

PPO × Family 第四讲习题



本讲习题**共包含两部分：分别是算法理论题和代码实践题**。同学们可以**选择其一项**完成并提交。（**当然也欢迎大家将两部分全部完成**，这样能够加深对课程的理解）

- 算法理论题提交方式：

发送邮件至 opendilab@pjlab.org.cn

请同学们严格按照下方格式命名邮箱主题/标题：

【PPO × Family】 + 学生名 + vol.4 (第几节课) + 作业提交日期

示例：【PPO × Family】 + 喵小DI + vol.4 +20230223

- 代码实践题提交方式：

PPO × Family 官方GitHub 上发起 Pull Request

- 地址：[PP0xFamily/tree/main/chapter4_reward/hw_submission](https://github.com/PP0xFamily/tree/main/chapter4_reward/hw_submission)

- PR示例：<https://github.com/opendilab/PP0xFamily/pull/5>

- 命名规范：

hw_submission(学生名称): add hw4 (第几节课) +作业提交日期

示例：hw_submission(nyz): add hw3_20230104

提交**截止时间为 2023.03.16 23:59 (GMT +8)**，逾期作业将不会计入证书考量。

如果其他问题请添加官方课程小助手微信（vx: OpenDILab），备注「课程」，小助手将邀请您加入官方课程微信交流群；或发送邮件至 opendilab@pjlab.org.cn

算法理论题

题目1 (distributional RL)

价值函数是强化学习算法中的核心概念。某一个策略下计算获得的一个状态的价值或一个动作状态对的价值，衡量了当前策略所能获取的最大累计回报的相对大小，因而对策略的价值函数进行优化，是学习获得更优策略的一种方式。优化价值函数的方法是通过使用贝尔曼最优方程 (Bellman Optimality Equation) 来进行价值函数的更新：

$$Q^*(s, a) = r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')$$

上式中， $Q^*(s, a)$ 为理论中的最优策略下，该状态 s 和动作 a 对应的价值函数， $r(s, a)$ 为这个状态动作对带来的回报。由于在实际操作中，我们刚开始只有某个一般策略 π 下的价值函数 $Q(s, a)$ ，即在该一般策略下，收集到的轨迹下游所有累计回报的期望，那么为了求得最优价值函数 $Q^*(s, a)$ ，我们需要使用算子来逐步更新我们对价值函数的建模估计。贝尔曼最优算子 \mathcal{T} 的形式为：

$$\mathcal{T}Q(s, a) = \mathbb{E}[r(s, a)] + \gamma \mathbb{E}_\pi[\max_{a' \in \mathcal{A}} Q(s', a')]$$

然而，由于环境中随机性的广泛存在（比如随机时刻风速的大小，会影响马拉松运动员的相对完成时间，而与运动员的比赛策略与个人能力无关），或是某些独立于动作与状态的随机的意外事件的发生（比如突然出现的自然灾害，会影响农作物的收成的相对大小，而与农业耕种的一整年的策略无关），这意味着每一个状态动作对的回报， $r(s, a)$ ，可能是一个随机数值。因此，在某个策略 π 下所生成的轨迹所有累计回报 $Z(s, a)$ ，也会成为一个概率分布：

$$Z(s_t, a_t) = \sum_{i=0}^{end} \gamma^i r(s_{t+i}, a_{t+i})$$

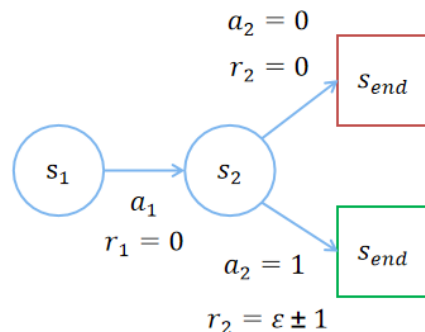
上式中，下一时刻的状态满足状态转移概率的分布，即 $s_{t+i} \sim P(\cdot | s_{t+i-1}, a_{t+i-1})$ ，下一时刻的动作遵循当前的策略下动作的概率分布， $a_{t+i} \sim \pi(\cdot | s_{t+i-1})$ ，累计回报将求和至所有的终止状态，记所有终止状态对应时刻的 i 为 end 。

