人工智能实验

强化学习

期末实验

组员：陈家豪　18308013
李骁达　18340095

目录

1 组员分工

学号	姓名	工作	贡献度
18308013	陈家豪	构建神经网络，完成对抗训练过程，撰写实验报告	50%
18340095	李骁达	实现 Double DQN 模型，完成游戏主体，负责优化调参	50%

2 算法原理

2.1 强化学习

强化学习又称为增强学习，是指一类从与环境交互中不断学习的问题以及解决这类问题的方法。在人工智能领域，一般用**智能体 Agent** 来表示一个具备行为能力的物体，那么强化学习考虑的问题就是**智能体 Agent** 和**环境 Environment** 之间交互的任务。任务包含一系列的**动作 Action**、**观察 Observation** 和**反馈值 Reward**。所谓的观察是指智能体从当前环境中所获得的信息，其集合就是智能体当前所处的**状态 State**。智能体根据自己所处的状态，确定下一步执行的动作，而这个动作会与环境发生交互，使得环境发生改变的同时，给予智能体一个反馈值，以表示这个动作的好坏。换言之，反馈值相当于一个量化标准，智能体的最终目的就是学习一个最优策略，使得自己得到的奖励尽可能多。基于上述理论，一个强化学习包含以下基本要素：

(1) 状态 s，是对环境的描述，时刻 t 下的状态为 s_t；

(2) 动作 a，是对智能体行为的描述，时刻 t 下的动作为 a_t；

(3) 策略 $\pi(a|s)$，是智能体根据当前环境 s 决定下一步动作 a 的函数，是一个概率表示；

(4) 反馈值 $r(s,a,s')$，是一个标量函数，指的是智能体根据当前状态 s 执行动作 a 后达到新状态 s' 的这个过程中，环境会反馈给智能体的一个奖励值；

除此以外，强化学习的前提假设还包括两方面内容。一方面，时间是离散的、有先后顺序的，可以得到一个类似于 $\{s_0,a_0,r_0,\cdots,s_t,a_t,r_t\}$ 的离散序列作为数据样本。此外，如果输入是确定的、输出也是确定的，每一次参数的调整都会造成确定性的影响。另一方面，强化学习中的状态、动作和反馈序列是一个马尔科夫决策过程（Markov Decision Process，MDP）。在 MDP 中，“未来只取决于当前”，未来的状态只跟当前的状态和动作有关，换言之：

$$p(s_{t+1}|s_t,a_t,\cdots,s_0,a_0)=p(s_{t+1}|s_t,a_t) \tag{1}$$

基于以上理论，我们可以得到强化学习的其他要素：

(5) 总回报 $G(\tau)=\sum\limits_{t=0}^{T-1}\gamma^t r(s_t,a_t,s_{t+1})$，指的是智能体和环境一次交互过程的轨迹 τ 所收到的累计奖励。其中 γ 是折扣因子，用于平衡长期回报和短期回报的影响；

(6) 目标函数 $\mathcal{F}(\theta)=\mathbb{E}_{\tau\sim p_\theta(\tau)}[G(\tau)]=\mathbb{E}_{\tau\sim p_\theta(\tau)}$。强化学习的最终目的是通过学习一个策略（$\theta$ 是参数）来最大化期望回报；

在现实应用中，处于某个状态的智能体无法得知未来的序列，从而无法根据目标函数无法指导下一个动作。为此，我们需要引入两个值函数来评估策略 π 在某个状态和动作下的期望回报：

(7) 状态值函数：$V^\pi(s) = \mathbb{E}[G_t|S_t = s]$，其中 $G_t = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}$。该函数表示的是智能体当前状态 S_t 为 s 时，执行策略 π 所得到的期望总回报。由于 $G_t = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1} = r_{t+1} + \gamma G_{t+1}$，故我们可以得到贝尔曼方程：

$$\begin{aligned} V^\pi(s) &= \mathbb{E}[(r_{t+1} + \gamma G_{t+1})|S_t = s] \qquad (2) \\ &= \mathbb{E}[(r_{t+1} + \gamma V^\pi(s'))|S_t = s] \qquad (3) \end{aligned}$$

(8) 状态-动作值函数：状态值函数只是提供了当前状态下的期望回报。如果我们在当前状态下执行了动作 a，即可得到该动作的期望回报：

$$Q^\pi(s,a) = [(r(s,a,s') + \gamma V^\pi(s'))|s,a] \qquad (4)$$

同样，我们可以得到 $Q^\pi(s,a) = \mathbb{E}_{s'}[r(s,a,s') + \gamma Q^\pi(s',a')|s,a]$

综上所述，为了获得最优策略，我们有三种思路：

(1) 直接优化策略 $\pi(a|s)$，使得智能体可以获取更高回报；

(2) 通过估计这两个值函数来引导智能体间接获得优化的策略；

(3) 融合上述两种做法，在优化策略的同时估计值函数；

2.2 值函数估计

值函数是对策略 π 的评估，求解最优策略相当于求解最优的值函数。基于值函数的策略学习方法主要有两种，分别是动态规划方式和蒙特卡洛方式。

2.2.1 动态规划

由于两种值函数均有贝尔曼方程的形式，故在得知状态转移概率 $p(s'|s,a)$ 和奖励 $r(s,a,s')$ 的前提下，我们可以基于贝尔曼方程迭代计算值函数。动态规划方法主要有两种算法，分别是策略迭代算法和值迭代算法。

策略迭代算法的目的是通过迭代计算值函数来使策略收敛到最优。具体来说，策略迭代算法分为两部分：首先是策略评估，算法基于当前策略、根据贝尔曼方程计算各个状态下的值函数，使得值函数收敛；然后是策略改进，基于当前的值函数更新、选择当前最佳策略。这两部分不断循环，直到策略 $\pi(a|s)$ 收敛，这就是最优策略。

值迭代算法与策略迭代算法不同，其直接使用贝尔曼最优方程、通过状态转移概率和奖励函数迭代计算最优值函数。获得最优值函数后，再通过状态-动作值函数来获取最优策略。相比于策略迭代算法，值迭代算法更加直接。

虽然这两种算法直观简便，但都要求模型已知，需要知道各个状态转移概率和奖励函数。此外，当状态数量较多时，算法效率较低。

2.2.2 蒙特卡洛

蒙特卡洛算法与蒙特卡洛积分概念类似，是一种基于采样的学习算法。对于一个策略 π，智能体从状态 s、执行动作 a 开始，通过随机游走的方式探索环境，计算得到总回报。重复 N 次实验后，即可得到该状态-动作值函数：

$$Q^{\pi}(s,a) \approx \frac{1}{N}\sum_{n=1}^{N} G(\tau_{s_0=s,a_0=a}) \tag{5}$$

根据状态-动作值函数，即可求出最优策略：

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q^{\pi}(s,a) \tag{6}$$

在如何获取下一步动作时，同样有不同的方式。显然，下一步动作的获取不能依靠当前的确定性策略 π，否则无法得到其他动作下的值函数，不利于改进策略。但完全采用随机的方式，会导致算法收敛速度过慢、不能更好测试当前策略是否有效。因此，我们一般使用 $\epsilon - greedy$ 策略：

$$\pi^{\epsilon}(s) = \begin{cases} \pi(s) = \underset{a}{\operatorname{argmax}}\, Q^{\pi}(s,a) & \text{if } p \in [0, 1-\epsilon] \\ \text{choose random action} & \text{if } p \in [1-\epsilon, 1] \end{cases} \tag{7}$$

