中国科学技术大学

University of Science and Technology of China

人工智能原理与技术

实验 4

姓名：高茂航

学号：PB22061161

日期：2024.7.6

实验 4

1 Torch 配置

```
PS F:\Desktop\Learning\Computer\AI\AI-Homework\Lab4 - Project> nvidia-smi
Mon Jul  1 21:42:46 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 531.14                 Driver Version: 531.14       CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                      TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 3050 L...  WDDM | 00000000:01:00.0 Off |                  N/A |
| N/A   40C    P0              12W /  N/A|      0MiB /  4096MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

```
PS F:\Desktop\Learning\Computer\AI\AI-Homework\Lab4 - Project> python
Python 3.9.18 (main, Sep 11 2023, 14:09:26) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> if torch.cuda.is_available():
...     print("current device: cuda.")
... else:
...     print("CUDA is not available.")
...
current device: cuda.
```

2 实验原理

2.1 二元零和马尔可夫博弈

一个涉及两个玩家在一系列状态中进行决策的过程，其中每个玩家的目标是最大化自己的累计奖励（或最小化对方的累计奖励），而这个过程的状态转移遵循马尔可夫性质。在每个状态，玩家需要基于当前的信息选择最优策略，而这个选择会影响到游戏的下一个状态和玩家的即时奖励。由于是零和博弈，一个玩家的收益等于另一个玩家的损失，使得这种博弈具有高度的竞争性。

2.2 Naive Self Play

让 AI 与其自身的副本进行对战，通过这种方式根据自身经验不断学习适应并调整，以提高其性能。该方法较为简单直接，如果要避免不收敛等问题，需要采取更复杂的算法。

2.3 Actor-Critic

这种方法旨在通过同时学习一个策略（即 Actor）和一个值函数（即 Critic）来平衡探索和利用，从而提高学习效率和性能。Actor 根据当前策略选择动作，并执行该动作。Critic 评估执行动作后的结果，即计算实际回报与预期回报之间的差异（TD 误差）。接着使用 TD 误差来更新 Critic 的值函数参数，使其预测更加准确。最后根据 Critic 提供的反馈（TD 误差），更新 Actor 的策略参数，以便在未来选择更好的动作。

3 模型设计

```
  class Actor(nn.Module):
    def __init__(self, board_size: int, lr=1e-4):
        super().__init__()
        self.board_size = board_size
        # BEGIN YOUR CODE
        self.conv_blocks = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, padding=1),
            nn.LeakyReLU(0.01),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.LeakyReLU(0.01),
        )
        self.linear_blocks = nn.Sequential(
            nn.Linear(in_features=board_size ** 2 * 128, out_features=board_size ** 2),
            nn.LeakyReLU(0.01),
            nn.Dropout(0.5),
        )
        # END YOUR CODE

        # Define your optimizer here, which is responsible for calculating the gradients and performing optimizations.
        # The learning rate (lr) is another hyperparameter that needs to be determined in advance.
        self.optimizer = torch.optim.Adam(params=self.parameters(), lr=lr)

    def forward(self, x: np.ndarray):
        if len(x.shape) == 2:
            output = torch.tensor(x).to(device).to(torch.float32).unsqueeze(0).unsqueeze(0)
        else:
            output = torch.tensor(x).to(device).to(torch.float32)
    # BEGIN YOUR CODE
    # 将output中不合法的位置置为零
        x_flat = output.flatten(start_dim=2)
        mask = (x_flat == 0).float()

        output = self.conv_blocks(output)
        output = nn.Flatten()(output)
        output = self.linear_blocks(output)
        mask = mask.reshape(-1, self.board_size ** 2)
        output = output.reshape(-1, self.board_size ** 2)
        output = output * mask

