

PPO × Family 第七讲习题作业



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.7 (第几节课) + 作业提交日期

示例：【PPO × Family】+ 喵小DI + vol.7 + 20230223

- 代码实践题提交方式：

PPO × Family 官方GitHub 上发起 Pull Request

- 地址：[PPOxFamily/chapter7_tricks/hw_submission](https://github.com/PPOxFamily/chapter7_tricks/hw_submission)
- PR示例：<https://github.com/opendilab/PPOxFamily/pull/5>
- 命名规范：

hw_submission(学生名称): add hw7 (第几节课) + 作业提交日期

示例：hw_submission(nyz): add hw7_20230104

提交 **截止时间为 2023.6.29 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 (GAE- λ)

(单项选择题) GAE- λ [1] 是优势函数估计的一种办法，它的数学形式是：

$$A_t^\lambda = \sum_{i=0}^{n-1} [(1-\lambda)\lambda^i (\sum_{j=0}^i (\gamma^j r_{t+j}) + \gamma^{i+1} V(s_{t+i+1}) - V(s_t))] \\ + [\lambda^n (\sum_{j=0}^n (\gamma^j r_{t+j}) + \gamma^{n+1} V(s_{t+n+1}) - V(s_t))]$$

当任务环境中，奖励 r_t 的方差特别大，算法中 λ 的选取应该：

- A、调大
- B、调小
- C、不变

题目2 (Entropy bonus)

(单项选择题) 对于一个 N 维的离散动作空间，PPO [2] 算法中最终加到整体 loss function 上的 entropy bonus 项的实现形式是？

- A、 $\frac{1}{T} \sum_{i=1}^T \sum_{j=1}^N \log(\pi(a_j|s_i))$
- B、 $-\frac{1}{T} \sum_{i=1}^T \sum_{j=1}^N \pi(a_j|s_i) \log(\pi(a_j|s_i))$
- C、 $\frac{1}{T} \sum_{i=1}^T \sum_{j=1}^N \pi(a_j|s_i) \log(\pi(a_j|s_i))$

题目3 (Gradient clipping)

(单项选择题) 梯度裁剪 (Gradient clipping) [3] 是一种有效的控制模型训练过程中梯度数值大小的训练技巧。通过设置一个最大的允许数值，如果梯度的范数超过该数值，则将梯度乘以一个系数修正到这个数值：

$$\nabla f \leftarrow C \times \frac{\nabla f}{\|\nabla f\|}, \text{ if } \|\nabla f\| \geq C$$

请问它一般应该在训练流程的哪个位置被应用？

- A、forward 后 backward 前

- B、backward 后 optimizer.step 前
- C、optimizer.step 后
- D、具体情况具体分析，没有标准方案

题目4（MC & TD）

（问答题）在强化学习算法中，为了学习和训练价值函数，一般有两种较为常见的预测范式。

1、蒙特卡洛估计（Monte Carlo Prediction）

初始化策略 π 与价值函数 $V(s)$ ，使用现有的策略生成一段决策轨迹，包括状态、动作和奖励的序列 $(s_t, a_t, r_t) \sim \tau$ ，对于该序列中的某一时刻的状态 s_t ，计算该时刻状态之后**所有奖励的和作为的累计回报**，（对于无限长序列，可使用折扣系数 γ ）。将多个序列的累积回报的期望作为该策略下，状态 s 对应的状态价值 $V(s)$ 的估计：

$$\hat{V}(s_t) = \mathbb{E}[G_t] = \mathbb{E}\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i\right]$$

2、时序差分法估计（Temporal Difference Prediction, TD Prediction）

初始化策略 π 与价值函数 $V(s)$ ，使用现有的策略生成一段决策轨迹，包括状态、动作和奖励的序列 $(s_t, a_t, r_t) \sim \tau$ ，对于该序列中的每一个状态 s ，计算通过该状态之后，**一定步长的奖励的累计值与下一个状态的价值函数的和**，近似作为通过该状态之后所有奖励的累计回报，（对于无限长序列，可对累计的过程使用折扣系数 γ ）。将多个序列的累积回报的期望作为该策略下，状态 s 对应的状态价值 $V(s)$ 的估计，对于步长为 n ，价值函数的 n-step TD($\lambda=1$) 估计计算式为：

$$\hat{V}(s_t) = \mathbb{E}\left[\left(\sum_{i=t}^{t+n} \gamma^{i-t} r_i\right) + \gamma^{n+1} V(s_{t+n+1})\right]$$


