选择最优动作，而 ε 会随着训练的迭代步数增加而不断降低，具体实现的代码为：

```
@torch.no_grad()
def take_action(self, state, mode="train"):
    worker_history, action_list = state
    if mode == "train" and np.random.random() < self.epsilon + 1.0 / (self.count / 10 + 1.5):
        res_action = random.choice(action_list)
    else:
        res_action = None
        max_q_value = 0.
        for action in action_list:
            project_index, discrete, continuous = action
            q_value = self.q_net(worker_history, (discrete, continuous)).item()
            if q_value > max_q_value:
                res_action = action
                max_q_value = q_value
        if res_action is None:
            res_action = random.choice(action_list)
    return res_action
```

另外，我们参考了 DQN 的论文，使用经验回放存储 Agent 与环境的交互过程，然后在训练过程中，从回放缓冲区中随机抽取样本进行训练，减小样本间的相关性，提高样本的利用率。实现的经验回放池的代码如下：

```
class ReplayBuffer:
```

```
    ''' 经验回放池 '''
    def __init__(self, capacity):
        self.buffer = collections.deque(maxlen=capacity)  # 队列,先进先出
        self.cnt = 0

    def add(self, state, action, reward, next_state, done):  # 将数据加入 buffer
        self.buffer.append((state, action, reward, next_state, done))
        self.cnt += 1

    def sample(self, batch_size):  # 从 buffer 中采样数据,数量为 batch_size
        transitions = random.sample(self.buffer, batch_size)
        return transitions

    def size(self):  # 目前 buffer 中数据的数量
        return len(self.buffer)
```

我们还使用了目标网络来使训练过程更加稳定，一个主网络用于对当前状态的 Q 值估计，另一个目标网络用于计算目标 Q 值，目标网络的参数会定期更新。实现的 DQN 更新的代码如下：

```
def update(self, transitions):
        loss_list = list()
        for state, action, reward, next_state, done in transitions:
            worker_history, action_list = state
            project_index, discrete, continuous = action

            q_values = self.q_net(worker_history, (discrete, continuous))
            if done:
                q_target = torch.Tensor([reward])
            else:
                next_worker_history, next_action_list = next_state
                max_next_q_values = max([self.target_q_net(next_worker_history, (next_discrete,
next_continuous))
                                         for next_project_index, next_discrete, next_continuous in
next_action_list])
                q_target = reward + self.gamma * max_next_q_values
            loss_list.append(F.mse_loss(q_values, q_target.detach()).view(-1))
        dqn_loss = torch.mean(torch.cat(loss_list, dim=-1), dim=-1)
        self.opt.zero_grad()
        dqn_loss.backward()
        self.opt.step()

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

```
                )
            self.count += 1
            return dqn_loss.cpu().item()
```

训练过程中，最大化参与者利益和最大化请求者利益的 Q 网络的损失函数变化曲线分别如图 6 和图 7 所示：

图 6-最大化参与者利益下 DQN 的 loss 曲线变化

图 7-最大化请求者利益下 DQN 的 loss 曲线变化

从图 6 和图 7 可以清楚地观察到随着迭代次数的增加，损失值出现一定的波

动并且有上升趋势（但是在下面的结果分析显示我们设计的 DQN 模型要优于随机策略），我们对这种现象分析了原因，认为可能得原因是模型训练欠拟合，所以 loss 值随着迭代会有上升现象；或者是因为值函数估计不稳定，即对于相同的状态或状态动作对，估计值变化较大，那么训练过程中的 loss 也会出现波动。这可能是由于近似值函数的表示能力不足或训练目标的选择等原因导致的。

为了显示 DQN 的训练效果，我们使用随机策略作为对比，比较两个策略在训练过程中获得的奖励值（reward）的变化。图 8 和图 9 展示了设计的 DQN 模型和 Random 模型随着迭代次数的增加，其奖励值的变化情况。

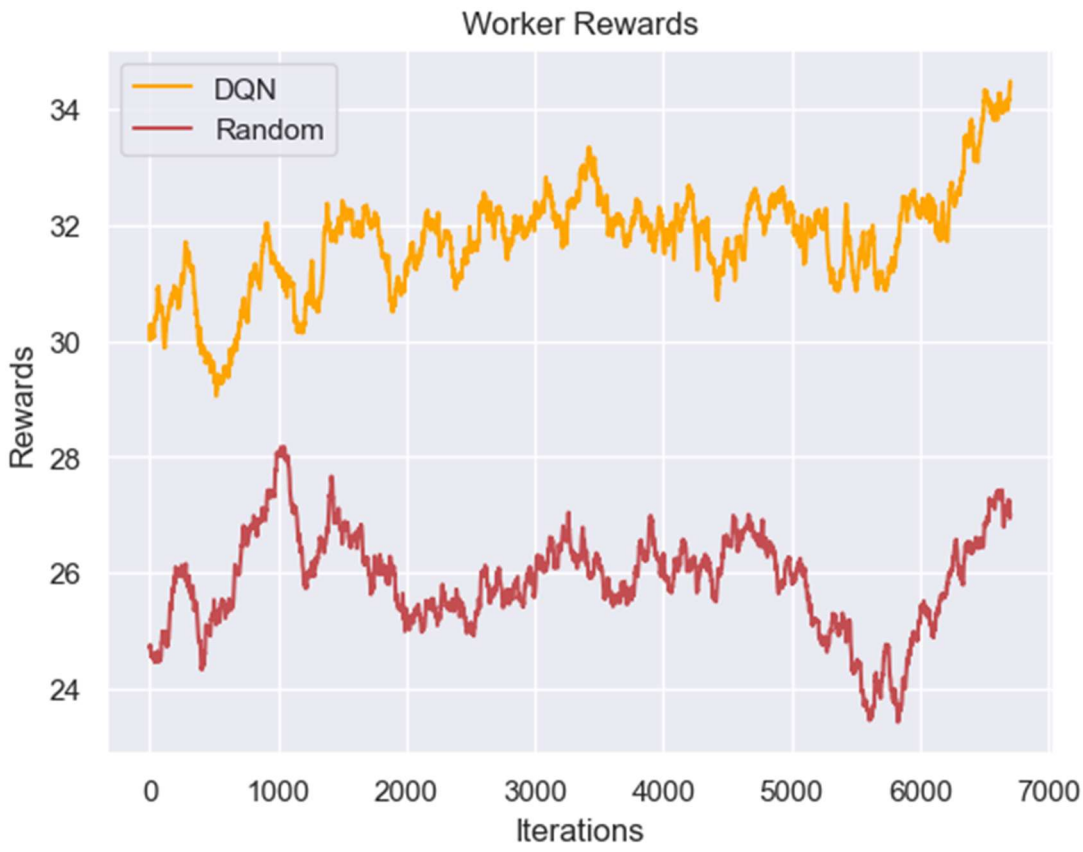

图 8-最大化参与者利益下 DQN 和 Random 模型奖励值的曲线变化

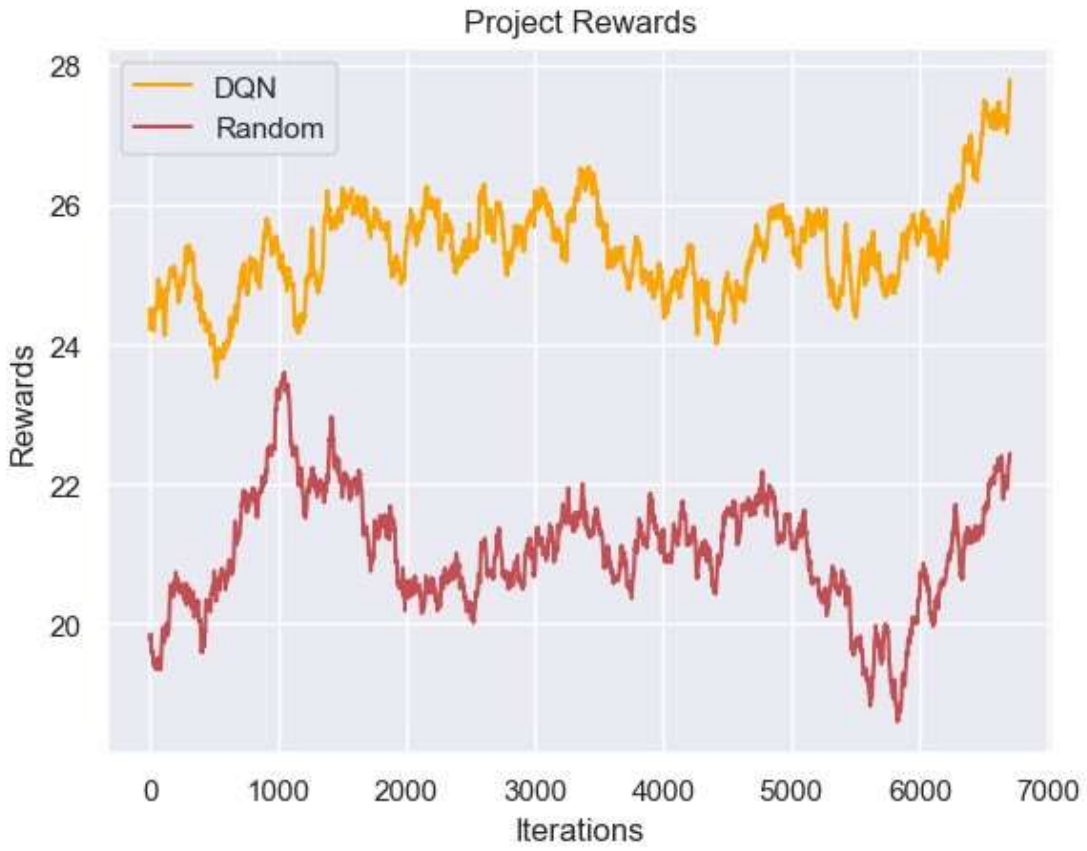

图 9-最大化请求者利益下 DQN 和 Random 模型奖励值的曲线变化

通过观察图 8 和图 9，可以得出一个明显的结论：DQN 模型在训练过程中明显优于随机策略，获得的奖励总体上明显高于随机策略所获得的奖励。可以清楚地看到，DQN 模型在训练过程中持续地获得较高的奖励。相比之下，随机策略的奖励水平相对较低且波动较大。

6.2 基于策略梯度算法的模型设计（Policy-based method）

该算法基本思想是通过梯度上升的方式来优化策略。

在本项目中，状态是参与者（worker）的被分配历史的每个项目（project），策略是指给予当前参与者（worker）分配哪个项目（project）。基于策略的模型输入是智能体的状态（state），也就是 worker 的历史被分配项目情况。

与 DQN 一样，worker 的状态需要经过 ProjectEncoder 向量化后再进入我们的策略梯度网络。策略梯度网络的核心思想是对策略$\pi(a \mid s, \theta)$建模。也就是说策略网络的输入是智能体的状态，而输出是决策的概率分布。因此我们设计了一个网络—PolicyNet 来拟合这个策略分布。模型结构如图 10 所示：

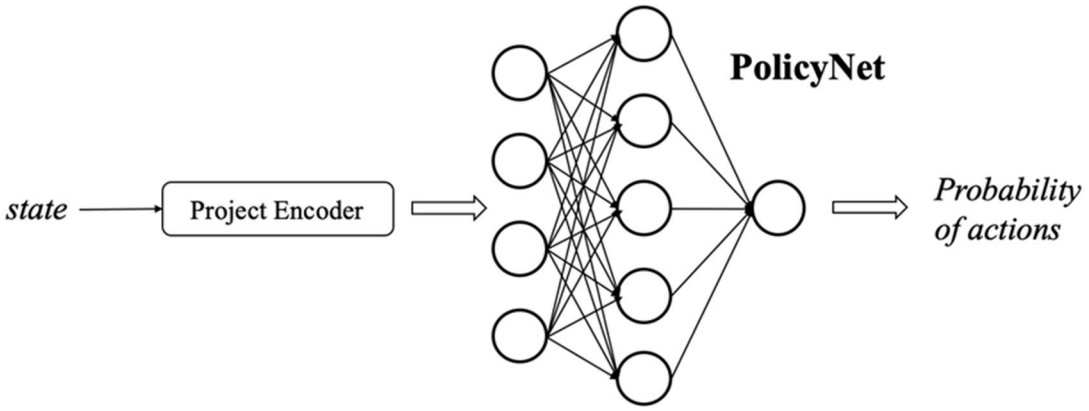

图 10-PolicyNet 模型结构

设计的 PolicyNet 具体代码如下：