相比于动态规划，蒙特卡洛法显然不需要知道模型相关参数，但采样精度对算法的准确性有着较大的影响。

2.2.3 Q-Learning 算法

时序差分学习方法结合了动态规划方法和蒙特卡洛方法，是一种强大的学习方法。对于蒙特卡洛方法而言，其模拟一段序列后、根据序列上各个状态的价值估计新的状态价值。而对于时序差分学习方法而言，其模拟一段序列，每行动几步，就根据新状态的价值估计执行前的状态价值。

Q-Learning 算法是一种时序差分学习方法，其更新公式如下所示：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(r_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{8}$$

类似于梯度下降，Q-Learning 没有直接更新值函数，而是使用 α 控制幅度，从而确保值函数最终收敛到最优值。此外，每一步动作的选择根据 $\epsilon - greedy$ 策略获取。综上所述，Q-Learning 算法流程如下：

(1) 初始化当前状态 S；

(2) 根据当前状态、采用 $\epsilon - greedy$ 策略获取下一步动作 A；

(3) 执行动作完毕后获取得知当前新状态 S' 和回报值 R，然后根据式8对 $Q(S, A)$ 进行更新；

(4) 令 $S = S'$，然后循环执行 (2) 和 (3)，直到到达终止状态；

(5) 初始化新的状态 S，重复执行 (1) 至 (5)，直到值函数 Q 收敛。

2.3 策略优化

与值函数估计不同，策略优化方法是一种基于策略函数的优化方法。其通过直接优化策略函数来获取最优策略。

在2.1中，我们得知了强化学习的目标函数 $\mathcal{F}(\theta)$。对策略参数 θ 求导，可以得到：

$$\frac{\partial\mathcal{F}(\theta)}{\partial\theta} = \frac{\partial}{\partial\theta}\int p_\theta(\tau)G(\tau)\mathrm{d}\tau \tag{9}$$

$$= \mathbb{E}_{\tau\sim p_\theta(\tau)}[\frac{\partial}{\partial\theta}\log p_\theta(\tau)G(\tau)] \tag{10}$$

$$= \mathbb{E}_{\tau\sim p_\theta(\tau)}[\sum_{t=0}^{T-1}\frac{\partial}{\partial\theta}\log\pi_\theta(a_t|s_t)\gamma^t G(\tau_{t:T}))] \tag{11}$$

显然，策略梯度只和策略函数以及总回报有关。因此，我们就可以用这个策略梯度不断更新策略函数参数。REINFORCE 算法是一种代表性算法，其根据当前策略生成一条轨迹 τ，然后计算相应的策略梯度。重复生成若干条轨迹后，取策略梯度的平均值对策略参数 θ 进行梯度下降，直到策略函数 π_θ 收敛。

当然，我们也可以结合策略梯度和时序差分学习方法进行强化学习，这就是演员-评论家算法。演员就是策略函数 $\pi_\theta(a|s)$，评论员就是值函数 $V(s)$。演员-评论家算法采用交替更新的方式，首先根据当前状态选取动作、获取即时奖励和新状态，然后分别根据值函数梯度和策略函数梯度对值函数参数、策略函数参数更新。不断迭代至 θ 收敛，此时的策略 π_θ 即为最优策略。

2.4 深度强化学习

2.4.1 DQN 算法

深度强化学习是一种新型的学习方法，其将深度学习的感知能力和强化学习的决策能力相结合，优势互补。深度强化学习的代表算法是 DQN 算法，其将神经网络应用到 Q-Learning 算法中，取得了良好的效果。

在2.2.3中，我们得知了 Q-Learning 算法的关键在于 $Q(s,a)$ 的迭代与更新。那么，我们完全可以通过深度学习来获取一个神经网络，从而拟合 $Q(s,a)$：

$$Q(s,a)\approx f(s,a,w) \tag{12}$$

其中，$f(s,a,w)$ 是一个近似函数，w 是函数的相关参数。在训练网络时，我们使用目标 Q 值 $r_{t+1}+\gamma\max_a A(S_{t+q},a)$ 作为标签样本训练网络，因此网络的目标值是：

$$y = \begin{cases} r & \text{if } s' \text{ is end.} \\ r+\gamma\max_{a'} f(s',a',w) & \text{if } s' \text{ is not end.}\end{cases} \tag{13}$$

的损失函数是：

$$L(w) = \mathbb{E}[(y-f(s,a,w))^2] \tag{14}$$

此外，DQN 算法还涉及到一个结构，称为经验池。因为在序列中样本具有连续性，如果每次得到样本后就更新 Q 值，会导致拟合函数受到样本分布的影响。因此，我们将样本数据先存储在经验池中，然后在训练网络时从经验池中随机采样数据进行训练，从而得到表现更好的网络。

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

图 1: DQN 算法伪代码

2.4.2 Nature DQN 算法

DQN 算法可以说是深度强化学习的开山鼻祖。在后续发展中，DQN 算法又衍生出了很多相关算法，其中 Nature DQN 是一种有着重大改进的 DQN 算法。在原始 DQN 算法中，我们使用同一个 Q 网络计算当前值和目标值，从而通过式14对 Q 网络进行更新。这会导致两个数值依赖性太强，不利于算法收敛。因此，在 Nature DQN 算法中，人们使用两个结构完全一致的网络进行训练，分别是训练网络和目标网络。训练网络 $f(s,a,w)$ 用来选择动作，并实时更新参数 w；而目标网络 $f'(s,a,w')$ 只是用来计算目标 Q 值，其参数 w' 不会实时更新，而是隔一定时间从训练网络同步参数过来。因此，这两个网络在训练过程中产生了差异性，有助于算法收敛。

以公式来表示的话，在 Nature DQN 算法中，目标值公式为：

$$y = \begin{cases} r & \text{if } s' \text{ is end.} \\ r + \gamma \max\limits_{a'} f'(s', a', w') & \text{if } s' \text{ is not end.} \end{cases} \tag{15}$$

那么，损失函数则为：

$$L(w) = \mathbb{E}[(y - f(s,a,w))^2] \tag{16}$$

2.4.3 Double DQN 算法

Double DQN 算法是基于 Nature DQN 算法的一种改进版本。在 DDQN 算法前，正如2.4.1中的式13和2.4.2中的式15，几乎所有类 Q-Learning 算法在计算目标 Q 值时，是通过贪婪法计算得到的，即直接选取各个动作中最大 Q 值来计算目标 Q 值。这会导致过度估计问题，即最终计算得到的算法模型会有较大偏差。

DDQN 算法为了解决这个问题，在计算目标 Q 值时将算法解耦成动作选择和动作评估两个步骤。首先，DDQN 网络在训练网络 f 得到最大 Q 值对应的动作：

$$a_{\max}(s', w) = \operatorname*{argmax}_{a'} f(s', a', w) \tag{17}$$

然后利用这个动作在目标网络 f' 中计算目标 Q 值：

$$y = r + \gamma f'(s', a_{\max}(s', w), w') \tag{18}$$

综合起来，在 DDQN 算法中，目标值公式为：

$$y = \begin{cases} r & \text{if } s' \text{ is end.} \\ r + \gamma f'(s', \underset{a'}{\operatorname{argmax}} f(s', a', w), w') & \text{if } s' \text{ is not end.} \end{cases} \tag{19}$$

损失函数为：

$$L(w) = \mathbb{E}[(y - f(s, a, w))^2] \tag{20}$$

除了目标值的计算以外，DDQN 算法与 Nature DQN 算法结构和流程一致。

2.5 强化学习在黑白棋上的应用

黑白棋是一种棋类游戏，通过相互翻转对方的棋子、最后以棋盘上谁的棋子多来判断胜负。

对于强化学习而言，状态 s 就是当前棋面局势。由于棋局大小为 8×8，每个位置有黑棋、白棋和无棋三种情况，故我们使用 one-hot 编码表示这三种情况，白棋为 $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$，空子为 $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$，黑棋为 $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$。因此，状态 s 是一个 3×64 的矩阵表示。在忽略棋盘上所有棋子的前提下，智能体有 64 个位置可以落子，还有一个不会落子的情况。因此，动作 a 也可以表示为一个 65 维的向量，当 $a[i] = 1$ 且 $i \in [0, 63]$ 时，智能体会在对应位置落子；当 $a[64] = 1$ 时，智能体不会落子。