当步长 $n = 1$ ，价值函数的 TD(λ) 计算式为：

$$\hat{V}(s_t) = \mathbb{E}[r_t + \gamma V(s_{t+1})]$$

此外，在训练中，往往使用一个小的的学习率 α 控制价值函数的更新学习速度：

$$V(s_t) \leftarrow V(s_t) + \alpha(\hat{V}(s_t) - V(s_t))$$

1. 假如有一个二维网格寻路的智能体决策场景，智能体从起点A1出发，前往终点C3，如下所示：

right	1	2	3
A	 →	↓	
B		→	↓

C			
---	--	--	--

在每一个状态上智能体只有向右移动，和向下移动两种动作，每个状态下每种动作的奖励记录在以下两张表格中：

Reward for →	1	2	3
A	4	1	0
B	3	7	0
C	7	8	0

Reward for ↓	1	2	3
A	7	3	8
B	4	1	6
C	0	0	0

假如初始策略 π 为均匀分布，价值函数 $V(s)$ 被初始化为零，折扣系数 $\gamma = 1$ ，学习率 $\alpha = 0.01$ 。

由于均匀的初始策略对应着均等概率的决策序列，请分别计算，这些序列对应的策略蒙特卡洛法和时序差分法 在步长 $n = 1$ 时， $TD(\lambda)$ 估计的价值函数，为了简化起见，仅需要计算 $\hat{V}(s_0)$ 。

提示：一共有6组决策序列，分别为：

A1→A2→A3→B3→C3； A1→A2→B2→B3→C3； A1→A2→B2→C2→C3

A1→B1→B2→B3→C3； A1→B1→B2→C2→C3； A1→B1→C1→C2→C3

对于策略蒙特卡洛法，假如序列为 A1→A2→A3→B3→C3，那么对应的累计奖励之和为：

$$G_0 = 4 + 1 + 8 + 6 = 19$$

同理，A1→A2→B2→B3→C3： $G_0 = 4 + 3 + 7 + 6 = 20$

A1→A2→B2→C2→C3： $G_0 = 4 + 3 + 1 + 8 = 16$

A1→B1→B2→B3→C3： $G_0 = 7 + 3 + 7 + 6 = 23$

A1→B1→B2→C2→C3： $G_0 = 7 + 3 + 1 + 8 = 19$

A1→B1→C1→C2→C3： $G_0 = 7 + 4 + 7 + 8 = 26$

对于时序差分法，假如序列为 $A1 \rightarrow A2 \rightarrow A3 \rightarrow B3 \rightarrow C3$ ，步长 $n = 1$ 时，TD(λ) 估计的奖励的累计回报为：

