

# 大作业

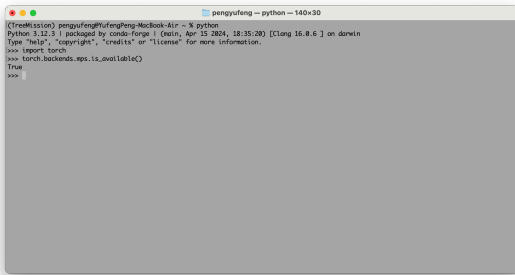
彭煜峰 PB22051087

2024 年 7 月 10 日

本次作业需独立完成，不允许任何形式的抄袭行为，如被发现会有相应惩罚。在上方修改你的姓名学号，说明你同意本规定。

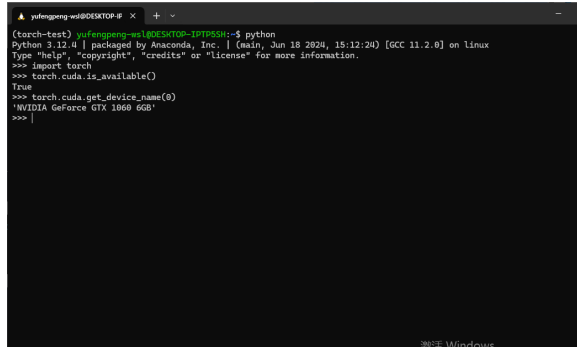
## 问题 0：环境配置

本次实验对网络的训练需要用到 torch 环境，我分别在 Windows Subsystem for Linux (WSL) 和 Apple Silicon 两种设备上对训练进行了测试，环境配置结果如图。



```
(base) penyufeng@penyufeng-MacBook-Air: ~ % python
Python 3.12.5 | packaged by conda-forge | (main, Apr 23 2024, 18:35:20) [Clang 16.0.6 ] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> import torch
>>> torch.backends.mps.is_available()
True
>>>
```

(a) torch for mps



```
(torch-test) yufeng@WSL:~$ python
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:12:20) [GCC 11.2.0] on Linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
>>> torch.cuda.get_device_name(0)
'NVIDIA GeForce GTX 1660 6GB'
>>>
```

(b) torch for wsl

图 1: torch 配置

## 问题 1：实验原理

### a. 二元零和马尔可夫博弈

二元零和马尔可夫博弈 (Two-Player Zero-Sum Markov Game) 是一类特殊的博弈论模型，常用于描述两个玩家之间的竞争性决策过程。这个模型结合了马尔可夫决策过程和零和博弈的特性。以下是其基本概念和结构：

#### 基本定义

- **状态空间 (State Space,  $S$ )** : 本次实验的状态空间为  $12 \times 12$  的棋盘的所有状态的集合；
- **动作空间 (Action Space,  $A$ )** : 本次实验的动作空间为棋盘上所有位置的集合；

- **状态转移概率 (Transition Probability,  $P(s'|s, a_1, a_2)$ )** : 给定当前状态  $s$  和两个玩家采取的行动  $a_1$  和  $a_2$ , 转移到下一个状态  $s'$  的概率。
- **奖励函数 (Reward Function,  $R(s, a_1, a_2)$ )** : 描述在当前状态  $s$  和玩家采取的行动  $a_1$  和  $a_2$  下, 玩家 1 获得的奖励。由于是零和博弈, 玩家 2 的奖励为玩家 1 奖励的相反数, 即  $-R(s, a_1, a_2)$ 。  
本次实验中, 奖励函数为

$$\mathcal{R}(s, a, s') = ((L_b^2(s') - L_w^2(s')) - ((L_b^2(s) - L_w^2(s))))$$

因此, 白棋黑棋的奖励满足零和博弈。

- **策略 (Policy,  $\pi$ )** : 玩家在每个状态下选择行动的概率分布。本次实验中采用 Actor 网络作为策略网络, 对概率进行预测。

## 特性

- **零和性质**: 在零和博弈中, 一个玩家的收益等于另一个玩家的损失。也就是说, 两个玩家的奖励总和始终为零。
- **马尔可夫性质**: 当前状态的决策仅依赖于当前状态和当前的行动, 不依赖于过去的状态和行动序列。这意味着状态转移和奖励函数都遵循马尔可夫性质。