所以我们的强化学习算法也就相应地需要转变为求得这个概率分布上的最优策略。在使用该优化算子进行策略提升时，其实质形式变为：

$$\mathcal{T}Z(s, a) = r(s, a) + \gamma Z(s', a')$$

其中， $s' \sim p(\cdot | s, a)$ ， $a' \sim \pi(\cdot | s')$

具体来说当价值函数成为一个分布 (distribution)，相比于一个数 (scalar)，在更新前后会发生什么潜在的影响，我们可以使用下面一个例子来形象化说明这一点：



对于一个由 $s_1 \rightarrow s_2 \rightarrow s_{\text{end}}$ 构成的MDP。在状态 s_1 时仅有 a_1 一个动作可以选择，奖励为 $r_1 = 0$ 。在状态 s_2 时有 $a_2 = 0$ 和 $a_2 = 1$ 两个动作可以选择，奖励分别为 $r_2 = 0$ （为确定性的数值）和 $r_2 = \epsilon \pm 1$ （为随机数，且事件发生的概率均等），其中 $\epsilon \geq 0$ 。

1、请计算理论上的最优动作价值函数的数值 $Q(s_1, a_1)$ ， $Q(s_2, a_2 = 0)$ ， $Q(s_2, a_2 = 1)$ 与最优动作价值函数的分布 $Z(s_1, a_1)$ ， $Z(s_2, a_2 = 0)$ ， $Z(s_2, a_2 = 1)$ ，为了简化计算，令 $\gamma = 1$

（示例：比如 $Q(s_2, a_2 = 1) = \mathbb{E}(r(s_2, a_2)) + 0 = \epsilon$ ， $Z(s_2, a_2 = 1) = r(s_2, a_2) + 0 = \epsilon \pm 1$ ）

2、假如当前策略的动作价值函数分布为 $Z(s_1, a_1) = \epsilon \pm 1$ ， $Z(s_2, a_2 = 0) = 0$ ， $Z(s_2, a_2 = 1) = -\epsilon \pm 1$ 。

i) 请计算使用贝尔曼最优算子后，目标动作价值函数的分布， $\mathcal{T}Z(s_1, a_1)$ ， $\mathcal{T}Z(s_2, a_2 = 0)$ ， $\mathcal{T}Z(s_2, a_2 = 1)$ 。

（示例：比如 $\mathcal{T}Z(s_2, a_2 = 1) = r(s_2, a_2 = 1) + 0 = \epsilon \pm 1$ ）

ii) 衡量两个概率分布之间的距离的测度方法有很多，其中之一为Wasserstein Metric [1]。假如标记 p 阶Wasserstein Metric 为 W_p ，其表达式如下：

$$W_p(\mu, \nu) = \left(\inf_{\gamma \in \Gamma(\mu, \nu)} \mathbb{E}_{(x, y) \sim \gamma} d(x, y)^p \right)^{\frac{1}{p}}$$

请计算使用贝尔曼最优算子前后，当前策略的动作价值函数与最优策略的动作价值函数的 1 阶Wasserstein Metric。比较并讨论两者的差异，然后尝试分析这种差异的影响。

题目2（更高级的探索算法：Upper Confidence Bound）

电影《流浪地球2》中，移山计划的验证项目建设过程中，月球太空设备仓库新到了一批来自两家制造厂商的同一款处理器芯片，不过由于两家厂商的工艺水平不同，在遭受一次太阳风暴高能粒子袭击后，两种芯片有少量被损坏，且比例不太相同，但是难以检测。有一天机器狗笨笨接到新任务，需要安装一定数量芯片到对应数量的工作机器上，希望那些机器可以被顺利启动并工作的比例越高越好。笨笨一次只能选择来自一个厂商的芯片（可以记为决策动作 $a = 0$ 与 $a = 1$ ），并安装到一台机器上，并尝试启动它（启动成功时，可以记为奖励 $r = 1$ ，启动失败记为奖励 $r = 0$ ）。最早设计决策

算法的工程师为了赶DDL交差，只给笨笨写了 $\epsilon - \text{greedy}$ 算法，现在我们需要一种更高级的探索算法 UCB (Upper Confidence Bound) [2]，帮助笨笨更好地完成这个工作。

请参考下文中的提示，证明机械狗笨笨需要采用的UCB算法的数学形式为：

$$a_t = \arg \max_a [Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}}]$$

其中 t 为第 t 次安装测试。 $N_t(a)$ 为累计至第 $t - 1$ 次安装时，选择 a 厂商的次数。 $Q_t(a)$ 为累计至第 $t - 1$ 次安装时的平均动作价值。 c 为平衡探索效益和利用效益之间的系数。