$$G_0 = 4 + V(s_1) = 4$$

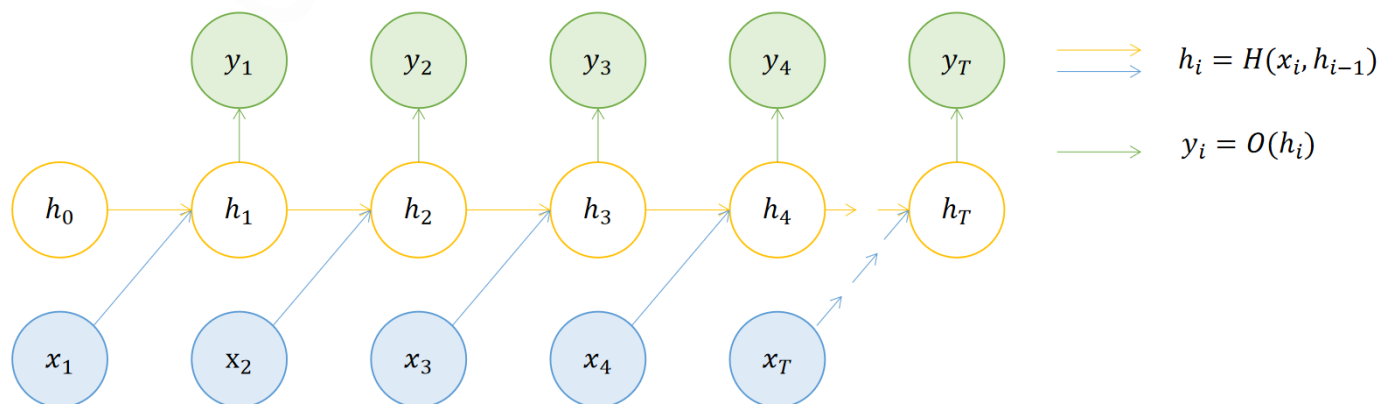
假如序列为 $A1 \rightarrow B1 \rightarrow B2 \rightarrow B3 \rightarrow C3$ ： $G_0 = 7 + V(s_1) = 7$

2. 请计算这两种方法估计的得到的价值函数 $\hat{V}(s_0)$ 的方差。
3. 结合上述简单的例子，分析和讨论，为什么在价值函数的训练过程中，蒙特卡洛法估计具有无偏性，但却往往对应着更大的方差；而时序差分法估计的结果是有偏的，但有时可以降低估计的方差。

题目5 (Orthogonal initialization)

(问答题) 深度神经网络的训练过程中经常会面临梯度爆炸与梯度消失的问题，从而影响训练过程的收敛性和稳定性，尤其是 RNN 等深层网络。为了稳定训练过程，机器学习研究者们使用了诸如梯度裁剪等各种不同的技巧来处理这一情况。另一个常用的技巧是参数的正交初始化 (Orthogonal initialization)。本题旨在介绍神经网络参数正交初始化的方法和原理。

回顾第五节课的理论题第一题的场景：下图是一个由 Recurrent Neural Network (RNN) 组成的序列到序列 (Seq2Seq) 模型的示意图，它的输入 x_t 是一个序列， $x_t \in \mathbb{R}^{d_x}$ ，输出也是一个序列， $o_t \in \mathbb{R}^{d_y}$ ，序列最大长度为 T ：



每一个时刻输入的信息 x_t ，都会与上一个时刻的隐藏状态 h_{t-1} ，一起通过转移函数 H ，生成当前时刻的隐藏状态：

$$h_t = H(x_t, h_{t-1} \mid \theta)$$

$$H : \mathbb{R}^{d_h \times d_x} \rightarrow \mathbb{R}^{d_h}$$

而每个时刻的隐藏状态 h_t ，通过一个输出函数 $o_t = O(h_t \mid \theta)$ ，输出当前时刻隐藏状态 h_t 的对应输出值 o_t 。如果将单个时刻的损失函数记为 $l(o_t, y_t)$ ，则所有时刻的平均损失为：

$$L = \frac{1}{T} \sum_{i=0}^T l(o_i(x_{1:i}), y_i)$$

在 $t = i$ 时刻的损失函数关于模型参数的梯度，会通过时刻 $t = i - 1$ 的隐藏状态向过去传递。RNN 模型的目标函数在训练时的梯度表达式， $\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial \theta}$ ，中将会出现类似如下形式的项：

$$\left[\frac{\partial l(o_i, y_i)}{\partial o_i} \frac{\partial O(h_i)}{\partial h_i} \frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}} \frac{\partial H(x_{i-1}, h_{i-2})}{\partial h_{i-2}} \frac{\partial H(x_{i-2}, h_{i-3})}{\partial h_{i-3}} \frac{\partial H(x_{i-3}, h_{i-4})}{\partial h_{i-4}} \dots \right]$$