    #以下几步解决output的归一化问题
        # 找出output中非零的位置
        non_zero_mask = output != 0
        # 把output中的零替换为一个很小的数
        output_adjusted = torch.where(non_zero_mask, output, torch.tensor(-1e10).to(output.device))
        # 对output_adjusted进行softmax操作
        softmax_output_adjusted = nn.functional.softmax(output_adjusted, dim=1)
        # 把softmax_output_adjusted中的非零位置的值替换回output中的值
        softmax_output_final  = torch.where(non_zero_mask, softmax_output_adjusted, output)
        output = softmax_output_final
        # END YOUR CODE

        return output


class Critic(nn.Module):
    def __init__(self, board_size: int, lr=1e-4):
```

```
57          super().__init__()
58          self.board_size = board_size
59          # BEGIN YOUR CODE
60          self.conv_layers = nn.Sequential(
61              nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1),
62              nn.LeakyReLU(0.01),
63              nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
64              nn.LeakyReLU(0.01),
65              nn.Flatten(),
66          )
67          self.fc_layers = nn.Sequential(
68              nn.Linear(in_features=board_size ** 2 * 128, out_features=board_size ** 2),
69              nn.LeakyReLU(0.01),
70              nn.Dropout(0.5),
71          )
72          # END YOUR CODE
73
74          self.optimizer = torch.optim.Adam(params=self.parameters(), lr=lr)
75
76      def forward(self, x: np.ndarray, action: np.ndarray):
77          indices = torch.tensor([_position_to_index(self.board_size, x, y) for x, y in action
        ]).to(device)
78          if len(x.shape) == 2:
79              output = torch.tensor(x).to(device).to(torch.float32).unsqueeze(0).unsqueeze(0)
80          else:
81              output = torch.tensor(x).to(device).to(torch.float32)
82
83          # BEGIN YOUR CODE
84          output = self.conv_layers(output)
85          q_values = self.fc_layers(output)
86          q_values = q_values.view(-1, self.board_size ** 2)
87          output = q_values.gather(1, indices.unsqueeze(1)).squeeze(1)
88
89          # END YOUR CODE
90
91          return output
92
```

一开始采用如下结构，结果最后 output 全为 0，胜率为 0.53，原因可能是：

(1) 如果网络参数权重初始化过小，经过 ReLU 激活函数后，所有负值都会被置为 0；

(2) 学习率设置不合理，可能导致网络权重在训练初期就更新到了一个不合理的范围，从而导致输出全为 0；

(3) 在深层网络中，梯度可能会随着传播逐渐减小，最终导致梯度消失。

将 nn.ReLU() 改为 nn.LeakyReLU(0.01) 并加上 nn.Dropout(0.5) 后 output 就正常了，上述问题可能主要是由第一个原因造成的。