提示：

i)

一般来说，最优算法的设计目标是为了最大化累计的回报，即：

$$\max \sum_t r_t(s_t, a)$$

我们可以把使用动作 a 的平均奖励记为 $Q(a)$ ：

$$Q(a) = \frac{1}{N} \sum_{i=1}^N r_i(s_i, a)$$

但是由于对环境信息的未知，智能体需要做探索性的尝试来获取更多信息，并利用已有信息决策，取得更优的奖励。不过由于客观条件的限制，在有限的时空内，探索和利用往往是互斥的，如何平衡多少程度的探索和多少程度的利用，会最终影响决策算法的最终表现。因此，如何用数学方法平衡探索 (Exploration) 和利用 (Exploitation)，是各类算法的核心区别。

对于常见的 $\epsilon - \text{greedy}$ 算法，它平衡探索和利用的方式是使用一个小概率 ϵ 做随机探索，再用 $1 - \epsilon$ 的概率来做最大化当前效益的利用。但是很明显，这种简单粗暴的方式存在问题，它忽视了探索过程带来的信息，我们可以设计算法利用这些信息从而逐渐降低不确定性，但 $\epsilon - \text{greedy}$ 算法因为设计上过于简单，并不会有效地利用这些信息。

而UCB方法使用了“**保持最乐观态度去面对未来的不确定性**”的原则，来平衡探索和利用这两者。

也就是说，在 UCB 策略中，对于任意 t 时刻，智能体会对某个动作的平均奖励做一个比较乐观的估计，可以记为 $\tilde{Q}_t(a)$ ，真实的平均奖励记为 $Q_t(a)$ ，这个乐观的估计是真实值的一个上界：

$$\tilde{Q}_t(a) \geq Q_t(a)$$

UCB方法的原则，是要去尽可能地信任这个上界，将它作为选择动作的依据，即选择所有动作中，乐观估计的上界最大的动作，作为本次决策的结果：

$$a_t = \arg \max_a [\tilde{Q}_t(a)]$$

需要注意的是，在每个第 t 时刻，这意味着已经具备前 $t - 1$ 时刻所有交互的信息。所以可以根据这些信息，来估计这个乐观的上界的范围。

所以原问题等价于，论述为什么公式 $\tilde{Q}_t(a) = Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}$ 是一种乐观形式的上界。

ii)

霍夫丁不等式 (Hoeffding's inequality):

如果 Z 为一系列独立同分布且有界的随机数, $Z_i \in [a, b]$, $-\infty < a \leq b < +\infty$, 那么对于所有的非负实数, $\delta \geq 0$, 都有如下不等式成立:

$$P\left(\frac{1}{n}\left[\sum_{i=1}^n (Z_i - \mathbb{E}[Z_i])\right] \geq \delta\right) \leq \exp\left(-\frac{2n\delta^2}{(b-a)^2}\right)$$
$$P\left(\frac{1}{n}\left[\sum_{i=1}^n (Z_i - \mathbb{E}[Z_i])\right] \leq -\delta\right) \leq \exp\left(-\frac{2n\delta^2}{(b-a)^2}\right)$$

代码实践题

题目1（奖励模型的训练实践）

在许多现实生活中决策任务里，往往许多问题都是一些稀疏奖励的问题，在解决问题的过程中，长期无法获得激励和回报，这会给强化学习算法的寻优探索和顺利训练带来一定的挑战。基于好奇心探索的内在奖励模型，是帮助强化学习算法是解决这个问题的一种思路。

在论文《Exploration by Random Network Distillation》中，研究者们提出了一种 random network distillation (RND) 方法来辅助探索，激励智能体访问其不熟悉（更新颖）的状态，并期望从这种内在奖励引导的探索中获得更大的潜在收益。

在本题目中，我们将尝试一个简化后的 RND 奖励模型训练实践任务。具体来说，我们收集了Minigrid [5] 环境（具体为“MiniGrid-Empty-8x8-v0”）强化学习训练过程中产生的部分数据，现在请你**尝试使用不同超参数设置**的 RND 预测网络，用于回归一个固定大小的目标网络的输出，并将预测网络和目标网络的输出的预测差值作为内在探索奖励的大小，观察 Tensorboard 中相关的数据记录结果，分析探索奖励的数值在预测网络各个训练阶段的变化，评估其奖励数值的大小和范围与模型本身的**欠拟合或过拟合**等因素之间的关联。