## 目标

每个玩家的目标是通过选择最优策略, 最大化 (玩家 1) 或最小化 (玩家 2) 累积奖励的期望值。

## b. 简单自我博弈

Naive Self-Play (简单自我对弈) 是一种用于训练智能体的方法, 特别是在强化学习和博弈论中。这种方法通过让智能体与自己的过去版本进行对弈来提高其策略。在本次实验中, 当前游戏步骤下的 **action** 和 **response** 都是通过 Actor 网络产生 **policy** 来执行的, 应用了简单自我博弈。

## c. Actor-Critic 方法

Actor-Critic 方法是一种广泛应用于强化学习中的算法, 结合了策略优化 (Policy Optimization) 和价值估计 (Value Estimation) 的优点。这个方法将智能体的学习过程分为两个部分: 演员 (Actor) 和评论家 (Critic)。Actor 负责基于当前状态产生策略, 即转移概率矩阵, Critic 负责对 Actor 执行策略后的 state 进行评分。评分以环境给的奖励为目标, 用均方损失和时间差分算法来更新网络。Actor 利用策略梯度算法, 基于 Critic 网络的评分对网络进行更新。

## 时间差分算法 (Temporal-Difference Algorithm)

时间差分算法是一种用于评估策略的在线学习方法, 通过利用贝尔曼优化方程逐步更新价值函数。时间差分误差  $\delta$  定义为:

$$\delta = r + \gamma V_w(s') - V_w(s)$$

其中,  $r$  是即时奖励,  $\gamma$  是折扣因子,  $V_w(s)$  是状态  $s$  下的价值函数估计,  $s'$  是下一状态。

使用时间差分误差更新价值函数的参数  $\omega$ :

$$\omega \leftarrow \omega + \alpha \delta \nabla_{\omega} V_{\omega}(s)$$

其中,  $\alpha$  是学习率。这一公式体现了贝尔曼方程中的递归关系, 即当前状态的价值通过即时奖励和下一状态的价值来估计。

## 策略梯度

策略梯度算法是一种用于优化策略的强化学习方法, 通过直接优化策略函数来最大化累积奖励。策略梯度的更新规则基于评论家的反馈, 即时间差分误差  $\delta$ 。策略梯度更新规则为:

$$\theta \leftarrow \theta + \beta \delta \nabla_{\theta} \log \pi_{\theta}(a|s)$$

其中,  $\theta$  是策略网络的参数,  $\beta$  是策略的学习率,  $\pi_{\theta}(a|s)$  是在状态  $s$  下选择动作  $a$  的概率,  $\log \pi_{\theta}(a|s)$  是策略的对数概率。

## 问题 2: 网络设计

### a. Actor 网络

#### 编程

```
class Actor(nn.Module):

    def __init__(self, board_size: int, lr=lr_a):
        super().__init__()
        self.board_size = board_size

        # BEGIN YOUR CODE
        self.residual_net = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=7, padding=3, stride
                =1),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(kernel_size=3, padding=1, stride=1),
            Residual(in_channels=64, out_channels=64, use_one_d=False, stride=1),
            Residual(in_channels=64, out_channels=64, use_one_d=False, stride=1),
            Residual(in_channels=64, out_channels=128, use_one_d=True, stride=2),
            Residual(in_channels=128, out_channels=128, use_one_d=False, stride=1),
            Residual(in_channels=128, out_channels=256, use_one_d=True, stride=2),
            Residual(in_channels=256, out_channels=256, use_one_d=False, stride=1),
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(in_features=256, out_features=self.board_size**2)
        )
        # END YOUR CODE

        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
    mask = (nn.Flatten()(output) == 0)

    output = self.residual_net(output)
    output = torch.softmax(output, dim=(-1)) + 1e-10

    output = output * mask
    output = output / torch.sum(output, dim=(-1), keepdim=True)
    # END YOUR CODE
    return output
```