```
class PolicyNet(nn.Module):
    "策略网络（全连接网络）"
    def __init__(self, num_projects, len_category, len_sub_category, len_industry, dim):
        """ 初始化策略网络，为全连接网络
            len_category: 父标签向量的长度
            len_sub_category: 子标签向量长度
            len_industry: 子子标签向量长度
            dim: embedding_dim
        """
        super(PolicyNet, self).__init__()
        self.project_encoder = ProjectEncoder(len_category, len_sub_category, len_industry, dim)
        self.empty_weight = nn.Parameter(torch.Tensor(dim))
        self.mlp = nn.Sequential(
            nn.Linear(dim, dim * 4),
            nn.ReLU(),
            nn.Linear(dim * 4, num_projects),
        )

    def forward(self, worker_history, action_list):
        if len(worker_history) == 0:
            worker_out = self.empty_weight
        else:
            worker_out = torch.mean(torch.cat([self.project_encoder(state) for state in
worker_history], dim=0).view(len(worker_history), -1), dim=0)
        mlp_input = worker_out
        mlp_out = self.mlp(mlp_input)
        mask_index = torch.tensor([action[0] for action in action_list])
        return masked_softmax(mlp_out, mask_index)
```

在本项目中我们使用的是一个单隐藏层的 MLP 去对策略的概率分布进行拟

合。MLP 的输出空间为项目（project）的总数，也就是 2501。

在模型训练的初始阶段，智能体为 PolicyNet 提供状态变量，经过网络计算后获得了对策略的预测概率分布。但考虑到一个 worker 的策略空间其实仅有该 worker 在线时间内存在的项目，所以我们的决策空间从原始决策空间变为了该 worker 当前的候选决策空间，故在我们得到策略空间的概率分布之后需要将非候选决策 mask 掉，再在候选决策空间做 softmax 操作，才能得到 worker 真实决策的概率分布。实现该操作的代码如下：

```
def masked_softmax(X, mask_index=None, value=-1e6):
    """
        遮盖 softmax
        X: (num_projects, )
        mask_index: (num_actions, ) 不需要遮盖的 index
    """
    if mask_index is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        mask_index = torch.tensor(list(set(torch.arange(X.shape[0]).numpy()) - set(mask_index.numpy())))
        X[mask_index] = value
    return nn.functional.softmax(X, dim=-1)
```

在得到 worker 决策概率分布之后，我们需要对决策进行采样，并计算当前的奖励值，以计算策略网络参数的梯度。待采样一个回合后，实现策略网络的参数更新。具体训练过程的代码如下：

```
def new_train(config, env, agent):
    start_time = time.time()
    print(f"环境名：{config.env_name}, 算法名：{config.alg}")
    print("开始训练智能体.....")
    # 记录每个 epoch 的奖励

