本周任务完成情况：

科研方面：

1、学习ppo算法,理解其主要原理为课题寻找方向，以下是ppo算法整体的训练框架代码：

```
class PPO:

    def __init__(self, policy_class, env, **hyperparameters):

        # PPO 初始化用于训练的超参数
        self._init_hyperparameters(hyperparameters)

        # 提取环境信息
        self.env = env
        self.obs_dim = env.observation_space.shape[0]
        self.act_dim = env.action_space.shape[0]

        # 初始化演员和评论家网络
        self.actor = policy_class(self.obs_dim, self.act_dim)

        self.critic = policy_class(self.obs_dim, 1)

        # 为演员和评论家初始化优化器
        self.actor_optim = Adam(self.actor.parameters(), lr=self.lr)
        self.critic_optim = Adam(self.critic.parameters(), lr=self.lr)

        # 初始化协方差矩阵，用于查询actor网络的action
        self.cov_var = torch.full(size=(self.act_dim,), fill_value=0.5)
        self.cov_mat = torch.diag(self.cov_var)

        # 这个记录器将帮助我们打印出每个迭代的摘要
        self.logger = {
            'delta_t': time.time_ns(),
            't_so_far': 0,          # 到目前为止的时间步数
            'i_so_far': 0,          # 到目前为止的迭代次数
            'batch_lens': [],       # 批次中的episodic长度
            'batch_rews': [],       # 批次中的rews回报
            'actor_losses': [],     # 当前迭代中演员网络的损失
        }

    def learn(self, total_timesteps):

        print(f"Learning... Running {self.max_timesteps_per_episode} timesteps per
episode, ", end='')
        print(f"{self.timesteps_per_batch} timesteps per batch for a total of
{total_timesteps} timesteps")
        t_so_far = 0 # 到目前为止仿真的时间步数
        i_so_far = 0 # 到目前为止，已运行的迭代次数
        while t_so_far < total_timesteps:
```

```
            # 收集批量实验数据
            batch_obs, batch_acts, batch_log_probs, batch_rtgs, batch_lens =
self.rollout()

            # 计算收集这一批数据的时间步数
            t_so_far += np.sum(batch_lens)

            # 增加迭代次数
            i_so_far += 1

            # 记录到目前为止的时间步数和到目前为止的迭代次数
            self.logger['t_so_far'] = t_so_far
            self.logger['i_so_far'] = i_so_far

            # 计算第k次迭代的advantage
            V, _ = self.evaluate(batch_obs, batch_acts)
            A_k = batch_rtgs - V.detach()


            # 将优势归一化 在理论上不是必须的，但在实践中，它减少了我们优势的方差，使收敛更加稳定
和快速。
            # 添加这个是因为在没有这个的情况下，解决一些环境的问题太不稳定了。
            A_k = (A_k - A_k.mean()) / (A_k.std() + 1e-10)

            # 在其中更新我们的网络。
            for _ in range(self.n_updates_per_iteration):

                V, curr_log_probs = self.evaluate(batch_obs, batch_acts)

                # 重要性采样的权重
                ratios = torch.exp(curr_log_probs - batch_log_probs)

                surr1 = ratios * A_k
                surr2 = torch.clamp(ratios, 1 - self.clip, 1 + self.clip) * A_k

                # 计算两个网络的损失。
                actor_loss = (-torch.min(surr1, surr2)).mean()
                critic_loss = nn.MSELoss()(V, batch_rtgs)

                # 计算梯度并对actor网络进行反向传播
                # 梯度清零
                self.actor_optim.zero_grad()
                # 反向传播，产生梯度
                actor_loss.backward(retain_graph=True)
                # 通过梯度下降进行优化
                self.actor_optim.step()

                # 计算梯度并对critic网络进行反向传播
                self.critic_optim.zero_grad()
                critic_loss.backward()
                self.critic_optim.step()
```

```
                self.logger['actor_losses'].append(actor_loss.detach())

            self._log_summary()

            if i_so_far % self.save_freq == 0:
                torch.save(self.actor.state_dict(), './ppo_actor.pth')
                torch.save(self.critic.state_dict(), './ppo_critic.pth')