假设转移函数 H 由一个全连接层与一个 ReLU 层构成，即：

$$h_t = H(x_t, h_{t-1} \mid \theta) = \text{ReLU}(Wh_{t-1} + Vx_t)$$

那么于是有：

$$\frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}} = \frac{\partial \text{ReLU}(\cdot)}{\partial (\cdot)} \frac{\partial (Wh_{t-1} + Vx_t)}{\partial h_{i-1}} = \frac{\partial \text{ReLU}(\cdot)}{\partial (\cdot)} \times W$$

当 $(Wh_{t-1} + Vx_t) \geq 0$ ，ReLU 层为非整流状态，那么上述梯度项将变为：

$$\left[\frac{\partial l(o_i, y_i)}{\partial o_i} \frac{\partial O(h_i)}{\partial h_i} \times W \times W \times W \times W \dots \right]$$

1. 假定 $d_h = D$ ，神经网络的参数矩阵 W 是一个 $D \times D$ 的矩阵，假如它被初始化为可以被奇异值分解成以下形式的矩阵：

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1D} \\ w_{21} & w_{22} & \dots & w_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D1} & w_{D2} & \dots & w_{DD} \end{bmatrix} = U \begin{bmatrix} \lambda_{11} & & & \\ & \lambda_{22} & & \\ & & \ddots & \\ & & & \lambda_{DD} \end{bmatrix} V^T = U \Lambda V^T$$

其中， U 和 V 是两个 $D \times D$ 的正交矩阵（Orthogonal matrix）[4]， Λ 是参数矩阵 W 的特征值构成的对角矩阵。

请分析 $W^N = W \times W \times W \dots \times W$ 的绝对大小与 Λ 的内在关系。

2. 结合第一问中的分析结果，讨论为什么这里需要将 RNN 的参数初始化为正交矩阵。

代码实践题

题目1（神经网络初始化）

课程第七讲的第五重境界——笨鸟先飞之 Init 部分讲解了神经网络参数初始化的设计思路，以及 PPO 算法在网络初始化部分所涉及的两种特殊技巧：**正交初始化和策略输出层的特殊缩放初始化**。现在，对于下方示例代码中给出的 PPO 算法神经网络示例代码（对应图片观察空间+连续动作空间），**请填写完成代码段中的 `init_weights` 函数，实现上文中提到的两种特殊初始化技巧。**

完整代码如下（也可从官网[链接](#)下载）：

```
1 from typing import Optional, Tuple, List
2 import torch
3 import torch.nn as nn
4 import treetensor.torch as ttorch
5
6
7 class PPOFModel(nn.Module):
8     mode = ['compute_actor', 'compute_critic', 'compute_actor_critic']
9
10    def __init__(
11        self,
12        obs_shape: Tuple[int],
13        action_shape: int,
14        encoder_hidden_size_list: List = [128, 128, 64],
15        actor_head_hidden_size: int = 64,
16        actor_head_layer_num: int = 1,
17        critic_head_hidden_size: int = 64,
18        critic_head_layer_num: int = 1,
19        activation: Optional[nn.Module] = nn.ReLU(),
20    ) -> None:
21        super(PPOFModel, self).__init__()
22        self.obs_shape, self.action_shape = obs_shape, action_shape
23
24        # encoder
25        layers = []
26        input_size = obs_shape[0]
27        kernel_size_list = [8, 4, 3]
28        stride_list = [4, 2, 1]
29        for i in range(len(encoder_hidden_size_list)):
30            output_size = encoder_hidden_size_list[i]
```

```

31         layers.append(nn.Conv2d(input_size, output_size,
kernel_size_list[i], stride_list[i]))
32         layers.append(activation)
33         input_size = output_size
34         layers.append(nn.Flatten())
35         self.encoder = nn.Sequential(*layers)
36
37         flatten_size = input_size = self.get_flatten_size()
38         # critic
39         layers = []
40         for i in range(critic_head_layer_num):
41             layers.append(nn.Linear(input_size, critic_head_hidden_size))
42             layers.append(activation)
43             input_size = critic_head_hidden_size
44         layers.append(nn.Linear(critic_head_hidden_size, 1))
45         self.critic = nn.Sequential(*layers)
46         # actor
47         layers = []
48         input_size = flatten_size
49         for i in range(actor_head_layer_num):
50             layers.append(nn.Linear(input_size, actor_head_hidden_size))
51             layers.append(activation)
52             input_size = actor_head_hidden_size
53         self.actor = nn.Sequential(*layers)
54         self.mu = nn.Linear(actor_head_hidden_size, action_shape)
55         self.log_sigma = nn.Parameter(torch.zeros(1, action_shape))
56
57         # init weights
58         self.init_weights()
59
60         def init_weights(self) -> None:
61             # You need to implement this function
62             raise NotImplementedError
63
64         def get_flatten_size(self) -> int:
65             test_data = torch.randn(1, *self.obs_shape)
66             with torch.no_grad():
67                 output = self.encoder(test_data)
68             return output.shape[1]
69
70         def forward(self, inputs: torch.Tensor, mode: str) -> torch.Tensor:
71             assert mode in self.mode, "not support forward mode:
{}/{}".format(mode, self.mode)
72             return getattr(self, mode)(inputs)
73
74         def compute_actor(self, x: torch.Tensor) -> torch.Tensor:
75             x = self.encoder(x)