    # 每 config.update_fre 记录一次
    reward_list = []
    loss_list = []

    for epoch in range(config.epochs):
        iteration_return = 0
        step = 0
        env.reset()
        with tqdm(total=config["worker_num"], desc='Episodes %d' % epoch) as worker_bar:
            for worker_iter in range(config.worker_num):
```

```
                    # 进行一个回合
                    state, done = env.get_obs()
                    while not done:
                        action = agent.sample_action(state)
                        next_state, reward, done = env.step(action)
                        agent.memory.push((state, action, reward))
                        step += 1
                        state = next_state
                        iteration_return += reward
                        if step % config.update_fre == 0:
                            reward_list.append(iteration_return)
                            iteration_return = 0
                    if len(agent.memory) != 0:
                        loss = agent.update()
                        loss_list.append(loss)
                    if (worker_iter + 1) % 10 == 0:
                        worker_bar.set_postfix({'episode': '%d' % (epoch + 1),
                                                'worker': '%d' % (worker_iter + 1),
                                                "loss_list": '%.3f' % np.mean(loss_list[-10: ]),
                                                "reward_list" : '%.3f' %
np.mean(reward_list[-10: ])})
                    env.worker_index += 1
                    env.worker_list_pos = 0
                    worker_bar.update(1)

        return reward_list, loss_list
```

策略智能体的 update 函数具体代码如下：

```
def update(self):
        state_pool, action_pool, reward_pool = self.memory.sample()
        state_pool, action_pool, reward_pool = list(state_pool), list(action_pool), list(reward_pool)
        # 对奖励进行修正，考虑未来，并加入衰减因子
        running_add = 0
        for i in reversed(range(len(reward_pool))):
            if reward_pool[i] == 0:
                running_add = 0
            else:
                running_add = running_add * self.gamma + reward_pool[i]
                reward_pool[i] = running_add
        self.optimizer.zero_grad()
        loss_list = []
        for i in range(len(reward_pool)):
            state = state_pool[i]
            action = torch.tensor(action_pool[i][0])
            reward = reward_pool[i]
```