    def rollout(self):
        """
            这就是我们从实验中收集一批数据的地方。由于这是一个on-policy的算法，我们需要在每次迭代
行为者/批评者网络时收集一批新的数据。
        """
        batch_obs = []
        batch_acts = []
        batch_log_probs = []
        batch_rews = []
        batch_rtgs = []
        batch_lens = []

        # 一回合的数据。追踪每一回合的奖励，在回合结束的时候会被清空，开始新的回合。
        ep_rews = []

        # 追踪到目前为止这批程序我们已经运行了多少个时间段
        t = 0

        # 继续实验，直到我们每批运行超过或等于指定的时间步数
        while t < self.timesteps_per_batch:
            ep_rews = []  每回合收集的奖励

            # 重置环境
            obs = self.env.reset()
            done = False

            # 运行一个回合的最大时间为max_timesteps_per_episode的时间步数
            for ep_t in range(self.max_timesteps_per_episode):

                if self.render and (self.logger['i_so_far'] % self.render_every_i ==
0) and len(batch_lens) == 0:
                    self.env.render()

                # 递增时间步数，到目前为止已经运行了这批程序
                t += 1

                #  追踪本批中的观察结果
                batch_obs.append(obs)

                # 计算action，并在env中执行一次step。
                # 注意，rew是奖励的简称。
                action, log_prob = self.get_action(obs)
                obs, rew, done, _ = self.env.step(action)
```

```
                # 追踪最近的奖励、action和action的对数概率
                ep_rews.append(rew)
                batch_acts.append(action)
                batch_log_probs.append(log_prob)

                if done:
                    break

            # 追踪本回合的长度和奖励
            batch_lens.append(ep_t + 1)
            batch_rews.append(ep_rews)

        # 将数据重塑为函数描述中指定形状的张量，然后返回
        batch_obs = torch.tensor(batch_obs, dtype=torch.float)
        batch_acts = torch.tensor(batch_acts, dtype=torch.float)
        batch_log_probs = torch.tensor(batch_log_probs, dtype=torch.float)
        batch_rtgs = self.compute_rtgs(batch_rews)


        # 在这批中记录回合的回报和回合的长度。
        self.logger['batch_rews'] = batch_rews
        self.logger['batch_lens'] = batch_lens

        return batch_obs, batch_acts, batch_log_probs, batch_rtgs, batch_lens

    def compute_rtgs(self, batch_rews):

        batch_rtgs = []

        # 遍历每一回合，一个回合有一批奖励
        for ep_rews in reversed(batch_rews):
            # 到目前为止的折扣奖励
            discounted_reward = 0

            # 遍历这一回合的所有奖励。我们向后退，以便更顺利地计算每一个折现的回报
            for rew in reversed(ep_rews):

                discounted_reward = rew + discounted_reward * self.gamma
                batch_rtgs.insert(0, discounted_reward)

        # 将每个回合的折扣奖励的数据转换成张量
        batch_rtgs = torch.tensor(batch_rtgs, dtype=torch.float)

        return batch_rtgs

    def get_action(self, obs):

        mean = self.actor(obs)

        # 用上述协方差矩阵中的平均行动和标准差创建一个分布。
        dist = MultivariateNormal(mean, self.cov_mat)
        action = dist.sample()
```

```
        log_prob = dist.log_prob(action)

        return action.detach().numpy(), log_prob.detach()

    def evaluate(self, batch_obs, batch_acts):
        """
            估算每个观察值，以及最近一批actor网络迭代中的每个action的对数prob。
        """

        # 为每个batch_obs查询critic网络的V值。V的形状应与batch_rtgs相同。
        V = self.critic(batch_obs).squeeze()

        # 使用最近的actor网络计算批量action的对数概率。
        mean = self.actor(batch_obs)
        dist = MultivariateNormal(mean, self.cov_mat)
        log_probs = dist.log_prob(batch_acts)

        # 返回批次中每个观察值的值向量V和批次中每个动作的对数概率log_probs
        return V, log_probs
```

2、在主流的简单的单智能提环境下完成ppo算法的部署，进行训练，并取得预期训练结果。
项目方面：
了解了天河超算平台，并配置好环境，在超算平台下进行模型训练。

下周任务：
完成10月份的汇报，系统梳理多智能体强化学习的相关知识。