```



```

76         x = self.actor(x)
77         mu = self.mu(x)
78         log_sigma = self.log_sigma + torch.zeros_like(mu) # addition aims to
broadcast shape
79         sigma = torch.exp(log_sigma)
80         return ttorch.as_tensor({'mu': mu, 'sigma': sigma})
81
82     def compute_critic(self, x: ttorch.Tensor) -> ttorch.Tensor:
83         x = self.encoder(x)
84         value = self.critic(x)
85         return value
86
87     def compute_actor_critic(self, x: ttorch.Tensor) -> ttorch.Tensor:
88         x = self.encoder(x)
89         value = self.critic(x)
90         x = self.actor(x)
91         mu = self.mu(x)
92         log_sigma = self.log_sigma + torch.zeros_like(mu) # addition aims to
broadcast shape
93         sigma = torch.exp(log_sigma)
94         return ttorch.as_tensor({'logit': {'mu': mu, 'sigma': sigma}, 'value':
value})
95
96
97 def test_ppof_model() -> None:
98     model = PPOFModel((4, 84, 84), 5)
99     print(model)
100    data = torch.randn(3, 4, 84, 84)
101    output = model(data, mode='compute_critic')
102    assert output.shape == (3, 1)
103    output = model(data, mode='compute_actor')
104    assert output.mu.shape == (3, 5)
105    assert output.sigma.shape == (3, 5)
106    output = model(data, mode='compute_actor_critic')
107    assert output.value.shape == (3, 1)
108    assert output.logit.mu.shape == (3, 5)
109    assert output.logit.sigma.shape == (3, 5)
110    print('End...')
111
112
113 if __name__ == "__main__":
114     test_ppof_model()

```

题目2（应用实践）

在课程第七讲（挖掘黑科技）几个应用中任选一个

- Atari 中的 Qbert 和 SpaceInvaders 环境（离散动作基准学术环境）
- MuJoCo 中的 Hopper 和 Halfcheetah 环境（连续动作基准学术环境）

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七讲的[示例 ISSUE](#)。

参考文献

- [1] Schulman J, Moritz P, Levine S, et al. High-dimensional continuous control using generalized advantage estimation[J]. arXiv preprint arXiv:1506.02438, 2015.
- [2] Schulman J, Wolski F, Dhariwal P, et al. Proximal policy optimization algorithms[J]. arXiv preprint arXiv:1707.06347, 2017.
- [3] Zhang J, He T, Sra S, et al. Why gradient clipping accelerates training: A theoretical justification for adaptivity[J]. arXiv preprint arXiv:1905.11881, 2019.
- [4] https://en.wikipedia.org/wiki/Orthogonal_matrix