```
                probs = self.policy_net(state[0], state[1])
                m = Categorical(probs)
                #  加权(reward)损失函数，加负号(将最大化问题转化为最小化问题)
                loss = -m.log_prob(action) * (reward + 1e-6)
                loss.backward()
                loss_list.append(loss.detach())
            self.optimizer.step()
            self.memory.clear()
            return np.array(loss_list).mean()
```

训练过程中，最大化参与者利益和最大化请求者利益的策略网络的损失函数变化曲线分别如图 11 和图 12 所示：

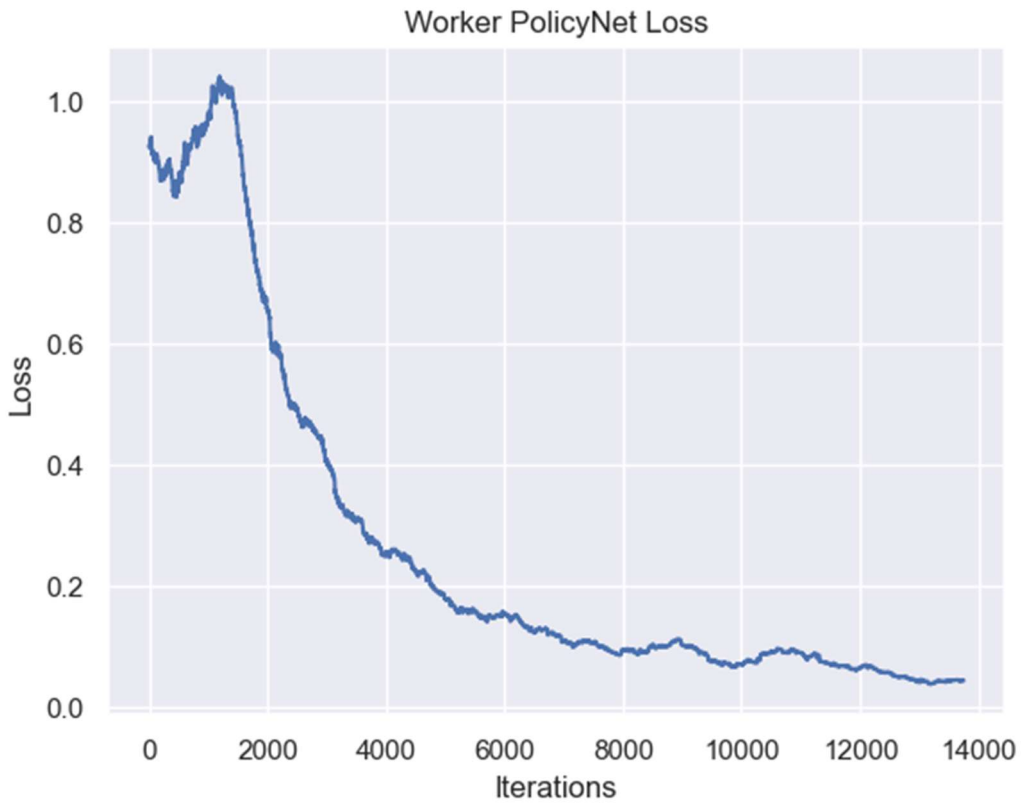

图 11-最大化参与者利益下 PolicyNet 的 loss 曲线变化

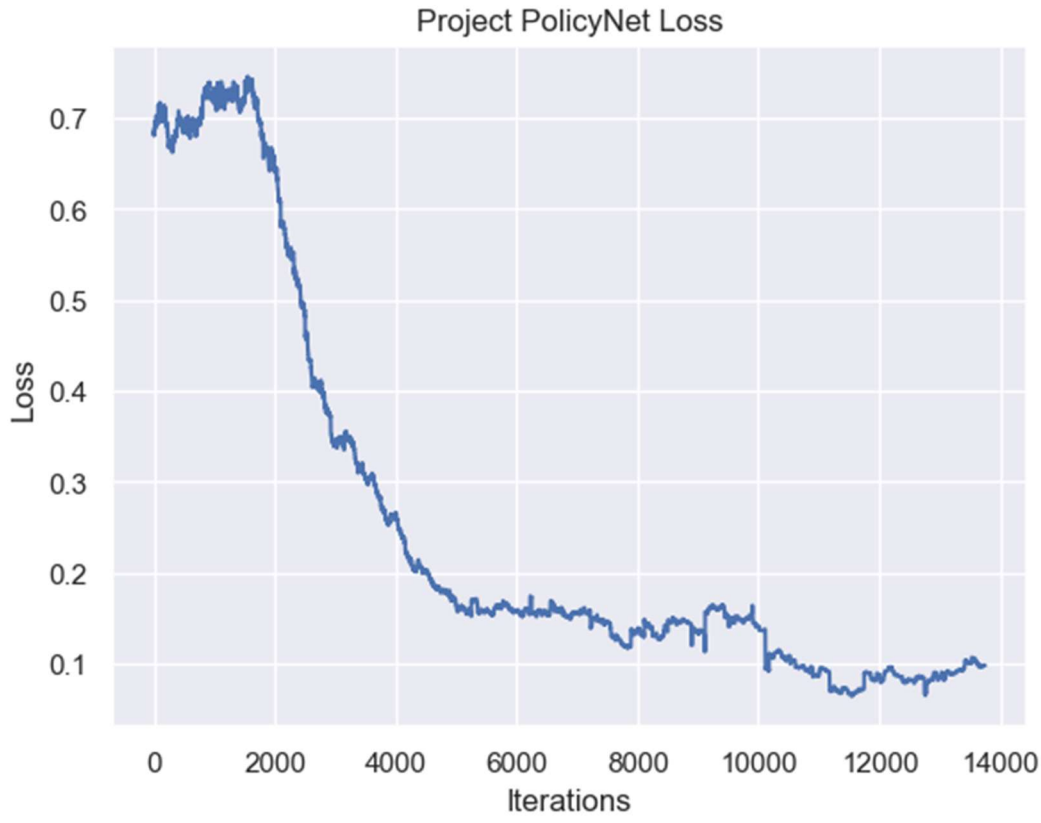

图 12-最大化请求者利益下 PolicyNet 的 loss 曲线变化

从图 11 和图 12 可以观察到，随着迭代次数的增加，最大化参与者利益和最大化请求者利益的策略网络的损失逐渐减小，表明模型逐渐收敛。

为了显示 PolicyNet 的训练效果，我们依然使用随机策略作为对比，比较两个策略在训练过程中获得的奖励值（reward）的变化。图 13 和图 14 展示了设计的 PolicyNet 模型和 Random 模型随着迭代次数的增加，其奖励值的变化情况。

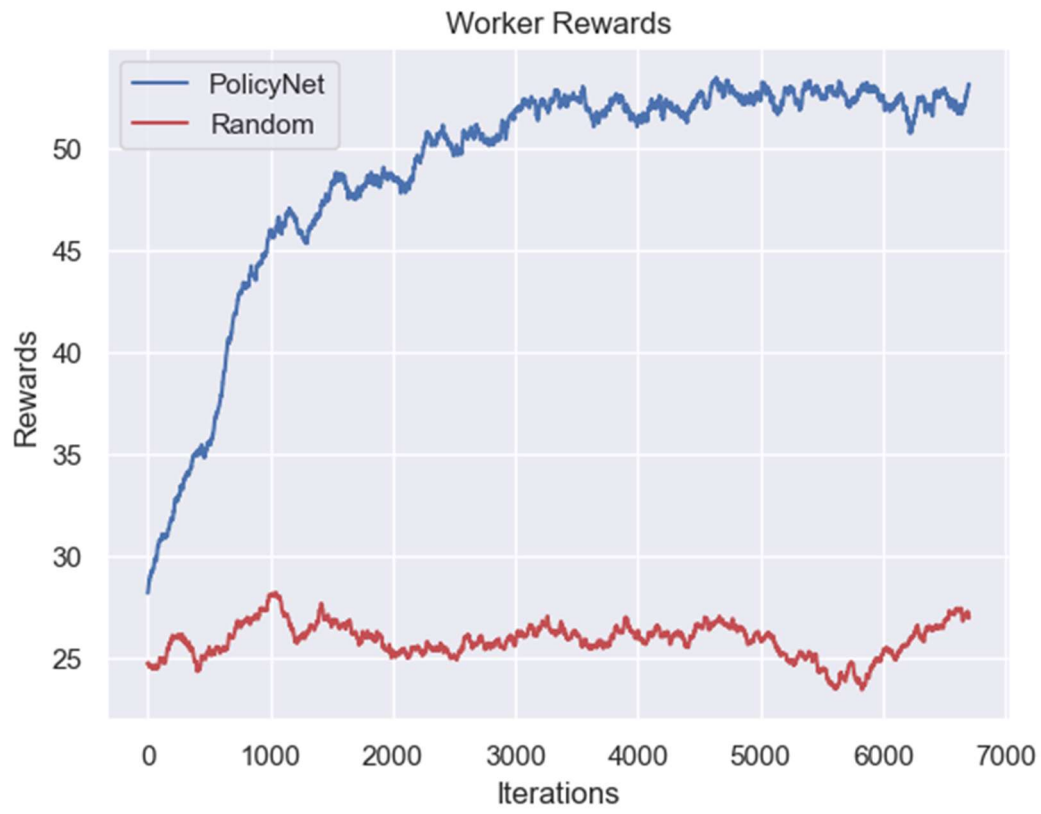

图 13-最大化参与者利益下 PolicyNet 和 Random 模型奖励值的曲线变化

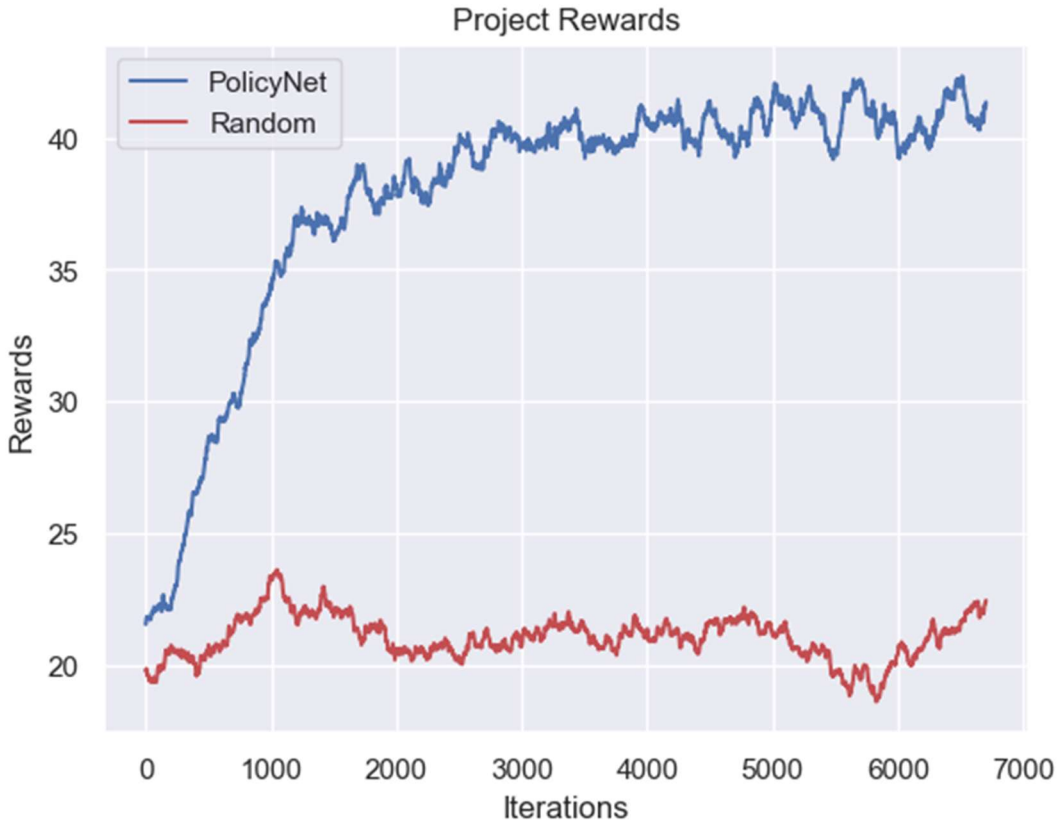

图 14-最大化请求者利益下 PolicyNet 和 Random 模型奖励值的曲线变化

通过观察图 13 和图 14，我们可以得出一个显著的结论：无论是追求最大化参与者利益还是最大化请求者利益，PolicyNet 在训练过程中获得的奖励总体上明显高于随机策略所获得的奖励。PolicyNet 在整个训练过程中取得了稳步的提升。与此相反，随机策略的奖励水平相对较低且波动较大。这些结果表明 PolicyNet 的训练是成功的，它能够学习到更好的策略，从而获得更高的奖励。相比之下，随机策略无法产生一致的、高质量的决策，导致获得的奖励较低。