强化学习的另一个重要参数是反馈函数 $r(s, a, s')$。理论上，按照与上次五子棋类似的思想，反馈函数可以使用黑白棋的评价函数构建，即 $r(s, a, s') = g(s')$，其中 $g(s')$ 就是一个棋局状态评价函数。然而，由于对黑白棋了解不深，加上在网上并没有找到很好的评价函数实现，故我们只是设置了一个简单的反馈函数，只有当 s' 为终局时 $r(s, a, s')$ 的值才为非零，其余情况下 $r(s, a, s') = 0$。

此外，对于当前状态 s，当智能体执行动作 a 后，棋盘必将达到另一种状态，因此 $p(s_{t+1}|s_t, a)$ 要么等于 1，要么等于 0。因此，在确定了强化学习的各个基本参数后，我们就可以采用上述介绍的强化学习算法构建一个黑白棋智能体。在经过一些尝试后，我们决定采用 DDQN 算法，这是因为该算法原理清晰、实验效果好。此外，该算法使用深度学习的相关内容，~~最得高大上的同时~~可以大大提高算法的准确率。

在普通的强化学习中，只有一个智能体在行动，通过连续的动作使得环境状态发生改变，从而得到最大期望回报。然而，与普通的强化学习不同，黑白棋是一个双方进行博弈的过程，各自的最终目标完全相反。此外，两者之间的动作是不连续的，需要轮流落子。因此，我们使用了 Adversarial-DDQN 模型，构建两个 DDQN 模型分别充当黑棋方和白棋方，在对抗的过程中进行训练。这两个 DDQN 模型的结构完全一致，各自的经验池只记录与自己的当前状态和动作有关的数据。

在本次实验中，我们构建了一个三层神经网络，输入 3×64 的状态张量 S，输出 1×65 的动作-状态值张量。根据相应动作 A 的索引，我们就可以从动作-状态值张量中提取其预测值 $Q(S, A)$。另外，本次实验中，我们还使用 DDQN 模型，那么对于单一模型而言，其需要两个完全相同的网络，分别是目标网络和训练网络。训练网络保存着上一次迭代后的网络权重，不会发生变化，而训练网络实时更新。我们在进行训练时更新的是训练网络的权重。每隔一定的迭代次数后，训练网络才会将其权重同步至目标网络。

由于采用了对抗型的网络，因此不同于普通 DDQN 算法的目标 Q 值计算公式19和损失函数公式20，在训练过程中，损失函数定义如下：

$$L_W(w_W) = \mathbb{E}[(y_W - f_W(s, a, w_W))^2] \tag{21}$$

$$L_B(w_B) = \mathbb{E}[(y_B - f_B(s, a, w_B))^2] \tag{22}$$

其中目标值为：

$$y_W = \begin{cases} r & \text{if } s' \text{ is end.} \\ r - \gamma f'_B(s', \underset{a'}{\text{argmax}}\, f_B(s', a', w_B), w'_B) & \text{if } s' \text{ is not end.} \end{cases} \tag{23}$$

$$y_B = \begin{cases} r & \text{if } s' \text{ is end.} \\ r - \gamma f'_W(s', \underset{a'}{\text{argmax}}\, f_W(s', a', w_W), w'_W) & \text{if } s' \text{ is not end.} \end{cases} \tag{24}$$

$L_W(w_W)$ 和 $L_B(w_B)$ 分别是白棋方训练网络和黑棋方训练网络的损失函数；$f_W(s, a, w_W)$ 和 $f_W(s, a, w_B)$ 分别是白棋方训练网络和黑棋方训练网络；$f'_W(s, a, w'_W)$ 和 $f'_W(s, a, w'_B)$ 分别是白棋方目标网络和黑棋方目标网络；s 是当前方执行动作前的状态，a 是执行的动作，而当执行完毕后，就变成了对手执行动作前的状态 s'，以及对手会执行的动作 a'。我们将对应的状态和动作输入网络，就可以得到对应的预测值。我们设置在己方终局胜利时奖励值为 50，失败时奖励值为-50，其余情况下奖励值均为 0。由于双方的目标都是最大化总回报，故我们使用 $r - \gamma f'(s', \underset{a'}{\text{argmax}}\, f(s', a', w), w')$ 而不是 $r + \gamma f'(s', \underset{a'}{\text{argmax}}\, f(s', a', w), w')$ 作为网络的拟合目标。

这样，我们就构成了两个可以互相对抗的 DDQN 网络，在训练过程中采用 $\epsilon - greedy$ 策略获取动作执行，在真正对弈的时候直接使用 $greedy$ 策略获取动作执行。