## 设计理由

该网络是一个残差网络，采用该网络结构能够保证优化过程中，新的函数空间永远包含旧的函数空间，从而避免了网络深度增加造成的负优化现象。

- 第一个卷积层使用  $7 \times 7$  的卷积核，对输入的  $12 \times 12$  棋盘进行卷积操作。通过使用 64 个卷积核，提取棋盘的低级特征。
- 对卷积层的输出进行批归一化，稳定训练过程，加速收敛。
- 最大池化操作减少特征图的尺寸，同时保留重要特征，并引入空间不变性。
- 通过多个残差块 (Residual Block) 提取更高级的特征。残差块可以有效解决深层网络中的梯度消失问题，并且在提取复杂特征时非常有效。
- 逐步增加通道数 (从 64 到 128，再到 256)，以提取更丰富的特征。
- 在特定残差块中，通过增加步幅实现下采样，减小特征图尺寸，提高计算效率，同时增加特征复杂性。
- 将每个通道的特征图缩小到  $1 \times 1$ ，即提取每个通道的全局信息。这一步将每个通道的特征压缩成单一值，同时保留全局信息。
- 将池化后的输出展平为一维向量，以便于输入全连接层。
- 将展平后的特征向量映射到棋盘上的每个位置 ( $12 \times 12$ )，输出为 144 维向量，表示在棋盘上每个位置落子的概率。

## constrained policy 的解决思路

首先基于传入的状态生成掩码 (mask)，然后将网络生成的概率转移矩阵与掩码做按元素乘积，将非法位置的概率置零，最后对概率矩阵做一次归一化处理，得到最终的概率转移矩阵。

## b. Critic 网络

### 编程

```
class Critic(nn.Module):

    def __init__(self, board_size: int, lr=lr_c):
        super().__init__()
        self.board_size = board_size
        # Define your NN structures here as the same. Torch modules have to be
        # registered during the initialization
        # process.

        # BEGIN YOUR CODE
        self.residual_layer = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=7, padding=3, stride
                =1),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(kernel_size=3, padding=1, stride=1),
            Residual(in_channels=64, out_channels=64, use_one_d=False, stride=1),
            Residual(in_channels=64, out_channels=64, use_one_d=False, stride=1),
            Residual(in_channels=64, out_channels=128, use_one_d=True, stride=2),
            Residual(in_channels=128, out_channels=128, use_one_d=False, stride=1),
            Residual(in_channels=128, out_channels=256, use_one_d=True, stride=1),
            Residual(in_channels=256, out_channels=256, use_one_d=False, stride=1),
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(in_features=256, out_features=self.board_size**2)
        )

        self.dense_layer_a = nn.Sequential(
            nn.Linear(in_features=1, out_features=8),
            nn.ReLU(),
            nn.Linear(in_features=8, out_features=16),
            nn.ReLU(),
            nn.Linear(in_features=16, out_features=32),
            nn.ReLU()
        )

        self.dense_layer_b = nn.Sequential(
            nn.Linear(in_features=self.board_size**2 + 32, out_features=256),
            nn.ReLU(),
            nn.Linear(in_features=256, out_features=128),
            nn.ReLU(),
            nn.Linear(in_features=128, out_features=64),
            nn.ReLU(),
            nn.Linear(in_features=64, out_features=1),
            nn.ReLU()
```

```
)  
# END YOUR CODE  
  
self.optimizer = torch.optim.Adam(params=self.parameters(), lr=lr)  
  
def forward(self, x: np.ndarray, action: np.ndarray):  
    indices = torch.tensor([_position_to_index(self.board_size, x, y) for x, y in  
        action]).to(device)  
    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  
    output = self.residual_layer(output)  
  
    indices = self.dense_layer_a(indices.reshape([indices.shape[0], 1]).float())  
    output = torch.cat((output, indices), dim=(-1))  
    output = self.dense_layer_b(output).squeeze(-1)  
    # END YOUR CODE  
  
    return output
```

## 设计理由

该网络通过对动作和状态的学习，输出对一局游戏中所有的动作的评分。在该网络的设计中，我同样利用了残差网络，和 Actor 网络的不同之处在于，我还将动作传入隐藏层进行特征提取，然后将 state 和 action 的特征进行拼接，再过一遍隐藏层，输出对动作的评分，这样使得网络能够有效地学习到动作和状态的特征。两个隐藏层分别对特征进行提取和压缩。