6.3 基于演员评论家算法的模型设计（Actor-Critic method）

所设计的基于 Actor-Critic method 的模型的整体结构如图 15 所示：

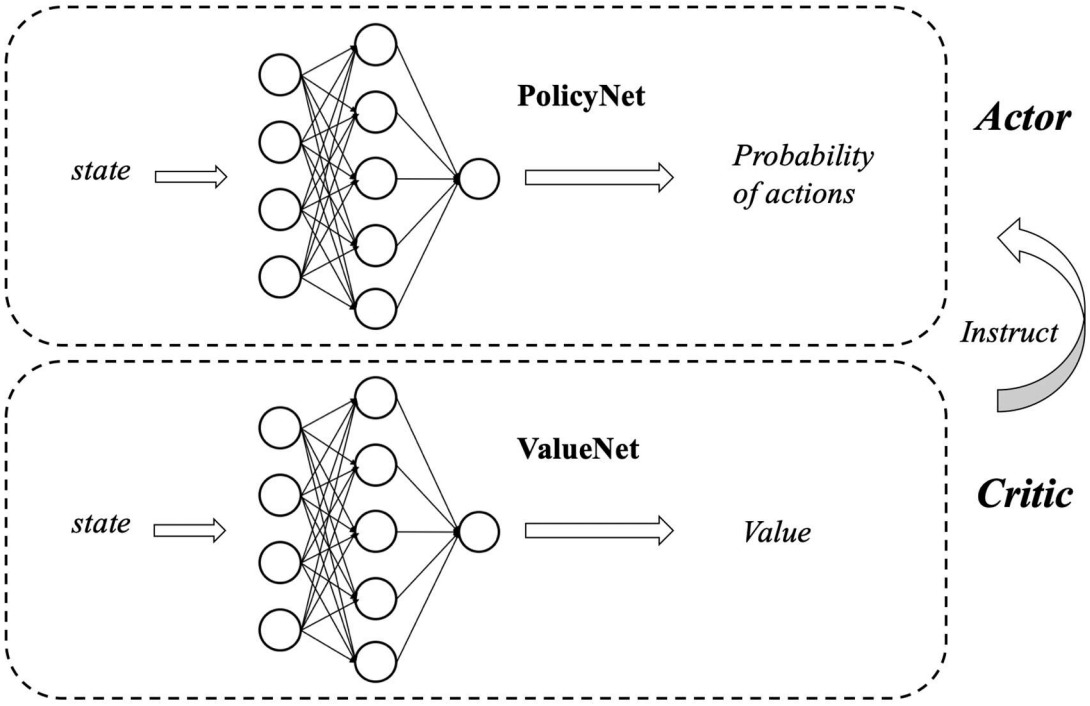

图 15-Actor-Critic 模型整体结构

在 REINFORCE 算法中，目标函数包含了一项轨迹回报，用于指导策略的更新。但通过蒙特卡洛采样的方法对策略梯度的估计是无偏的，方差非常大。

在 Actor-Critic 算法中，我们引入一个值函数来提供更准确的指导，并引导策略学习的过程。我们将梯度表示为以下形式：

$$g = \mathbb{E}\left[\sum_{t=0}^{T}\left(r_t + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\nabla_\theta \pi_\theta(a_t|s_t)\right]$$

其中$r_t + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)$为时序差分残差。Actor-Critic 算法中的 Actor 部分依据上公式，采用策略梯度进行更新。

对于 Critic 部分，我们将 Critic 价值网络表示为V_w，参数为w。于是，我们可以采取时序差分残差的学习方式，对于单个数据定义如下价值函数的损失函数：

$$\mathcal{L}(w) = \frac{1}{2}\left(r + \gamma V_w(s_{t+1}) - V_w(s_t)\right)^2$$

与 DQN 中一样，我们采取类似于目标网络的方法，将上式中$r + \gamma V_w(s_{t+1})$作为时序差分目标，不会产生梯度来更新价值函数。因此，价值函数的梯度为：

$$\nabla_w \mathcal{L}(w) = -\left(r + \gamma V_w(s_{t+1}) - V_w(s_t)\right)\nabla_w V_w(s_t)$$

下面从 Worker 的角度出发，描述算法细节。

在本实验中，用于对 project 进行编码的神经网络与前两个算法所使用的一致，不再进行赘述。

Critic 部分，我们采用的用来估计 Value 的神经网络主要使用了三层神经网络来进行学习。神经网络的输入为，Worker 的当前状态（即当前参与者的历史推荐）经过 Encoder 后的平均数；输出为一个标量，表示当前状态的 value。具体代码如下：