3 伪代码/流程图

3.1 对抗训练流程

两个 DDQN 网络对抗训练的流程如图2所示。

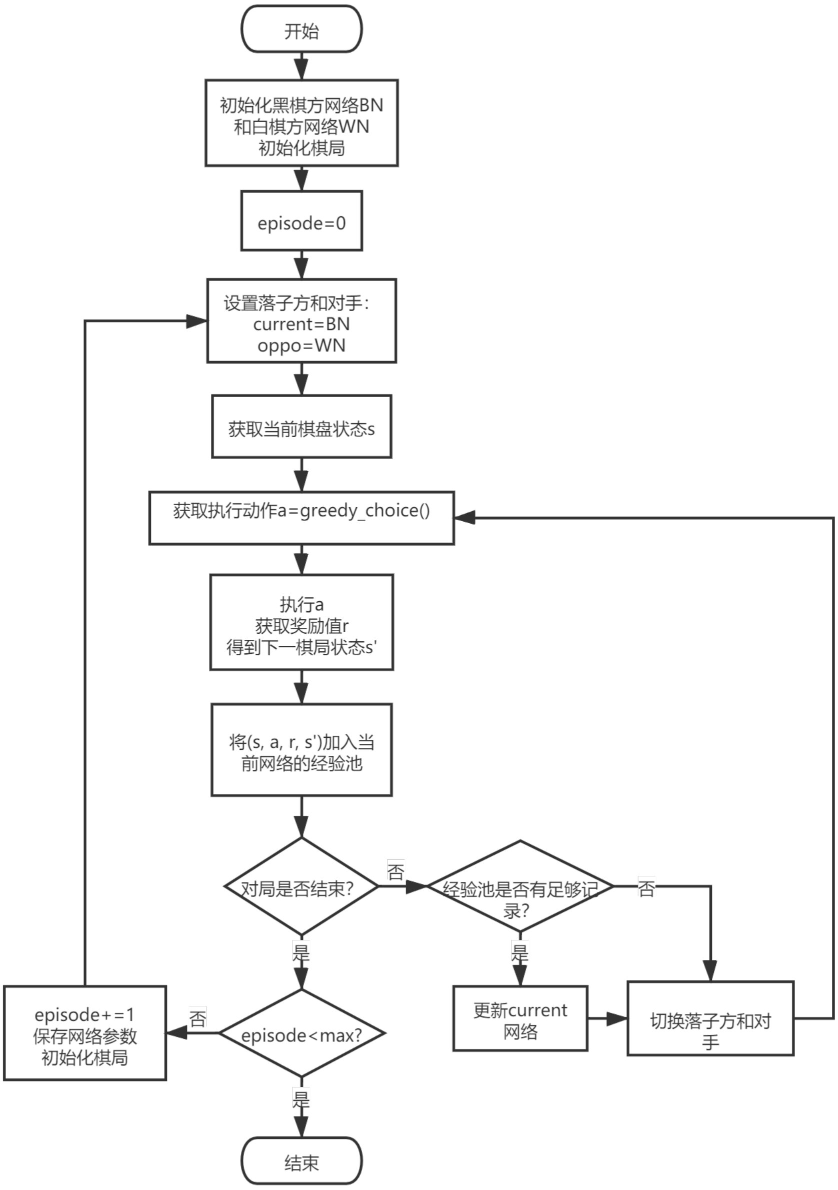

图 2: 对抗训练流程图

首先，我们构建两个 DDQN 模型充当黑棋方和白棋方，每个 DDQN 模型都有一个训练网络和目标网络。对抗训练过程与图1的伪代码类似，每个 episode 完成一局对弈。两个智能体轮流下棋，获取棋盘状态后根据 $\epsilon - greedy$ 策略获取下一步执行动作，具体流程见3.2。执行完毕后，将相应信息存入经验池，并对网络进行更新，更新流程见3.3。通过不断迭代，即可训练网络。

3.2 greedy_choice() 流程

$\epsilon - greedy$ 策略实现的流程图如图3所示。

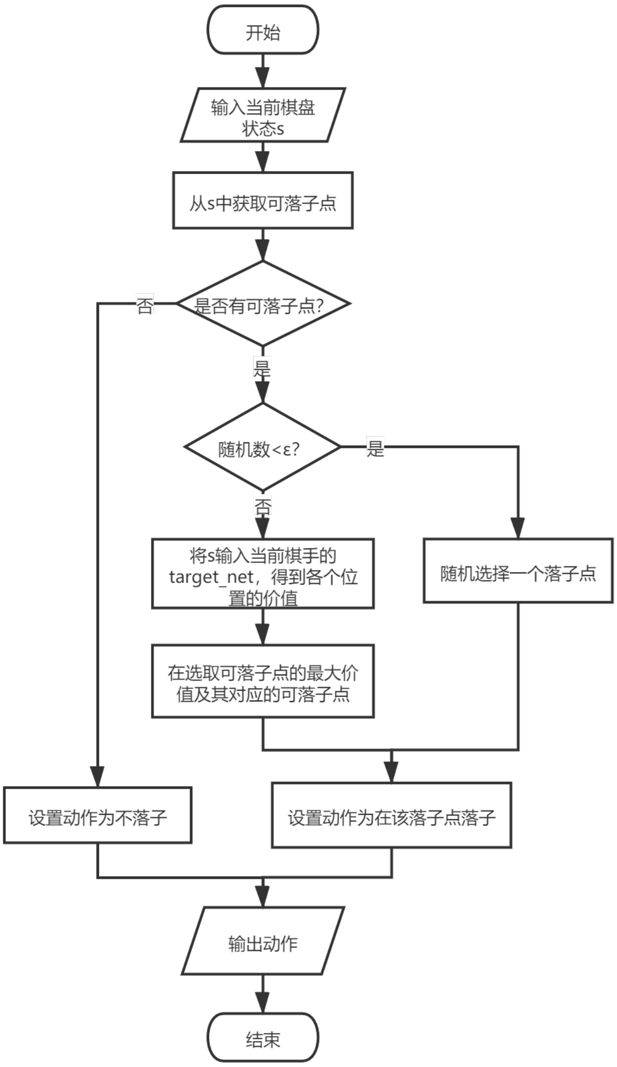

图 3: greedy_choice() 流程图

在训练网络时，我们设置 ϵ 为一个较小的值，保证智能体的探索具有一定的随机性。而在正式对弈时，我们设置 $\epsilon = 0$，这样就变成一个贪心算法流程，智能体每次都依据最优值选择动作。

3.3 网络更新流程

网络更新流程图如图4所示。

3.4 人机对弈流程

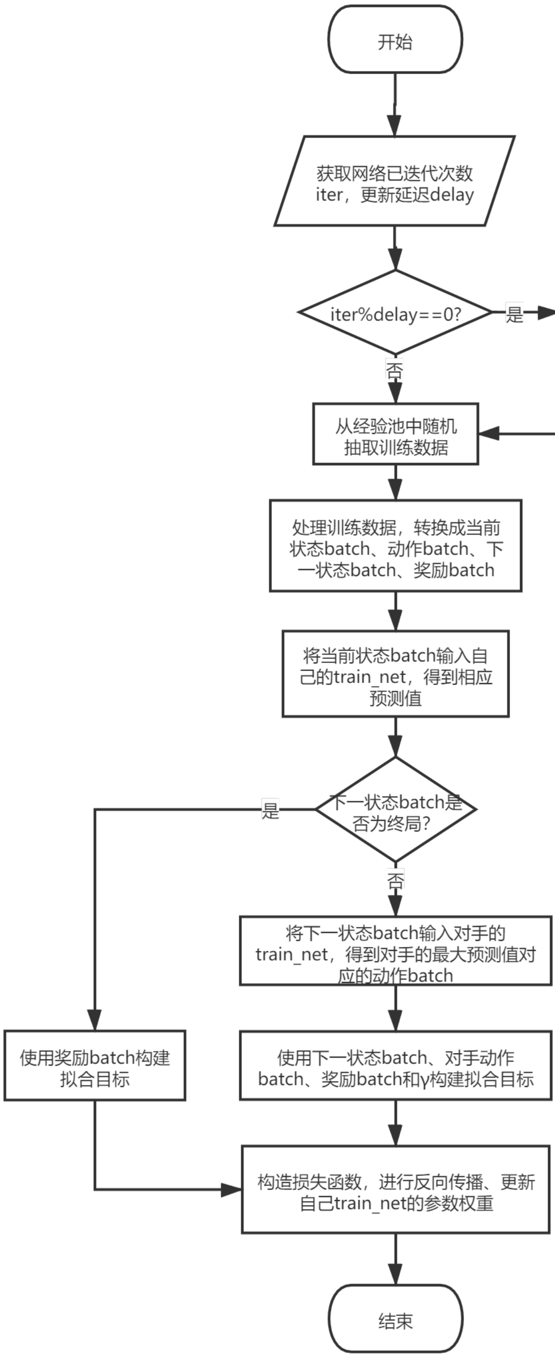

图 4: 网络更新流程图

首先，我们根据迭代次数决定目标网络和训练网络是否需要同步。然后我们从经验池中随机抽取一定量的数据作为训练 batch，得到相应的损失函数。最后，我们通过反向传播更新当前模型的训练网络，从而完成一次更新。

人机对弈流程图如图5所示。与3.1流程类似，不过从智能体之间的对弈变成玩家与智能体之间的对弈。此外，智能体不再使用 $\epsilon - greedy$ 策略进行探索，而是直接使用贪心策略选取最优值进行落子。

