

强化学习

贪婪算法

算法常规流程

Base_agent

基于价值的方法

基于随机策略的方法

基于确定策略的方法DPG、DDPG

- 特性
- 1、在线学习
 - 2、基于Q值的算法
 - 3、不直接更新策略

- replay_buffer
- 1、收集数据
 - 2、采样
 - 3、更新网络

- replay_buffer
- 使用namedtuple、deque构建经验池
 - Add_experience
 - sample_experiences — 选择数据，根据参数，确定是否格式化数据
 - pick_experiences — 从经验池中随机选择数据
 - separate_out_data_types — 格式化数据
 - _len_

- 1、单个model
- 2、模型计算Q_value
- 3、包含历史model的数据

reset_game

- step
- 1、选择动作
 - 2、执行动作
 - 3、是否满足学习条件
 - 4、保存经验
 - 5、是否满足结束条件，否则重复1~4

pick_action

learn

- *calculate_loss
- 1、通过模型计算当前的current_q_value
 - 2、通过模型计算next_state_q_value
 - 3、Q_expected=rewards+gamma*next_state_q_value
 - 4、计算current_q_value和Q_expected之间的loss

calculate_q_targets

calculate_q_values_for_next_states

calculate_q_values_for_current_states

calculate_expected_q_values

locally_save_policy

time_for_q_network_to_learn

right_amount_of_steps_taken — 是否达到学习条件，执行learn

sample_experiences — 获取数据

- DDQN
- 两个网络，特定条件同步参数
 - calculate_q_values_for_next_states — 基于Q_target_network计算Q_values

DDQN_PEP — 两个网络，优先经验回放

- 1、模型计算Action概率
- 2、单个model
- 3、每一幕才进行一次学习

reset_game

- step
- 1、获取、保存动作及log_probabilities
 - 2、执行动作
 - 3、保存reward
 - 4、是否达到学习条件，否则执行1~3
- 每完成一幕才能进行学习

- pick_and_conduct_action_and_save_log_probabilities
- 1、获取动作及log_probabilities
 - 2、保存动作及log_probabilities
 - 3、执行动作

pick_action_and_get_log_probabilities — 获取动作及log_probabilities

store_log_probabilities — 保存log_probabilities

store_action

store_reward

- actor_learn
- 1、计算一幕的rewards
 - 2、计算一幕的loss
 - 3、loss反向传播（优化模型）

calculate_episode_discounted_reward — 计算每一幕的rewards
gamma ** np.arange(len(self.episode_rewards))
rewards = np.dot(discounts, self.episode_rewards)

calculate_policy_loss_on_episode — 计算一个batch_size的总损失（一幕）

time_to_learn — 设定学习条件：done is true

- 1、模型计算Action概率
- 2、两个model
- 3、支持离散和连续动作

- 算法思路
- 1、两个模型，策略模型、价值模型
 - 2、通过策略模型获取动作
 - 3、

核心代码：
td_target = rewards + self.gamma * self.critic(next_states) * (1 - dones)
td_delta = td_target - self.critic(states)
advantage = rl_utils.compute_advantage(self.gamma, self.lmbda, td_delta.cpu()).to(self.device)
old_log_probs = torch.log(self.actor(states).gather(1, actions)).detach()

for _ in range(self.epochs):
log_probs = torch.log(self.actor(states).gather(1, actions))
ratio = torch.exp(log_probs - old_log_probs)
surr1 = ratio * advantage
surr2 = torch.clamp(ratio, 1 - self.eps, 1 + self.eps) * advantage # 截断
actor_loss = torch.mean(torch.min(surr1, surr2)) # PPO损失函数
critic_loss = torch.mean(F.mse_loss(self.critic(states), td_target.detach()))

calculate_policy_output_size — 动作类型：离散动作、连续动作

- step
- 1、更新探索率
 - 2、多进程获取多个链路（幕）数据
 - 3、更新模型
 - 4、更新优化函数（学习率等）
 - 5、同步模型

policy_learn

calculate_all_discounted_returns — 计算所有幕的returns

calculate_all_ratio_of_policy_probabilities — 1、计算新旧模型获取各个动作的log_probability
2、计算log_probability比率

calculate_log_probability_of_actions — 计算动作的log_probability

calculate_loss — 1、将数据通过torch.clamp限制在指定范围
2、将returns限定

```
def calculate_advantage(gamma, lmbda, td_delta):  
    td_delta = td_delta.detach().numpy()  
    advantage_list = []  
    advantage = 0.0  
    for delta in td_delta[::-1]:  
        advantage = gamma * lmbda * advantage + delta  
        advantage_list.append(advantage)  
    advantage_list.reverse()  
    return torch.tensor(advantage_list, dtype=torch.float)
```

clamp_probability_ratio — 数据截断到指定范围

take_policy_new_optimisation_step

equalise_policies — 同步模型参数

save_result

- AC
- 1、2个网络，价值网络（Critic_评委），策略网络（Actor_演员）
 - 2、价值网络计算loss: loss_fuc(critic(states), rewards + critic(next_states))
 - 3、策略网络计算loss:
td_delta = td_target - self.critic(states) # 时序差分误差
log_probs = torch.log(self.actor(states).gather(1, actions))
actor_loss = torch.mean(-log_probs * td_delta.detach())

- A3C
- run_n_episodes
 - print_results
 - update_shared_model

- Actor_Critic_Worker
- set_seeds
 - run
 - calculate_new_exploration
 - reset_game_for_worker
 - pick_action_and_get_critic_values
 - calculate_log_action_probability
 - calculate_total_loss
 - calculate_discounted_returns
 - normalise_discounted_returns
 - calculate_critic_loss_and_advantages
 - calculate_actor_loss
 - put_gradients_in_queue