```
class ValueNetForAC(nn.Module):
    # ActorCritic 算法的 Value 网络
    def __init__(self, dim):
        super(ValueNetForAC, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(dim, dim * 4),
            nn.ReLU(),
            nn.Linear(dim * 4, dim * 4),
            nn.ReLU(),
            nn.Linear(dim * 4, 1),
        )
        model_init(self)

    def forward(self, worker_state):
        mlp_out = self.mlp(worker_state)
        return mlp_out
```

Actor 部分，其策略网络与 6.2 节中的 PolicyNet 大致相同。输入也是 Worker 的当前状态，输出维度为 2501，代表项目（project）的总数。然后也用了先 mask，再 softmax 的操作。

```
class PolicyNetForAC(nn.Module):
    # ActorCritic 算法的 Policy 网络
    def __init__(self, num_projects, dim, device):
```

```
        """ 初始化策略网络，为全连接网络
            num_projects: 可选择的动作数量
            dim: embedding_dim
        """
        super(PolicyNetForAC, self).__init__()
        self.device = device
        self.mlp = nn.Sequential(
            nn.Linear(dim, dim * 4),
            nn.ReLU(),
            nn.Linear(dim * 4, dim * 4),
            nn.ReLU(),
            nn.Linear(dim * 4, num_projects),
        )

    def forward(self, worker_state, worker_action_list):
        mlp_out = self.mlp(worker_state)
        for idx, action_list in enumerate(worker_action_list):
            effective_idx = set(action[0] for action in action_list)
            mask_index = torch.tensor(list(set(np.arange(mlp_out.shape[-1])) -
effective_idx)).to(self.device)
            mlp_out[idx][mask_index] = -1e6

        output = nn.functional.softmax(mlp_out, dim=-1)
        return output
```

Agent 部分。take_action 部分，与 6.2 节相似。输入 Worker 的当前状态到神经网络，然后根据返回的各个 action 概率进行采样。将采样到的行为，作为 agent 的行为，返回给环境。update 部分，输入为(s, a, r, s')。根据定义，s包括 woker 在t步的：当前参与者的历史推荐（包括正确推荐的问题和错误推荐的问题）和当前该参与者可以回答的候选问题列表。s'则包括 worker 在 t+1 步的历史推荐和候选问题列表。a则表示 agent 所选择的行为。r表示根据该行为，env 所给出的奖励值。具体代码如下：

```
    def update(self, transition_dict):
        states = transition_dict["states"]
        states_work_history = list(map(lambda x:x[0], states))
        states_action_list = list(map(lambda x:x[1], states))

        actions = transition_dict["actions"]
        action_idx = torch.tensor([action[0] for action in actions]).to(self.device).reshape(-1,1)
        rewards = transition_dict["rewards"]