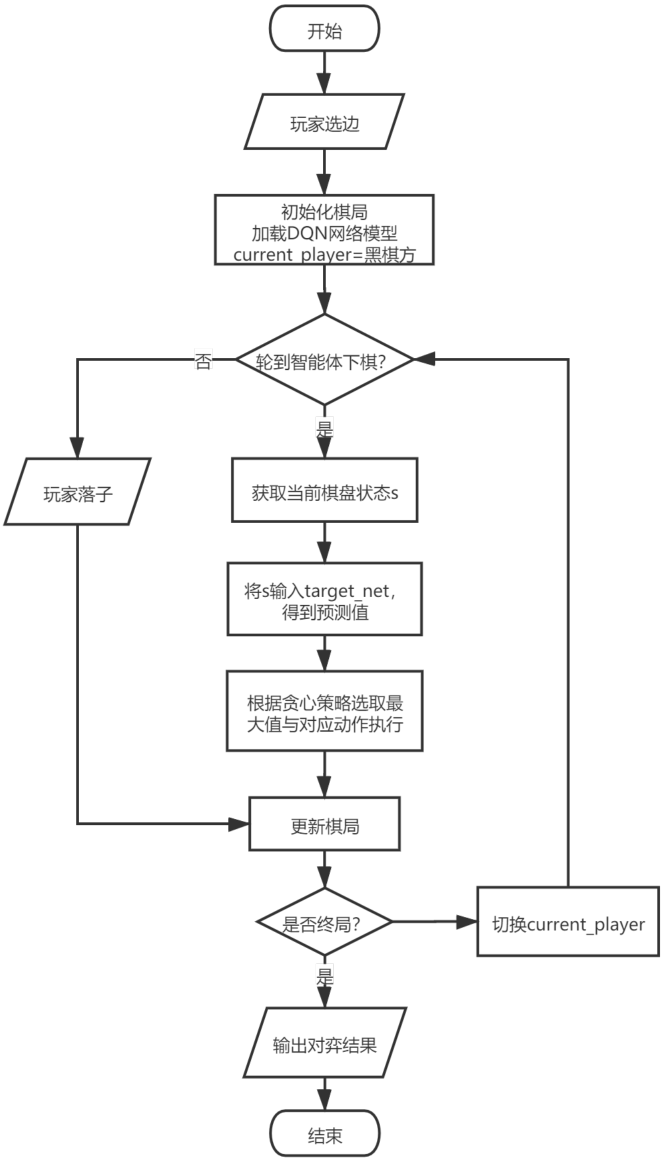

图 5: 人机对弈流程图

4 关键代码

4.1 构建神经网络结构

网络构建代码如下所示。我们搭建了一个三层神经网络，包括两层卷积层和一个全连接层。两个智能体的目标网络和训练网络结构和参数均保持一致。

```
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv1d(3, 8, 3, 1, 1)
        self.conv2 = nn.Conv1d(8, 12, 3, 1, 1)
        self.linear = nn.Linear(12 * STATE_COUNT, (STATE_COUNT + 1))

    def forward(self, x):
```

```python
        conv1_out = F.relu(self.conv1(x))
        conv2_out = F.relu(self.conv2(conv1_out))
        conv2_out = conv2_out.view(conv2_out.shape[0], -1)
        output = self.linear(conv2_out)
        return output
```

4.2 构建 Double DQN 模型

我们将 Double DQN 模型的相关参数和方法集成为一个 *Double_DQN* 类，初始化参数如下所示：

```python
class Double_DQN:
    def __init__(self, player):
        self.pool = np.zeros((POOL_SIZE, RECORD_LENGTH))      # 经验池
        self.record_counter = 0                               # 经验池计数
        self.iteration_counter = 0                            # 迭代计数
        self.train_Q, self.target_Q=DQN().to(device), DQN().to(device)#训练网络, 目标网络
        self.player = player                                  # 所属棋方
        self.optimizer = torch.optim.Adam(self.train_Q.parameters(), lr=LR)
        self.criteria = nn.MSELoss()
```

$\epsilon - greedy$ 策略的实现代码如下所示，我们首先获取一个随机数，然后进行选择：

```python
    def greedy_choice(self, othello):
        """
        epsilon-greedy选择函数
        """
        # 获取可落子的位置
        possible_moves = othello.get_possible_moves(self.player)
        possible_moves = list(possible_moves)

        if len(possible_moves) == 0:
            return (0, 64)                    # 表示此时没有动作

        # epsilon贪心策略
        if np.random.uniform() < EPSILON:
            # 从possible_moves里面随机选一个坐标
            idx = np.random.randint(0, len(possible_moves), 1)[0]
            pos = possible_moves[idx]
        else:
            # 根据当前最优策略选择坐标
            s = board_to_onehot(othello.board).view(1, 3, -1).to(device)
            # 通过训练网络获取预测值
            action_values = self.train_Q(s)[0]
            moves_idx = [pos[0] * BOARD_SIZE + pos[1] for pos in possible_moves]
            moves_values = action_values[moves_idx]
            # 选取最大价值和对应坐标
            _, idx = torch.max(moves_values, 0)
            # 获取动作
```

```
            move = moves_idx[idx]
            pos = (int(move // BOARD_SIZE), int(move % BOARD_SIZE))
        return pos
```

更新经验池的方法如下所示，我们根据计数器不断增加、替换经验池中的训练数据：

```
    def expand_pool(self, s, a, r, s_, is_end):
        """
        更新经验池
        """
        record = np.hstack((s.flatten(), a[0] * BOARD_SIZE + a[1], r, s_.flatten(), is_end
            ))
        self.pool[self.record_counter % POOL_SIZE] = record
        self.record_counter += 1
```

最后，更新网络的代码如下所示。我们根据需要将训练网络与目标网络进行同步，然后从经验值中获取随机数据，对网络进行训练：

```
    def update_network(self, oppo_train_Q, oppo_target_Q):
        """
        更新网络
        """
        self.iteration_counter += 1
        if self.iteration_counter % UPDATE_DELAY == 0:
            # 加载训练网络至目标网络
            self.target_Q.load_state_dict(self.train_Q.state_dict())
        # 从经验池中获取一个batch
        random_indices = np.random.choice(POOL_SIZE, BATCH_SIZE)
        record_batch = self.pool[random_indices, :]

        # 棋面/状态batch
        s_batch = torch.tensor(record_batch[:, :STATE_COUNT], dtype=torch.float).to(device
            )
        s_batch = batch_to_onehot(s_batch).to(device)
        # 动作batch
        a_batch = torch.tensor(record_batch[:, STATE_COUNT:STATE_COUNT + 1], dtype=torch.
            int64).to(device)
        # 奖励batch
        r_batch = torch.tensor(record_batch[:, STATE_COUNT + 1:STATE_COUNT + 2], dtype=
            torch.float).to(device)
        # 下一棋面/状态/对手batch
        oppo_s_batch = torch.tensor(record_batch[:, STATE_COUNT + 2:STATE_COUNT * 2 + 2],
            dtype=torch.float).to(device)
        oppo_s_batch = batch_to_onehot(oppo_s_batch).to(device)
        # 是否结束batch
        is_end_batch = record_batch[:, STATE_COUNT * 2 + 2]

        batch_prediction = self.train_Q(s_batch).gather(1, a_batch)
        # 得到对手train_Q中在s'处值最大的动作
```

```
        oppo_batch_moves = torch.max(oppo_train_Q(oppo_s_batch).detach(), 1)[1].view(BATCH_SIZE, 1)
        # 然后将对手target_move中该动作的值作为Q值
        batch_target = r_batch - GAMMA * oppo_target_Q(oppo_s_batch).detach().gather(1, oppo_batch_moves)

        for idx in range(BATCH_SIZE):
            if is_end_batch[idx] == 1:
                batch_target[idx] = r_batch[idx]

        self.optimizer.zero_grad()
        loss = self.criteria(batch_prediction, batch_target)
        loss.backward()          # 反向传播
        self.optimizer.step()    # 更新权重
```