具体的示例代码如下，只需选择其中不同的超参数配置进行训练，并观察和分析相应表现：

数据下载链接为（包含训练集和验证集）：[传送门](#)

下方的代码也可以从官网链接下载：[传送门](#)

```
1 # pip install minigrid
2 from typing import Union, Tuple, Dict, List, Optional
3 from multiprocessing import Process
4 import multiprocessing as mp
5 import random
6 import numpy as np
7 import torch
8 import torch.nn as nn
9 import torch.nn.functional as F
10 import torch.optim as optim
11 import minigrid
12 import gymnasium as gym
13 from torch.optim.lr_scheduler import ExponentialLR, MultiStepLR
14 from tensorboardX import SummaryWriter
15 from minigrid.wrappers import FlatObsWrapper
16
17 random.seed(0)
18 np.random.seed(0)
19 torch.manual_seed(0)
20 if torch.cuda.is_available():
21     device = torch.device("cuda:0")
22 else:
23     device = torch.device("cpu")
24
25 train_config = dict(
26     train_iter=1024,
27     train_data_count=128,
28     test_data_count=4096,
29 )
30
31 little_RND_net_config = dict(
32     exp_name="little_rnd_network",
33     observation_shape=2835,
34     hidden_size_list=[32, 16],
35     learning_rate=1e-3,
36     batch_size=64,
37     update_per_collect=100,
38     obs_norm=True,
39     obs_norm_clamp_min=-1,
40     obs_norm_clamp_max=1,
41     reward_mse_ratio=1e5,
42 )
43
44 small_RND_net_config = dict(
45     exp_name="small_rnd_network",
46     observation_shape=2835,
47     hidden_size_list=[64, 64],
```

```
48     learning_rate=1e-3,
49     batch_size=64,
50     update_per_collect=100,
51     obs_norm=True,
52     obs_norm_clamp_min=-1,
53     obs_norm_clamp_max=1,
54     reward_mse_ratio=1e5,
55 )
56
57 standard_RND_net_config = dict(
58     exp_name="standard_rnd_network",
59     observation_shape=2835,
60     hidden_size_list=[128, 64],
61     learning_rate=1e-3,
62     batch_size=64,
63     update_per_collect=100,
64     obs_norm=True,
65     obs_norm_clamp_min=-1,
66     obs_norm_clamp_max=1,
67     reward_mse_ratio=1e5,
68 )
69
70 large_RND_net_config = dict(
71     exp_name="large_RND_network",
72     observation_shape=2835,
73     hidden_size_list=[256, 256],
74     learning_rate=1e-3,
75     batch_size=64,
76     update_per_collect=100,
77     obs_norm=True,
78     obs_norm_clamp_min=-1,
79     obs_norm_clamp_max=1,
80     reward_mse_ratio=1e5,
81 )
82
83 very_large_RND_net_config = dict(
84     exp_name="very_large_RND_network",
85     observation_shape=2835,
86     hidden_size_list=[512, 512],
87     learning_rate=1e-3,
88     batch_size=64,
89     update_per_collect=100,
90     obs_norm=True,
91     obs_norm_clamp_min=-1,
92     obs_norm_clamp_max=1,
93     reward_mse_ratio=1e5,
94 )
```

```

95
96 class FCEncoder(nn.Module):
97     def __init__(
98         self,
99         obs_shape: int,
100         hidden_size_list,
101         activation: Optional[nn.Module] = nn.ReLU(),
102     ) -> None:
103         super(FCEncoder, self).__init__()
104         self.obs_shape = obs_shape
105         self.act = activation
106         self.init = nn.Linear(obs_shape, hidden_size_list[0])
107
108         layers = []
109         for i in range(len(hidden_size_list) - 1):
110             layers.append(nn.Linear(hidden_size_list[i], hidden_size_list[i + 1])
111             layers.append(self.act)
112         self.main = nn.Sequential(*layers)
113
114     def forward(self, x: torch.Tensor) -> torch.Tensor:
115         x = self.act(self.init(x))
116         x = self.main(x)
117         return x
118
119 class RndNetwork(nn.Module):
120     def __init__(self, obs_shape: Union[int, list], hidden_size_list: list) -> N
121         super(RndNetwork, self).__init__()
122         self.target = FCEncoder(obs_shape, hidden_size_list)
123         self.predictor = FCEncoder(obs_shape, hidden_size_list)
124
125         for param in self.target.parameters():
126             param.requires_grad = False
127
128     def forward(self, obs: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
129         predict_feature = self.predictor(obs)
130         with torch.no_grad():
131             target_feature = self.target(obs)
132         return predict_feature, target_feature
133
134 class RunningMeanStd(object):
135     def __init__(self, epsilon=1e-4, shape=(), device=torch.device('cpu')):
136         self._epsilon = epsilon
137         self._shape = shape
138         self._device = device
139         self.reset()
140
141     def update(self, x):