        next_states = transition_dict["next_states"]
```

```
11.             next_states_work_history = list(map(lambda x:x[0], next_states))
12.             next_states_action_list = list(map(lambda x:x[1], next_states))
13.
14.             dones = transition_dict["dones"]
15.
16.             self.actor_optimizer.zero_grad()
17.             self.critic_optimizer.zero_grad()
18.
19.             worker_state = torch.concat([self.worker_history_to_embedding(worker_history) for
worker_history in states_work_history])
20.             rewards = torch.tensor(rewards, dtype=torch.float).view(-1, 1).to(self.device)
21.             next_worker_state = torch.concat([self.worker_history_to_embedding(worker_history) for
worker_history in next_states_work_history])
22.
23.         #       # 时序差分目标
24.             td_target = rewards + self.gamma * self.critic(next_worker_state)
25.             td_delta = td_target - self.critic(worker_state)   # 时序差分误差
26.             log_probs = torch.log(self.actor(worker_state, states_action_list).gather(1, action_idx))
27.             actor_loss = torch.mean(-log_probs * td_delta.detach())
28.             # 均方误差损失函数
29.             critic_loss = torch.mean(F.mse_loss(self.critic(worker_state), td_target.detach()))
30.             actor_loss.backward(retain_graph=True)   # 计算策略网络的梯度
31.             critic_loss.backward()   # 计算价值网络的梯度
32.             self.actor_optimizer.step()   # 更新策略网络的参数
33.             self.critic_optimizer.step()   # 更新价值网络的参数
34.
35.             return actor_loss.item(), critic_loss.item()
```

根据之前介绍的公式，需要计算$r+\gamma V_w(s_{t+1})$和$r_t+\gamma V(s_{t+1})-V(s_t)$，即代码中的 24-25 行。然后在计算当前状态可选择的各个行为所对应的概率，即代码中的 26 行。Actor 的 loss 为概率与时序差分残差的乘积再求平均，即代码中的 27 行。Critic 的 loss 为$V_w(s_t)$与$r+\gamma V_w(s_{t+1})$的均方误差。这里为了提高代码运行效率，选择在进行完一个 worker 的 episode 才进行更新，所以需要对 actor_loss 和 critic_loss 加上 torch.mean 进行一个求平均操作。

训练部分，与之前 reinforce 的过程相似，不在进行赘述。下面分析实验结果：

训练过程中，最大化参与者利益的 Critic 和 Actor 的损失函数变化曲线如图 16 和图 17 所示，最大化请求者利益的 Critic 和 Actor 的损失函数变化曲线如图 18 和图 19 所示。

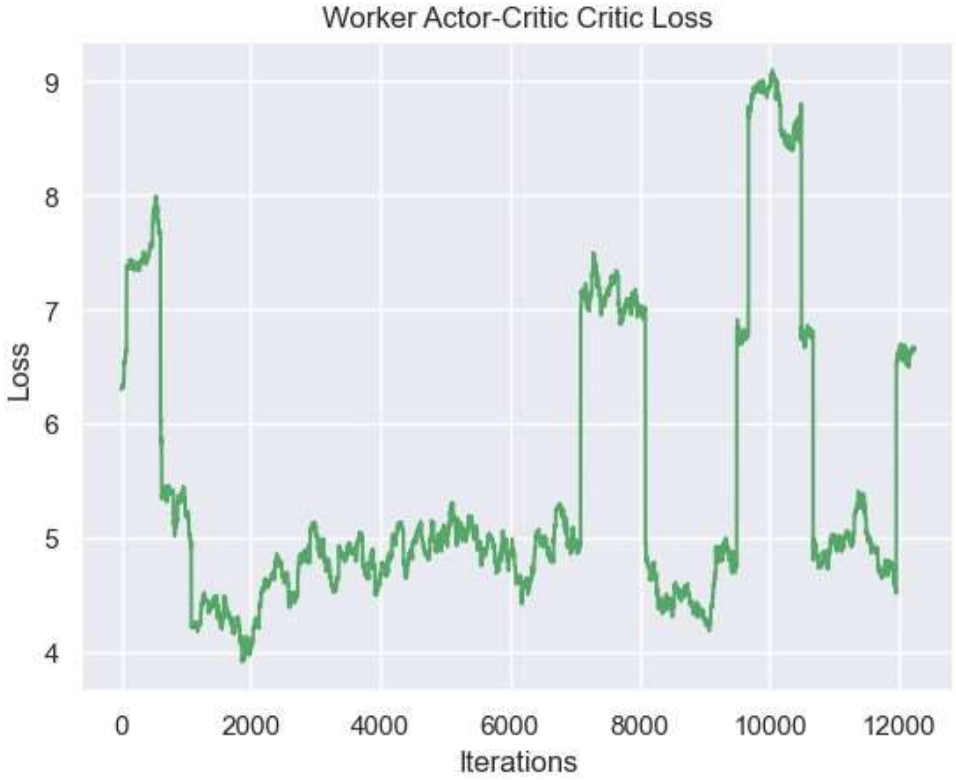

图 16-最大化参与者利益时 Critic 的损失函数变化曲线

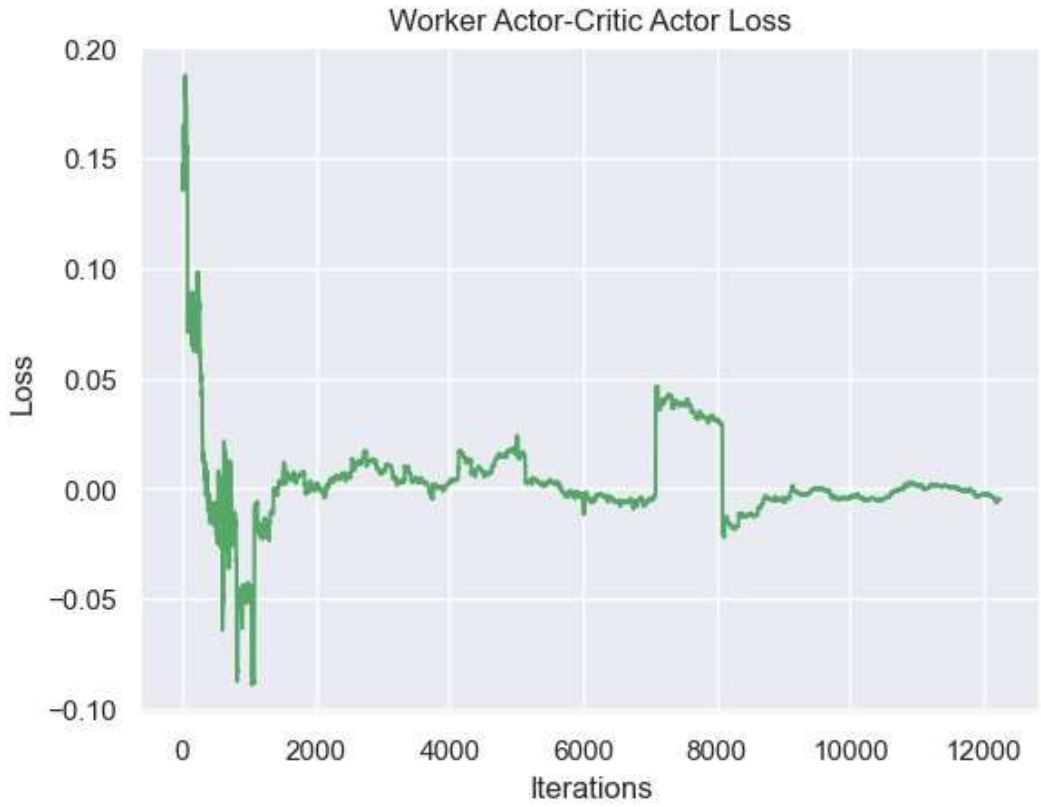

图 17-最大化参与者利益时 Actor 的损失函数变化曲线

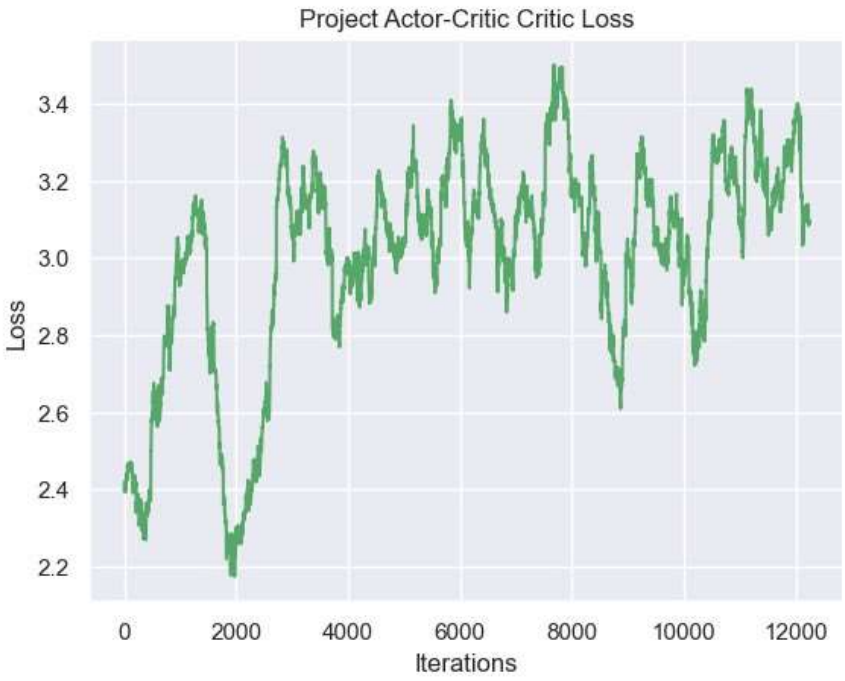

图 18-最大化请求者利益时 Critic 的损失函数变化曲线

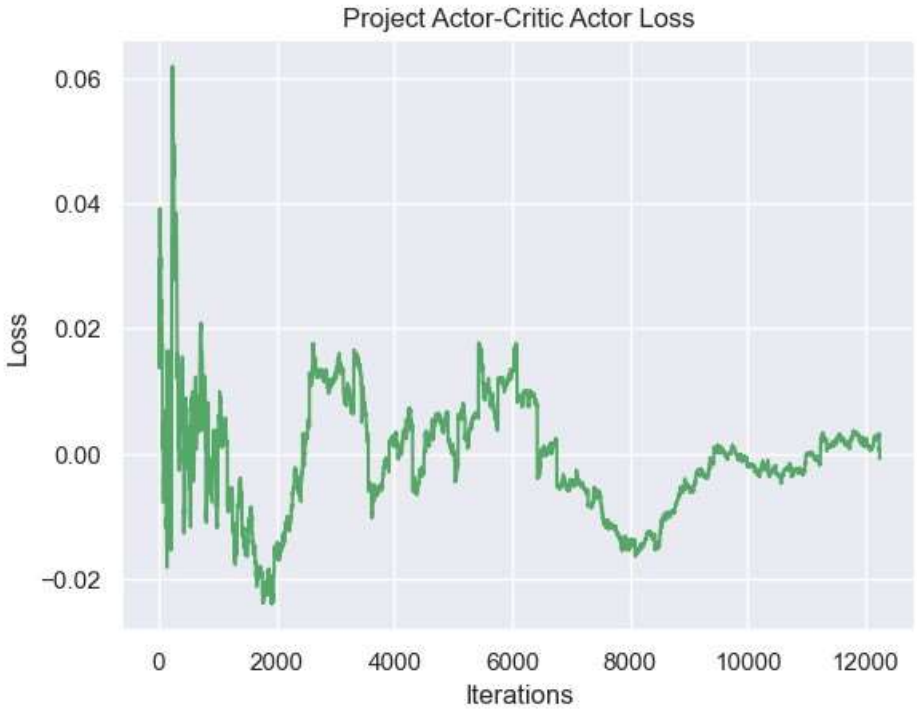

图 19-最大化请求者利益时 Actor 的损失函数变化曲线