4.3 对抗训练的实现

在构建好 DDQN 模型后，我们就可以生成两个智能体，进行对抗学习。对抗训练的实现代码如下所示：

```
if __name__ == '__main__':
    AI_first_network, AI_second_network = Double_DQN(BLACK), Double_DQN(WHITE)
    epoch_start = 0
    AI_first_network.train_Q.load_state_dict(torch.load('......'))
    AI_second_network.train_Q.load_state_dict(torch.load('......'))

    for episode in range(epoch_start, EPISODE):
        othello = Othello(BOARD_SIZE, WHITE, BLACK)
        counter = 0
        if episode % 10 == 0:
            print(episode)
        while True:
            # 先手
            s = othello.board    # 获取棋面状态
            a_first = AI_first_network.greedy_choice(othello)# 获取执行动作
            othello.add_chess(a_first, BLACK)        # 执行动作
            r = othello.game_over() * 50.0 * BLACK# 获取奖励值
            s_ = othello.board                       # 获取下一状态
            is_end = 1 if abs(r) > 0 else 0          # 判断是否为终局
            AI_first_network.expand_pool(s, a_first, r, s_, is_end)# 将数据存入经验池
            counter += 1
            if is_end:
                break
            if AI_first_network.record_counter >= BATCH_SIZE:
                # 更新网络
                AI_first_network.update_network(AI_second_network.target_Q)
            # 后手
            ......
```

```
29        # 每100个episode保存一次模型
30        if (episode + 1) % 200 == 0:
31            print('当前episode: ', episode)
32            torch.save(AI_first_network.train_Q.state_dict(), ......)
33            torch.save(AI_second_network.train_Q.state_dict(), ......)
```

在第 2 至 5 行中，我们构建了两个 DDQN 模型，并加载相应的网络参数。然后我们进行 *EPISODE* 轮对弈，每一轮对弈都是黑棋方先手，两个智能体轮流下棋、将数据存入各自的经验池进行学习。每隔若干轮次对弈后，将网络参数进行保存，便于后续训练。

正式游戏的流程与上述训练相似，不同的是智能体不需要将数据保存到经验池进行学习、更新网络，并且第 15 行中的动作获取采用贪心策略。由于代码类似，故不在这里贴出正式游戏的详细代码，具体代码可见附件 start_here.py。

5 实验过程与结果分析

正如2.5和3所述，我们使用了对抗式 Double DQN 算法实现黑白棋的深度强化学习。我们设置 $\gamma = 0.9, \epsilon = 0.1$，梯度大小为 0.001，batch 的大小为 32，经验池的大小为 200，每隔 10 次迭代训练网络和目标网络同步一次。经过简单测试，我们发现黑棋先手模型中训练 22000 个 episode 的效果较好，白棋后手模型中训练 30000 个 episode 的效果较好。因此我们采用这两个参数作为模型参数。

为了检验智能体的下棋水平，我们在网上找了另一个黑白棋程序进行对弈。我方智能体黑棋先手，对方程序白棋后手。在进行第一步落子时，棋局状态如图6所示，智能体有四个位置可下。查看后端，发现神经网络给出这四个位置的 Q 值按序号分别为 $-0.6509, -1.0879, -0.9966, -0.8114$。理论上，由于相互对称，这四个位置的预测值应该一致，但可能由于长期训练中悟到了什么东西，导致智能体认为最上方的位置最优，故在此落子。

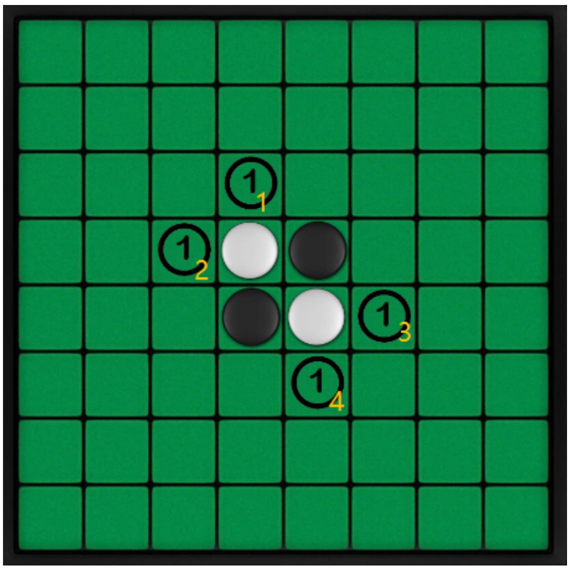

图 6: 棋局初始状态，我方智能体可下的四个位置

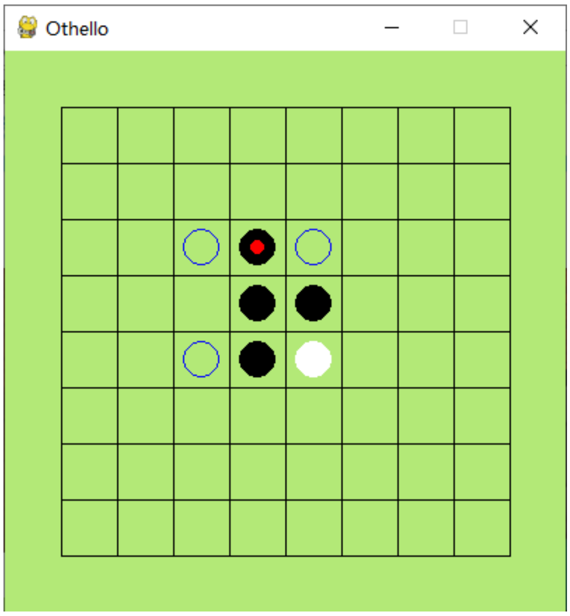

图 7: 第 1 手：我方智能体落子后的状态

如图8所示，对方程序选择在 [4 2] 处落子，轮到我方程序时有五个位置可以落子。此时，网络对这五个位置的预测 Q 值按序号分别为 $-3.0287, -2.3897, -1.2278, -0.9966, -0.7451$。因此，我方智能体选

择在 [5, 5] 处落子。我们认为，可能这是因为如果在靠左的地方落子，会受到白子的夹击，而落在最右处既可以防止被夹击，也可以更快占据优势更大的角落，因此网络基于最右处的位置最高值。