```



```

142     batch_mean = np.mean(x, axis=0)
143     batch_var = np.var(x, axis=0)
144     batch_count = x.shape[0]
145
146     new_count = batch_count + self._count
147     mean_delta = batch_mean - self._mean
148     new_mean = self._mean + mean_delta * batch_count / new_count
149     # this method for calculating new variable might be numerically unstable
150     m_a = self._var * self._count
151     m_b = batch_var * batch_count
152     m2 = m_a + m_b + np.square(mean_delta) * self._count * batch_count / new
153     new_var = m2 / new_count
154     self._mean = new_mean
155     self._var = new_var
156     self._count = new_count
157
158     def reset(self):
159         if len(self._shape) > 0:
160             self._mean = np.zeros(self._shape, 'float32')
161             self._var = np.ones(self._shape, 'float32')
162         else:
163             self._mean, self._var = 0., 1.
164         self._count = self._epsilon
165
166     @property
167     def mean(self) -> np.ndarray:
168         if np.isscalar(self._mean):
169             return self._mean
170         else:
171             return torch.FloatTensor(self._mean).to(self._device)
172
173     @property
174     def std(self) -> np.ndarray:
175         std = np.sqrt(self._var + 1e-8)
176         if np.isscalar(std):
177             return std
178         else:
179             return torch.FloatTensor(std).to(self._device)
180
181     class RndRewardModel():
182
183         def __init__(self, config) -> None: # noqa
184             super(RndRewardModel, self).__init__()
185             self.cfg = config
186
187             self.tb_logger = SummaryWriter(config["exp_name"])
188             self.reward_model = RndNetwork(

```

```

189         obs_shape=config["observation_shape"], hidden_size_list=config["hidd
190     ).to(device)
191
192     self.opt = optim.Adam(self.reward_model.predictor.parameters(), config["
193     self.scheduler = ExponentialLR(self.opt, gamma=0.997)
194
195     self.estimate_cnt_rnd = 0
196     if self.cfg["obs_norm"]:
197         self._running_mean_std_rnd_obs = RunningMeanStd(epsilon=1e-4, device
198
199     def __del__(self):
200         self.tb_logger.flush()
201         self.tb_logger.close()
202
203     def train(self, data) -> None:
204         for _ in range(self.cfg["update_per_collect"]):
205             train_data: list = random.sample(data, self.cfg["batch_size"])
206             train_data: torch.Tensor = torch.stack(train_data).to(device)
207             if self.cfg["obs_norm"]:
208                 # Note: observation normalization: transform obs to mean 0, std
209                 self._running_mean_std_rnd_obs.update(train_data.cpu().numpy())
210                 train_data = (train_data - self._running_mean_std_rnd_obs.mean)
211                 train_data = torch.clamp(
212                     train_data, min=self.cfg["obs_norm_clamp_min"], max=self.cfg
213                 )
214
215             predict_feature, target_feature = self.reward_model(train_data)
216             loss = F.mse_loss(predict_feature, target_feature.detach())
217             self.opt.zero_grad()
218             loss.backward()
219             self.opt.step()
220             self.scheduler.step()
221
222     def estimate(self, data: list) -> List[Dict]:
223         """
224         estimate the rnd intrinsic reward
225         """
226
227         obs = torch.stack(data).to(device)
228         if self.cfg["obs_norm"]:
229             # Note: observation normalization: transform obs to mean 0, std 1
230             obs = (obs - self._running_mean_std_rnd_obs.mean) / self._running_me
231             obs = torch.clamp(obs, min=self.cfg["obs_norm_clamp_min"], max=self.
232
233         with torch.no_grad():
234             self.estimate_cnt_rnd += 1
235             predict_feature, target_feature = self.reward_model(obs)