从图 16 和图 18 来看，由于我们基于 Actor-Critic 算法设计的模型中 Critic 是用的基于值的方法，出现了和 6.1 节中 DQN 的 loss 相同的问题，即出现一定的波动且有上升的趋势。从图 17 和图 19 来看，Actor 的 loss 值逐渐下降，并趋于一个稳定的状态，表明模型逐渐收敛。

为了显示 Actor-Critic 的训练效果，我们依然使用随机策略作为对比，比较两个策略在训练过程中获得的奖励值（reward）的变化。图 20 和图 21 展示了设计的 PolicyNet 模型和 Random 模型随着迭代次数的增加，其奖励值的变化情况。

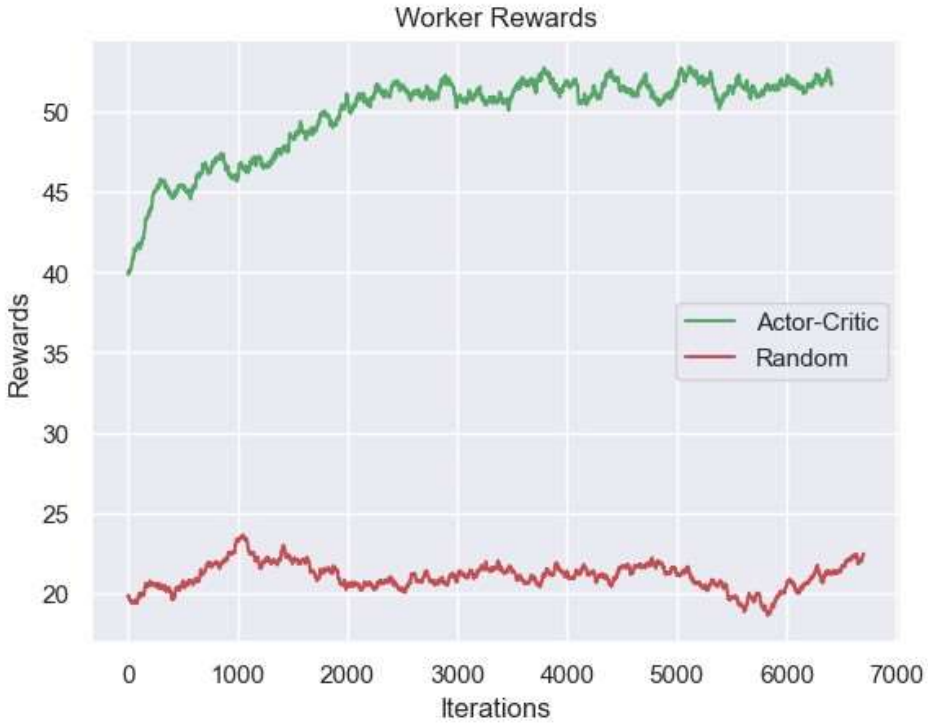

图 20-最大化参与者利益下 Actor-Critic 和 Random 模型奖励值的曲线变化

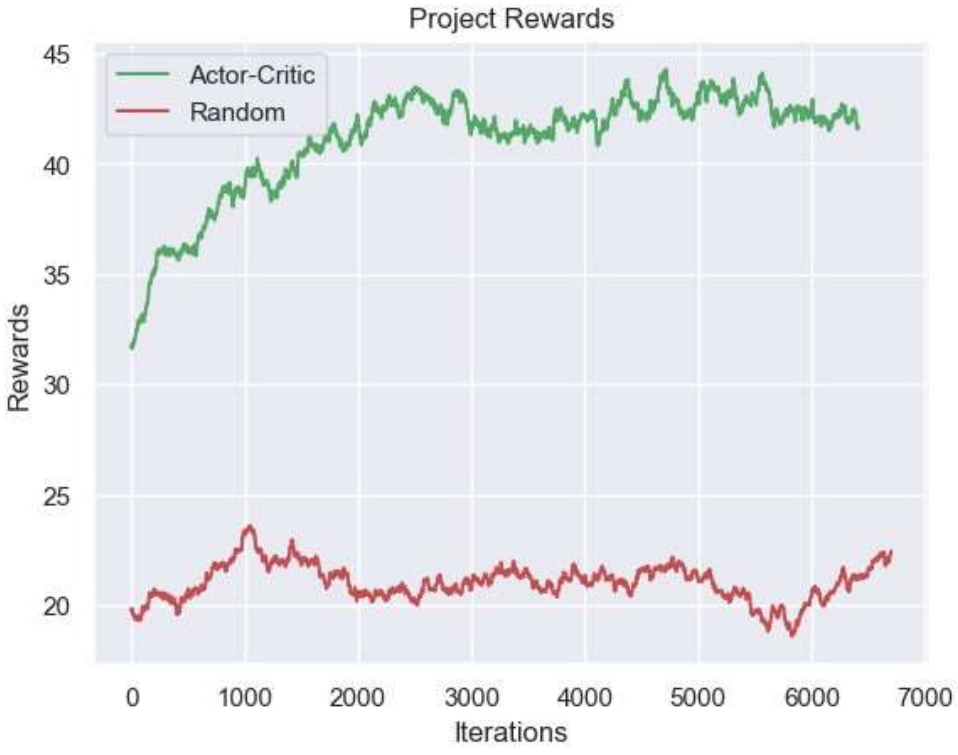

图 21-最大化请求者利益下 Actor-Critic 和 Random 模型奖励值的曲线变化

从图 20 和图 21 中可以看出，无论是在最大化参与者利益还是在最大化请求者利益的任务中，我们设计的 Actor-Critic 模型要明显优于随机策略。

6.4 总体比较分析

首先我们比较了 DQN、策略梯度、Actor-Critic 模型和随机策略在训练过程中获得的奖励值（reward）的变化。图 22 和图 23 展示了设计的三种模型和 Random 策略随着迭代次数的增加，其奖励值的变化情况。

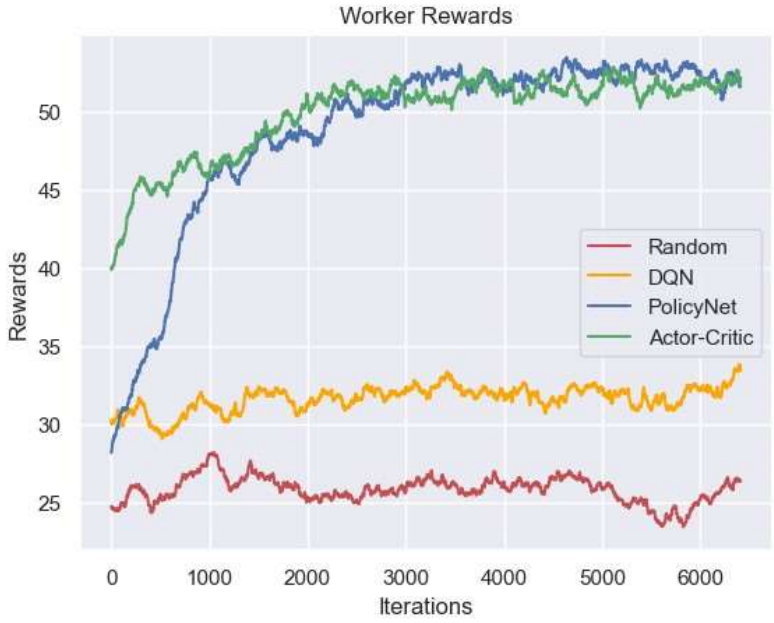

图 22-最大化参与者利益下三种模型和 Random 模型奖励值的曲线变化

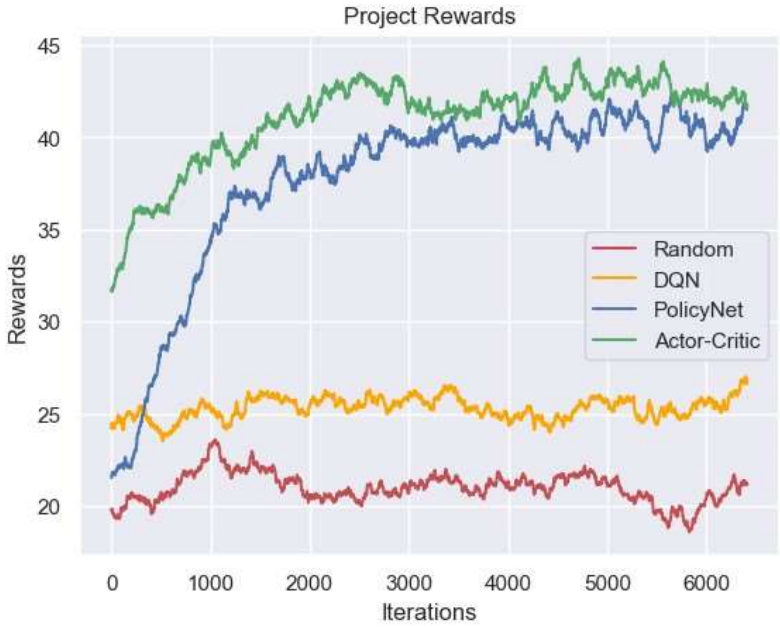

图 23-最大化请求者利益下三种模型和 Random 模型奖励值的曲线变化

从图 22 中可以看出，在最大化参与者利益的任务中，Actor-Critic 和 PolicyNet 训练最终达到的奖励值是最高的，要明显优于 DQN 模型和随机策略。但 Actor-Critic 在初始迭代时的奖励值更高，比 PolicyNet 能更快趋于稳定状态。DQN 虽然比设计的另外两个模型效果要差，但是也明显高于随机策略。

从图 23 中可以看出，在最大化请求者利益的任务中，四种策略在训练时的优劣排序是 Actor-Critic 最好，PolicyNet 其次，DQN 其次，随机策略最差。这两个图说明了我们设计的三种模型的有效性，也可以分析出基于 Actor-Critic 设计的模型最好，更适合众包推荐任务。

表 4 是在最大化参与者利益任务中，测试集上 Actor-Critic、PolicyNet、DQN 和随机策略 Accuracy 和 Rewards 的最终结果，表 5 是在最大化请求者利益任务中，测试集上 Actor-Critic、PolicyNet、DQN 和随机策略 Accuracy 和 Rewards 的最终结果。

表 4-最大化参与者利益时三种模型和随机策略测试上 Accuracy 和 Rewards 结果

Models	Accuracy	Rewards
Random	0.1173	2559
DQN	0.1743	**3850**
PolicyNet	0.1835	3841
Actor-Critic	**0.1889**	3751

表 5-最大化请求者利益时三种模型和随机策略测试上 Accuracy 和 Rewards 结果

Models	Accuracy	Rewards
Random	0.1056	2124
DQN	0.1544	**3137**
PolicyNet	0.1588	2799
Actor-Critic	**0.1599**	2644

从表 4 和表 5 中可以看出，设计的三种模型在最大化参与者利益和最大化请求者利益时 Accuracy 和 Rewards 两项指标都明显高于随机策略。Actor-Critic 在 Accuracy 指标上表现更好，DQN 在 Rewards 指标上表现更好。

七、总结

在本次深度学习和强化学习课程的大作业中，我们研究了强化学习在众包任务推荐中的应用。问题要求最大化参与者的利益和最大化请求者的利益，并使用课程中所学习的三类算法（Value-based method、Policy-based method 和 Actor-Critic method）来解决众包任务推荐问题。DQN 算法基于值函数的方法，通过

学习任务的值函数，使得代理能够选择具有最高价值的任务进行推荐。策略梯度算法基于策略的方法，通过直接学习策略函数，使得代理能够根据当前状态选择最优的行动。Actor-Critic 算法结合了值函数和策略函数的优势。Critic 函数用于估计值函数，Actor 函数用于生成策略。我们训练了 DQN 模型、PolicyNet 模型以及 Actor-Critic 模型，并通过实验评估了其在参与者和请求者利益上的表现。我们的代码放在 https://github.com/Skydzl/RL105。

参考：

[1] Gronauer S, Diepold K. Multi-agent deep reinforcement learning: a survey[J]. Artificial Intelligence Review, 2022, 55(2): 895-943.

[2] Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.