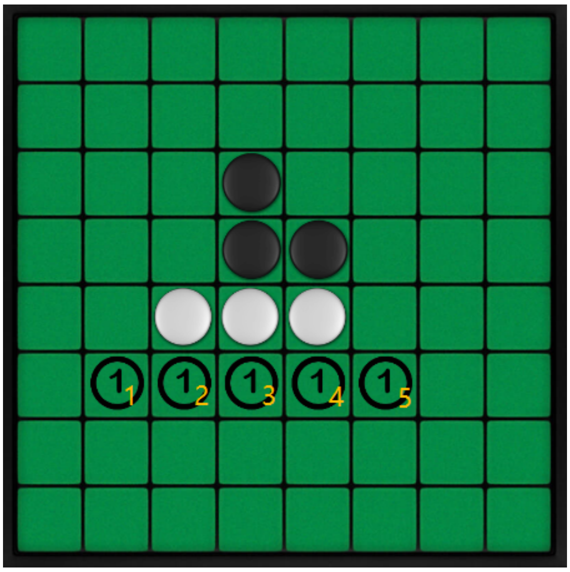

图 8: 第 2 手：对方程序落子后的状态

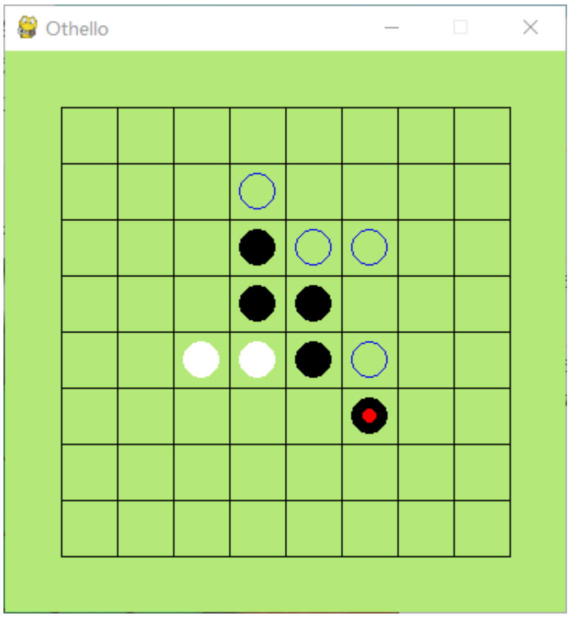

图 9: 第 3 手：我方智能体落子后的状态

由于在开局阶段局势不明显，无法有效解释我们智能体的决策依据，故我们直接快进到棋盘中局进行分析。如图10所示，对方程序落子后，我方智能体有 11 个位置可落子，对此网络的预测 Q 值按序号分别为 $-6.3670, -1.5513, -3.2760, -2.0857, -2.3666, -1.9265, -2.1887, -1.6364, -9.6355, -4.5839, -10.5698$。因此，我方智能体选择在序号为 2 的地方落子，如图11所示。这 11 个位置中，序号为 11 的位置得分最低，这是因为在此落子的话我们智能体得到的收益最少（只能翻 1 个子），而且对于后续的局势没有帮助，一旦对手程序在 [7, 3] 处落子，就会将竖排的三个黑子转成白子，从而围困住左下角，因此黑棋继续在左下棋面发力的意义不大。相比之下，在上边落子不仅可以取得较高的收益，而且处于边上的棋子难以被翻动，故网络给上边的位置较高的评价。然而，为什么序号为 1 的位置得分第二低呢？我们猜测可能是因为虽然在这个位置落子可以暂时得到较高的收益（翻五个子），但接下来白棋可以在 [0, 3] 处落子，在取得收益的同时获得三个边子，取得更大的优势。相比之下，序号为 2 的位置不仅可以得到较高的收益，而且可以避免对方白棋在棋面上边有较多作为，故给予该位置最高预测值。

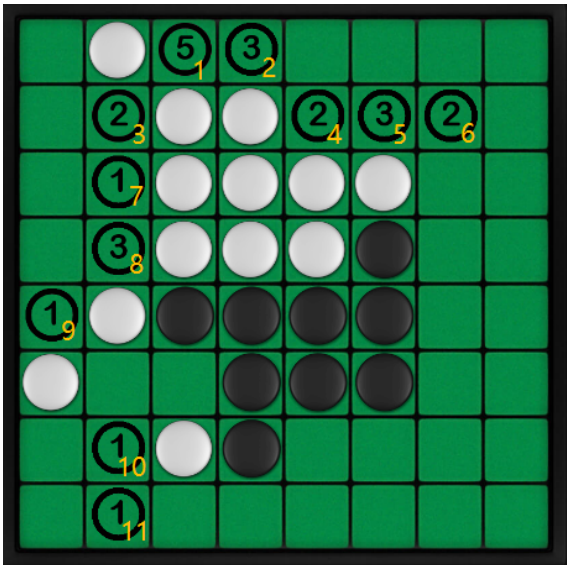

图 10: 对方程序落子后的状态

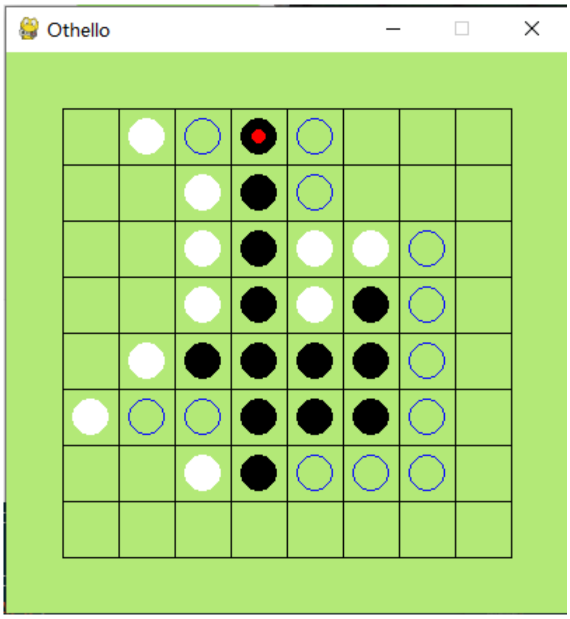

图 11: 我方智能体落子后的状态

继续对弈，此时对方程序落子后的棋面状态如图12所示，我方智能体有 10 个位置可落子，网络对这十个位置的预测 Q 值按序号分别为 $-6.3670, -2.7734, -10.5344, -2.1887, -9.6355, -2.3897, -4.6352, -4.5839, -7.4419, -10.5698$，因此智能体在序号 4 的位置落子，如图13所示。序号 3 和序号 10 的位置得分最低。对于前者而言，我们猜测这是因为在此落子后，白棋可以在 $[2, 0]$ 继续落子，从而在左边占据优势；而对于后者而言，我们也不知道为什么分数这么低……序号为 2 的位置很不错，但我方智能体似乎认为序号 4 的位置更好。

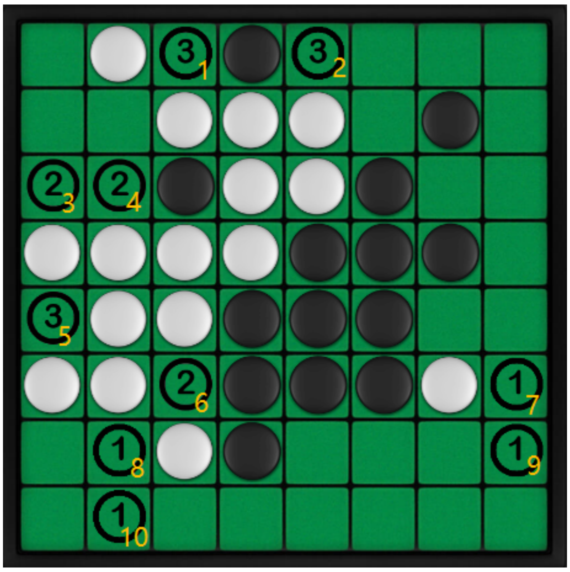

图 12: 对方程序落子后的状态

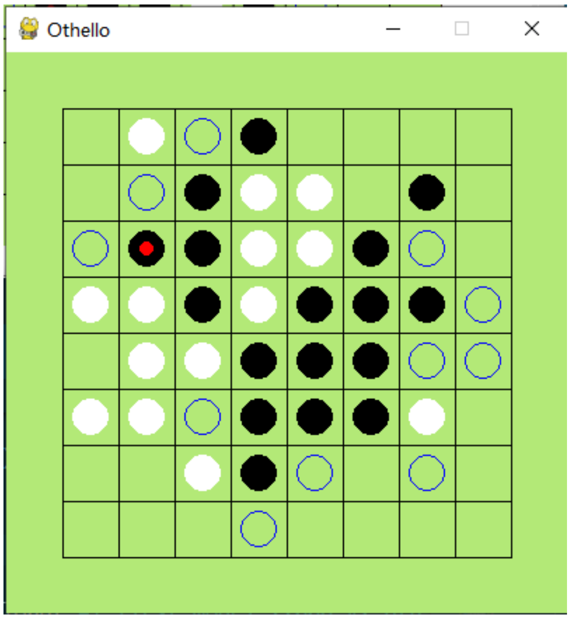

图 13: 我方智能体落子后的状态

接下来的另一个对弈局面如图14和15所示。我方智能体对图14十个位置的网络预测值按序号分别为 $-10.5291, -4.7170, -4.2262, -9.7096, -10.5344, -7.8202, -7.4419, -10.0850, -6.8865, -6.7812$。我们觉得序号 5 的位置非常理想，但我方智能体却给予了最低评分。序号 3 的位置被我方智能体给予了最高预测值，故将黑子落在此处。