```

```

236         mse = F.mse_loss(predict_feature, target_feature, reduction='none').
237         self.tb_logger.add_scalar('rnd_reward/mse', mse.cpu().numpy().mean())
238
239         # Note: according to the min-max normalization, transform rnd reward
240         rnd_reward = mse * self.cfg["reward_mse_ratio"]  #(mse - mse.min())
241
242         self.tb_logger.add_scalar('rnd_reward/rnd_reward_max', rnd_reward.ma
243         self.tb_logger.add_scalar('rnd_reward/rnd_reward_mean', rnd_reward.m
244         self.tb_logger.add_scalar('rnd_reward/rnd_reward_min', rnd_reward.mi
245
246         rnd_reward = torch.chunk(rnd_reward, rnd_reward.shape[0], dim=0)
247
248     def training(config, train_data, test_data):
249         rnd_reward_model = RndRewardModel(config=config)
250         for i in range(train_config["train_iter"]):
251             rnd_reward_model.train([torch.Tensor(item["last_observation"]) for item
252             rnd_reward_model.estimate([torch.Tensor(item["last_observation"]) for it
253
254     def main():
255         env = gym.make("MiniGrid-Empty-8x8-v0")
256         env_obs = FlatObsWrapper(env)
257
258         train_data = []
259         test_data = []
260
261         for i in range(train_config["train_iter"]):
262
263             train_data_per_iter = []
264
265             while len(train_data_per_iter) < train_config["train_data_count"]:
266                 last_observation, _ = env_obs.reset()
267                 terminated = False
268                 while terminated != True and len(train_data_per_iter) < train_config
269                     action = env_obs.action_space.sample()
270                     observation, reward, terminated, truncated, info = env_obs.step(
271                     train_data_per_iter.append(
272                         {
273                             "last_observation": last_observation,
274                             "action": action,
275                             "reward": reward,
276                             "observation": observation
277                         }
278                     )
279                     last_observation = observation
280             env_obs.close()
281
282         train_data.append(train_data_per_iter)

```

```

283
284 while len(test_data) < train_config["test_data_count"]:
285     last_observation, _ = env_obs.reset()
286     terminated = False
287     while terminated != True and len(train_data_per_iter) < train_config["te
288         action = env_obs.action_space.sample()
289         observation, reward, terminated, truncated, info = env_obs.step(acti
290         test_data.append(
291             {
292                 "last_observation": last_observation,
293                 "action": action,
294                 "reward": reward,
295                 "observation": observation
296             }
297         )
298         last_observation = observation
299     env_obs.close()
300
301 p0 = Process(target=training, args=(little_RND_net_config, train_data, test_
302 p0.start()
303
304 p1 = Process(target=training, args=(small_RND_net_config, train_data, test_d
305 p1.start()
306
307 p2 = Process(target=training, args=(standard_RND_net_config, train_data, tes
308 p2.start()
309
310 p3 = Process(target=training, args=(large_RND_net_config, train_data, test_d
311 p3.start()
312
313 p4 = Process(target=training, args=(very_large_RND_net_config, train_data, t
314 p4.start()
315
316 p0.join()
317 p1.join()
318 p2.join()
319 p3.join()
320 p4.join()
321
322 if __name__ == "__main__":
323     mp.set_start_method('spawn')
324     main()

```

题目2（应用实践）

在课程第四讲（解密稀疏奖励空间）几个应用中任选一个

- minigrid 迷宫（奖励的稀疏性）
- metadrive 自动驾驶（奖励的多尺度变化）

根据课程组给出的[示例代码](#)，训练得到相应的智能体。最终提交需要上传相关训练代码、日志截图或最终所得的智能体效果视频（replay），具体样式可以参考第四讲的[示例 ISSUE](#)。

参考文献

- [1] https://en.wikipedia.org/wiki/Wasserstein_metric
- [2] Auer P. Using confidence bounds for exploitation-exploration trade-offs[J]. Journal of Machine Learning Research, 2002, 3(Nov): 397-422.
- [3] https://en.wikipedia.org/wiki/Thompson_sampling
- [4] <https://github.com/openscilab/awesome-exploration-rl>
- [5] <https://minigrid.farama.org/environments/minigrid/>