```
1       self.conv_layers = nn.Sequential(
2         nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1),
3         nn.ReLU(),
4         nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1),
5         nn.ReLU(),
6         nn.Flatten(),
7       )
8       self.fc_layers = nn.Sequential(
9         nn.Linear(in_features=board_size ** 2 * 128, out_features=board_size ** 2),
10        nn.ReLU(),
11      )
12
```

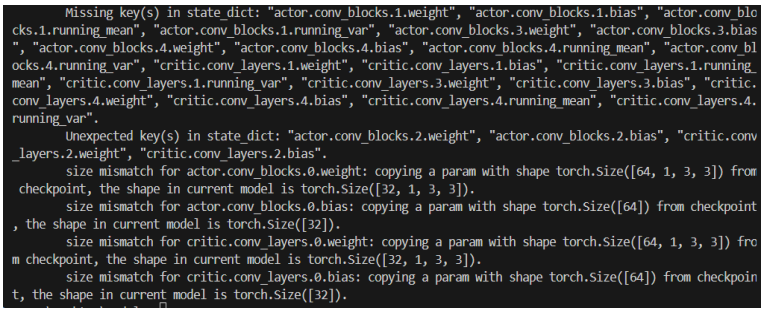
4 constrained policy 的解决思路

具体过程在上一部分的 forward 部分代码中体现，主要是将非法位置置为 0 和归一化两部分。前者一开始碰到的问题是理解错代码，以为每次都要加载一个棋盘，而不是处理 x，导致报错；后者的问题在于，假如对整个张量运用 softmax 函数，会导致非法位置归一化后不为 0，而若只对合法位置归一化就很麻烦，后来查了资料得知可以把非法位置的值替换为一个很小的负数，然后再对整体进行 softmax 操作，这样操作后非法位置还是 0。

5 optimize 函数中 2 个 bug 的的解决方案

在 actor_loss.backward() 和 critic_loss.backward() 后面分别加上 self.actor.optimizer.step() 和 self.critic.optimizer.step()，因为这两对操作分别完成了 actor 和 critic 网络的梯度计算和参数更新，确保了每次迭代中，模型的参数都能根据损失函数的反向传播结果进行有效更新。

6 其他 bug 及解决方案



这个 bug 出现在 opponent_loader.py 中加载的模型结构不对时，由于当时在 evaluator.py 中把 random_response 设为 True，以为这样就不用管 opponent_loader.py 了，结果就发生了报错。

此外，在训练过程时偶尔会因为内存溢出而导致训练终止，一般要通过中止进程或重启来解决。

7 难以理解的点及解释

不太清楚的是保存到 model.pth 文件的模型是什么形式的，后来阅读代码和查阅资料后了解到训练并保存的模型是一个 PyTorch 模型的状态字典（state dictionary）。状态字典包含了模型中所有可训练参数（如权重和偏置）的映射。当使用 torch.save(model.state_dict(), f"checkpoints/model_{}.pth") 保存模型时，它不会保存模型的结构，只保存参数。这意味着在加载模型时，需要先定义模型的结构，然后才能加载这些参数。.pth 格式的文件是 PyTorch 的一种常用文件格式，一般用于存储模型参数。