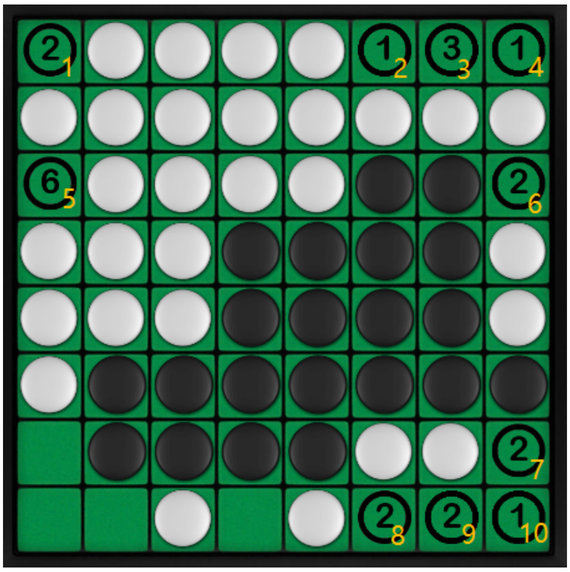

图 14: 对方程序落子后的状态

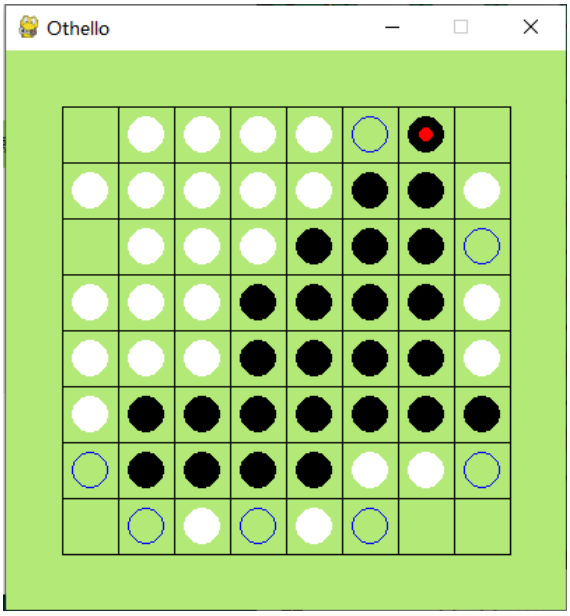

图 15: 我方智能体落子后的状态

图16是官子阶段，棋局即将结束。此时我方智能体有三个位置可以落子。网络对这三个位置预测值分别为 $-10.5291, -10.5344, -10.5798$。其实三个位置无论先下哪个意义都不大，因为对方程序都无子可下，我方智能体都会获得胜利。但序号 1 的位置是角落位置，本身具有较高战略价值，而序号 2 的位置可以得到非常高的收益（翻 9 个子），故这两个位置的分数都要高于序号 3 的位置。最终，我方智能体按照序号 1、2、3 的顺序连续落子，取得了胜利。

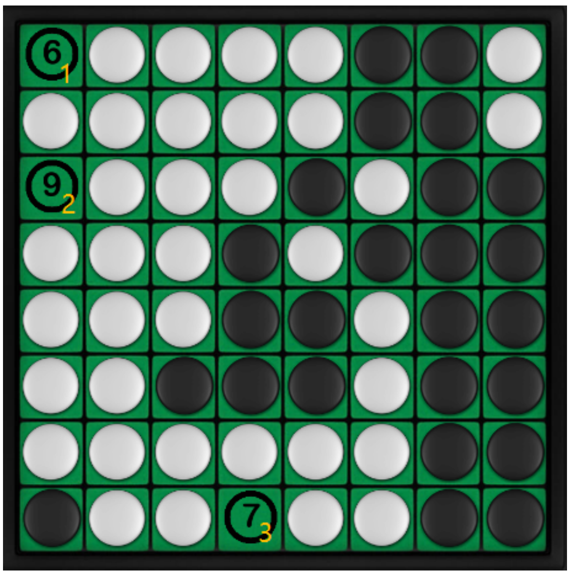

图 16: 对方程序落子后的状态

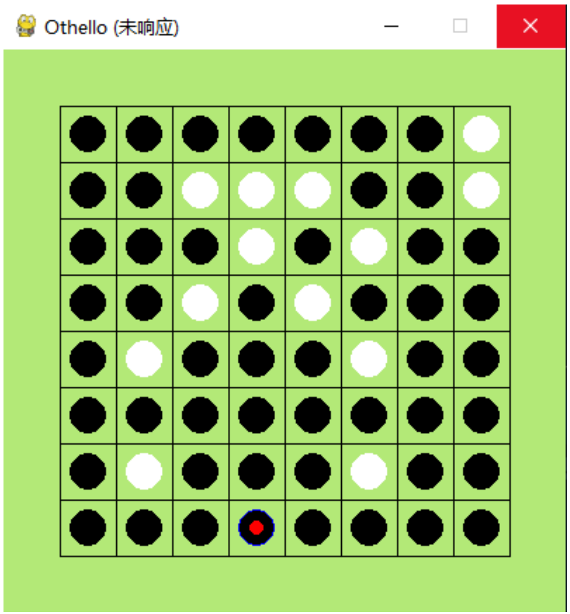

图 17: 我方智能体连下三子，取得胜利

由于时间有限，我们只具体分析了一局黑白棋中我方智能体的表现。在分析过程中，由于 DDQN 用到了神经网络，可解释性较差，加上我们对黑白棋了解不深，故只能进行一些简单的解释与分析，分析也不一定准确。

在后续的实验中，我们使用我们的 DDQN 模型与网上的其他黑白棋程序进行对弈。当对方程序难度为简单或中等时，我方智能体基本都能取得胜利，而难度为困难时，则会互有胜负。此外，我们也遇到了一些棋力十分高强的黑白棋程序，即使是简单模式也无法取得胜利。我们猜测可能是因为里面使用

了优秀的评价函数配合博弈树搜索，从而取得良好的效果。然而，本次实验中我们只能使用强化学习的内容，故没有在这方面进行相关尝试。

此外，我们也发现后手模型似乎比先手模型水平更强。这可能是因为先手模型一开始面对的是对称局面，在初期落子阶段表现不佳，而后手模型可以根据先手的落子实时调整学习，故水平更优秀。

6 创新点

(1) 使用了更合理、有效的方式对棋面状态和动作进行 one-hot 编码；

(1) 使用了深度强化学习，在 DQN 算法和 Nature DQN 算法的基础上实现了 Double DQN 模型；

(2) 构建了对抗模型，通过使用两个 Double DQN 网络模拟对弈，从而进行学习迭代；

7 实验总结与感想

本次实验是人工智能实验课程的最后一个实验，要求我们使用强化学习的知识构建一个黑白棋智能体。经过讨论与尝试，我们使用对抗式 Double DQN 构建黑白棋对弈模型。在经过训练后，我们尝试用自己的模型与网上的一些黑白棋程序进行对弈，取得了不错的成果，也从侧面证明了算法的有效性。不过，在实验过程中，我们也发现了模型的一些缺陷，例如可解释性差、训练次数会严重影响模型的表现、棋力依旧难以和顶级黑白棋程序抗衡等。此外，两个网络模型理论上可以通过某些转换变成同一个模型，从而只需要训练一个模型即可。这些都是我们的后续改进方向。

与往常实验不同，本次实验我们还需要和其他小组的黑白棋程序进行对弈比赛。希望我们的模型能在比赛中取得一定的成绩。