## 问题 3:optimize 中的 bug 解决

### a. 问题

该方法的问题在于计算梯度后没有对网络进行更新，由此导致学习无效。

### b. 编程

```
def optimize(self, policy, qs, actions, rewards, next_qs, gamma, eps=epsilon):  
  
    targets = rewards + gamma * next_qs  
    critic_loss = nn.MSELoss()(targets, qs)  
    indices = torch.tensor([_position_to_index(self.board_size, x, y) for x, y in  
        actions]).to(device)  
    aimed_policy = policy[torch.arange(len(indices)), indices]
```

```
actor_loss = -torch.mean(torch.log(aimed_policy + eps) * qs.clone().detach())

self.actor.optimizer.zero_grad()
actor_loss.backward()
nn.utils.clip_grad_norm_(self.actor.parameters(), max_norm=1.0)
self.actor.optimizer.step()

self.critic.optimizer.zero_grad()
critic_loss.backward()
nn.utils.clip_grad_norm_(self.critic.parameters(), max_norm=1.0)
self.critic.optimizer.step()
return actor_loss, critic_loss
```

## 问题 4: 学习中遇到的 bug

1. 张量维度不匹配, 在模型设计过程中, 经常遇到张量维度不匹配的问题, 即上一层输出的张量形状不满足下一层输入形状的要求;
2. 概率计算总和不为 1, 在一开始的处理中, 我首先对 Actor 网络的输出做了 softmax 激活, 然后将置零的部分平均分配加到其余部分, 如此处理导致训练过程中出现概率矩阵总和不为 1 的情况;
3. 归一化计算稳定性, 由于归一化计算存在除法, 因此会出现计算溢出现象, 为了解决该问题, 我给除数加上了一个极小的  $\epsilon$ , 以此来避免该问题。

## 问题 5: 难以理解的点和对实验的思考

1. 一开始从 `utils.py` 中没有看出零和博弈的体现, 感觉对 `opponent` 的奖励没有做什么约束, 后来发现其实黑棋的奖励函数同时限制了白棋的奖励;
2. 感觉训练过程中策略熵值下降的很快, 模型在大约 100 ~ 200 轮游戏后对动作空间的探索就开始降低了, 导致无法学习到非常好的策略, 或许可以通过调整奖励函数来鼓励模型探索更丰富的策略;
3. 训练过程很不稳定, 感觉模型训练过程受权重初始化影响很大, 同样的模型和参数在不同的训练中效果差别较大, 目前还没有找到好的解决方案, 或许可以采用更合理的神经网络结构?
4. Critic 网络似乎没有怎么更新, 从开始训练到结束 loss 都在震荡, 调整学习率也没有什么改变。
5. 模型可能会由于无法的好好的分数而摆烂, 由于熵值下降, 模型若无法获得较高的评分, 可能会由于拒绝探索而导致模型一直处于一个比较摆烂的状态, 或许可以通过调整奖励函数来优化;
6. 模型从完全随机的策略开始学习, 需要经过很多局游戏才能学到一个比较好的策略, 且由于一开始时, 模型的策略必然不能很好, 导致模型收敛情况的不确定性很强。因此可以考虑利用模仿学习先得到一个比较可靠的策略网络, 再根据该网络学习价值网络, 参考 Alpha-Go 的学习策略。由此可以让智能体学会一些人类的下棋思路, 在此基础上学习可以减少完全随机的策略探索, 优化模型训练。