8 loss 和 entropy 曲线分析

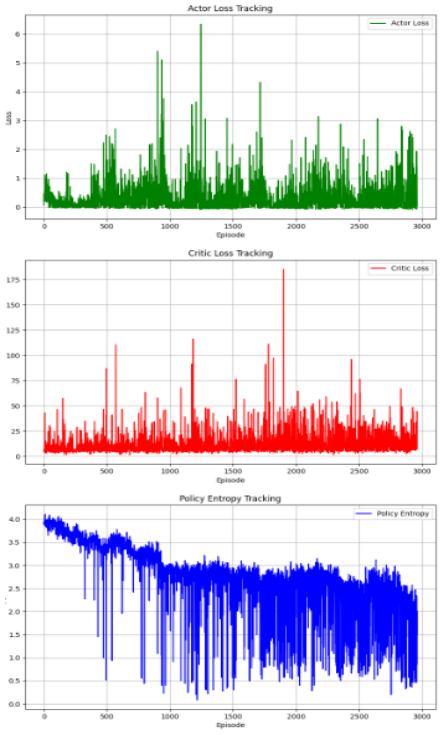
Actor 的 Loss 含义: Actor 的任务是根据当前状态选择动作，其目标是最大化期望奖励。Actor 的 loss 通常是基于策略梯度的，意在评估所选动作与最优动作之间的差距。这个 loss 的计算通常涉及到所选择动作的概率和这个动作带来的预期回报（或优势）。我们希望最大化这个预期回报，但在优化过程中，我们实际上是最小化一个负的期望回报（即最大化期望回报的负数形式），这样做是为了符合常规优化问题的形式，即最小化 loss。

Critic 的 Loss 含义: Critic 的任务是评估当前状态或动作的价值，即预测从当前状态开始，采取特定动作能获得的期望回报。Critic 的 loss 是基于值函数的预测误差，通常是时间差分（TD）误差或均方误差。这个误差衡量了 Critic 的预测值与实际获得的回报之间的差异。我们希望这个误差尽可能小，这意味着 Critic 的预测越准确越好。

在 Actor-Critic 架构中，policy 对应的熵是一个衡量 policy 随机性的指标。熵越高，表明 policy 的行为选择越随机；熵越低，表明 policy 的行为选择越确定性。在强化学习中，较高的熵有助于探索，可以防止算法过早地收敛到局部最优解，而较低的熵有助于算法在找到好的策略后稳定执行。因此，在训练初期，我们可能希望保持较高的熵以鼓励探索；在训练后期，当我们希望算法开始利用已学习的策略时，可能会逐渐降低熵的值。

9 对网络参数和结构的一些尝试

上述模型的相关图像如下：



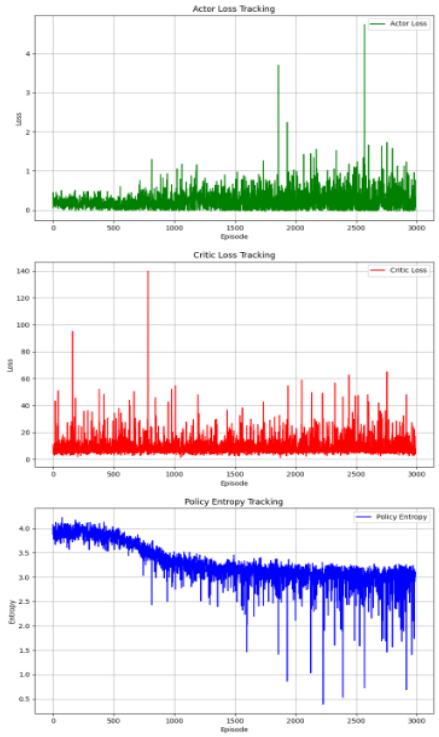
为了叙述方便，以下将该模型记为模型 1，它和随机噪声对弈的胜率为 1，和自己对弈的胜率约为 0.5。

试着把优化器换成随机梯度下降优化器（SGD），结果胜率明显下降，原因可能如下：Adam 优化器结合了具有自适应学习率的特性，这使得它在很多情况下都能比较快速地收

敛，并且对学习率的初始选择不是特别敏感。Adam 优化器能够通过计算梯度的一阶矩估计和二阶矩估计来调整每个参数的学习率，使得它在训练初期和面对稀疏梯度时表现更好。相比之下，SGD 优化器通过对每个参数更新时使用固定的学习率来进行梯度下降。虽然 SGD 在很多简单问题上表现良好，但在处理复杂的非凸优化问题或是深度神经网络时，由于缺乏自适应学习率调整，可能需要更多的时间来调整学习率和其他超参数以达到良好的收敛效果。此外，SGD 对于稀疏数据或是在训练初期可能不如 Adam 那样表现出色。

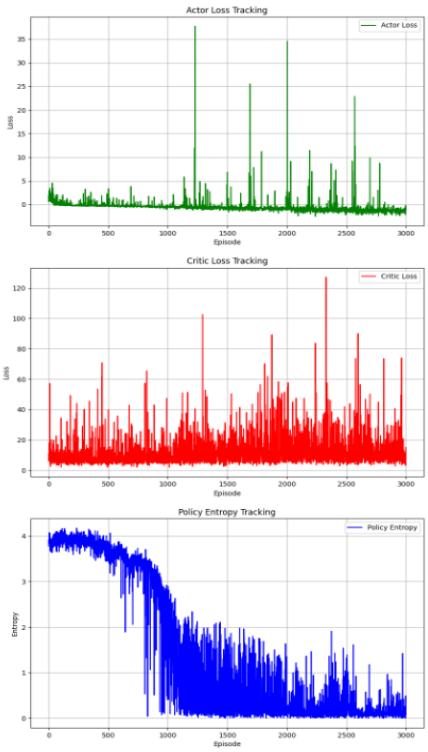
此外还试了 RMSprop 优化器，该优化器结合了梯度平方的指数衰减平均调整学习率，适合处理非平稳目标，但对于本实验的效果略差于 Adam 优化器。

尝试了将模型 1 的学习率改为 1e-5（记为模型 2），相关图像如下：



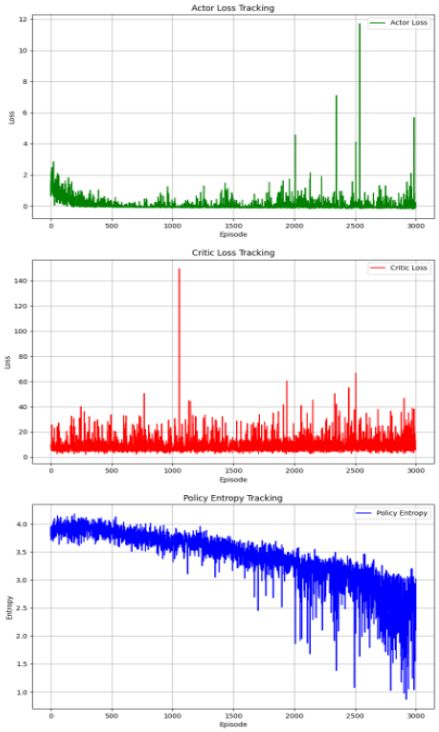