## 问题 6: 分析 loss 和 entropy 曲线

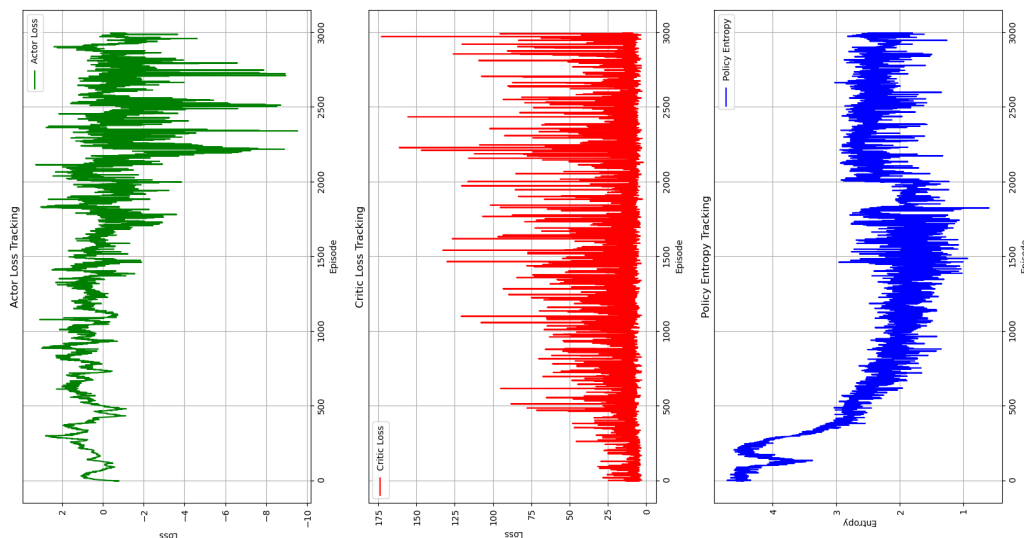


图 2: Loss 曲线

- Actor 网络的 loss 曲线代表其策略的有效程度，理论上会逐渐增大收敛到一个上界。从我的实验上来看，该曲线在前 200 局游戏中确实在上升，后期开始不稳定，且方差极大。
- Critic 网络的 loss 代表裁判的打分和环境奖励的接近程度，该曲线理论上应该开始时较高，逐渐降低趋于零，但是我的实验结果却是它一直在震荡，且方差越来越大。我不理解，我用该网络测试了别的模型，收敛速度极快。
- Entropy 曲线是策略的熵值，代表了模型探索动作空间的欲望，理论上应该是开始训练时较高，后来逐渐降低。

从实验 loss 曲线来看，模型一开始探索欲望较高，Actor 网络很快学习到了比较有效的策略，因此熵值开始下降，模型探索欲望下降。此时由于 Critic 网络的不稳定，导致评分不稳定，引起 Actor 网络也开始不稳定。观察图像可以发现，Critic 网络的 loss 突变（增大）点会使得 Actor 网络的 loss 降低，这是评分不准确引起的。在训练后期 Actor 网络的粉持续走低的情况下，模型的熵值开始上升，说明模型在调整探索策略，企图提高得分，然而还是由于 Critic 网络的不稳定，导致模型最终没有收敛到理想情况。

## 问题 7: 调整网络结构和超参数

由于实验结果一直不够理想，我尝试了多种网络结构，包括 LeNet, AlexNet 和 ResNet, 调整了卷积核的大小, 模型的深度等。但是我的实验结果表示模型对这些参数都不敏感, 尤其是 Critic 网络, 仍然保持震荡。但是模型对学习率的选取非常敏感, 只是我测试了多种学习率, 效果均不理想。学习率设置较大, 模型 loss 会双双趋于无穷, 设置过小则 Actor 网络 loss 可能稳定为零。且通过实验我发现, 该



网络似乎是一个不太稳定且初值敏感的系统，由于网络的初始化是随机的，因此同样的超参和网络结构，可能导致完全不同的训练结果。我实在没有找到合适的解法，这或许跟模型一开始完全随机下棋有关？但是训练出来的模型和随机噪声对弈又可以做到完胜，100% 的胜率。我实在是无法理解。

## 问题 8: 纳什均衡

不一定达成纳什均衡。纳什均衡是指在一个博弈中，每个玩家都选择了最优策略，且没有玩家可以通过单方面改变自己的策略来获得更好的结果。达到纳什均衡需要模型充分探索动作空间，我们的模型达到的最优点可能只是局部最优而不一定是全局最优。

## 课程反馈

这学期选修这门课我收获良多，从最简单的模型开始逐步了解了强化学习的原理、步骤，通过每次作业阅读代码，填写代码，我基本掌握了独立建模利用强化学习解决简单问题的能力。这门课唯一的问题是感觉很多代码没有让我们自己上手从零开始建模，使得在很多时候对于问题的关键有点一知半解，或许可以让我们多做一些代码上的工作，这样也能更好地理解理论。