模型 2 和随机噪声对弈的胜率为 0.991，和自己对弈的胜率为 0.524，模型 2（先手）和模型 1 对弈先手胜率为 0.832，模型 1（先手）和模型 2 对弈先手胜率为 0.253，可见模型 2 由于模型 1（故提交的也是模型 2），运行 player.py 发现模型 2 是一直自顾自地连成一条直线而不去管对方，目前还不清楚这种策略是否有较好的泛化性能。

也尝试了把网络改为 1，32，128 的结构（记为模型 3），相关图像如下：



模型 3 和随机噪声对弈的胜率为 0.936，和自己对弈的胜率为 0.789。

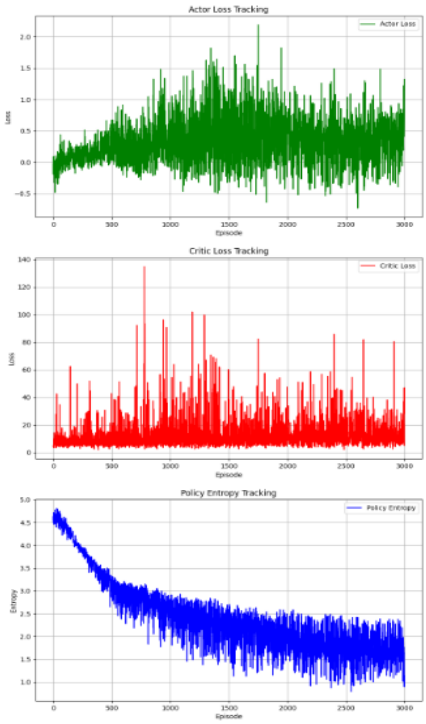
还尝试把模型 4 的学习率改为 1e-5，相关图像如下：



模型 4 和随机噪声对弈的胜率为 0.822，和自己对弈的胜率为 0.525。总的来说模型 3，4 不如模型 1，2 的效果好。

后来试着写了前馈神经网络 (FNN) 和循环神经网络 (RNN) 等，均由于对相关内容还不够熟悉，没能调试成功，最后尝试了 transformer，效果稍好于前述的 CNN 的模型，但训练

的时间更长，相关图像如下：



可见熵的曲线下降较快，而与上述其他图像差异最大的地方是 actor loss 在训练中后期较为均匀，且出现了负值。

此外本来还想增加迭代次数，但由于时间关系没能实现。一开始还碰到的一个问题是发现不同网络训练出来的模型不能直接对弈，后来发现是因为模型的参数不同，需要在 opponent_loader.py 里载入正确的模型结构。

10 思考题

在最理想的情况下，即使模型的权重收敛到使得 actor loss 和 critic loss 最优的点，黑棋和白棋双方的策略也不一定达成纳什均衡，原因如下：

（1）纳什均衡指的是在一个非合作博弈中，每个参与者选择的策略使得在其他参与者的策略给定的情况下，自己的策略最优，这意味着在纳什均衡点，没有任何一个参与者可以通过单方面改变自己的策略来获得更好的结果。而 actor loss 和 critic loss 的最优化目标是最小化预测误差和提高决策的准确性，这与博弈论中的策略最优化不完全相同；

（2）模型权重的收敛到最优点意味着在给定的训练数据和模型结构下，模型找到了一组参数，这组参数能够最小化 loss 函数。然而，这并不保证模型找到的是全局最优解，也不保证这个最优点就对应着纳什均衡。特别是在复杂的环境和策略空间中，存在多个局部最优解，而这些局部最优解可能并不对应于纳什均衡；

（3）在实际的博弈中，环境可能是动态变化的，参与者的策略也可能随时间变化。即使模型在某一时刻达到了 loss 的最优，也不能保证这个状态在环境变化后仍然是最优的，此外，实际环境中的不确定性和信息的不完全性也可能阻碍达成纳什均衡；

（4）纳什均衡的达成依赖于所有参与者策略的相互适应。即使一个参与者找到了自己的最优策略，如果其他参与者的策略改变，那么原来的最优策略可能不再是最优的。因此，即使

模型找到了最优的 loss 点，也需要所有参与者的策略共同适应才能达到纳什均衡。

11 课程反馈

本课程较好地锻炼了代码能力，加深了对人工智能常见算法的理解。虽然每次实验都感觉比较难，每次至少要花费整整几天的时间，但做下来还是很有收获，也提高了 debug 和独立思考的能力，感到特别困难的部分还是集中在数学推导方面。