

人工智能lab12 实验报告

CCY TRY

一、实验原理

1.1 背景介绍

监督学习 (*Supervised learning*) :

监督学习即具有特征 (feature) 和标签 (label) 的，即使数据是没有标签的，也可以通过学习特征和标签之间的关系，判断出标签—分类。简言之：提供数据，预测标签。比如对动物猫和狗的图片进行预测，预测label为cat或者dog。

实现方法：通过已有的一部分输入数据与输出数据之间的对应关系（训练集），生成一个函数，将输入映射到合适的输出（测试集），例如分类。

无监督学习 (*Unsupervised learning*) :

无监督学习即只有特征，没有标签，通过数据之间的内在联系和相似性将他们分成若干类—聚类。根据数据本身的特性，从数据中根据某种度量学习出一些特性。例如，一个人没有见过恐龙和鲨鱼，如果给他看了大量的恐龙和鲨鱼，虽然他没有恐龙和鲨鱼的概念，但是他能够观察出每个物种的共性和两个物种间的区别的，并对这两种动物予以区分。简言之：给出数据，寻找隐藏的关系。

在期中之前的实验作业中，我们着重实现的是有监督学习：即提供一系列数据和它对应的标签，通过不断学习他们之间存在的潜在关联而对新数据做出预测。但是有监督学习也存在缺陷，例如在训练自动下棋的模型时，它的思路是将当前棋盘的状态作为输入数据，其对应的最佳落子位置 (action) 作为标签。但是棋盘状态千变万化，我们很难做到每一种状态下的最佳action是确定的，即使能做到，数据量也非常庞大。所以在解决这类问题时，我们就提出了强化学习这一概念。

强化学习 (*Reinforcement learning*) :

强化学习与半监督学习类似，均使用未标记的数据，但是强化学习通过算法学习是否距离目标越来越近，我理解为激励与惩罚函数。简言之：通过不断激励与惩罚，达到最终目的。

区别：

- (1) 监督学习有反馈，无监督学习无反馈，强化学习是执行多步之后才反馈。
- (2) 强化学习的目标与监督学习的目标不一样，即强化学习看重的是行为序列下的长期收益，而监督学习往往关注的是和标签或已知输出的误差。

(3) 强化学习的奖惩概念是没有正确或错误之分的，而监督学习标签就是正确的，并且强化学习是一个学习+决策的过程，有和环境交互的能力（交互的结果以惩罚的形式返回），而监督学习不具备。

1.2 强化学习的基本模型

简单来说，强化学习就是一个智能体（Agent）采取行动（Action）从而改变自己的状态（State）获得奖励（Reward）与环境（Environment）发生交互的循环过程。具体包含如下八个因素：

第一个是环境的状态 S ， t 时刻环境的状态 S_t 是它的环境状态集中的某一个状态。

第二个是个体的动作 A ， t 时刻个体采取的动作 A_t 是它的动作集中的某一个动作。

第三个是环境的奖励 R ， t 时刻个体在状态 S_t 采取的动作 A_t 对应的奖励 R_{t+1} 会在 $t+1$ 时刻得到。

下面是稍复杂一些的模型要素。

第四个是个体的策略(policy) π ，它代表个体采取动作的依据，即个体会依据策略 π 来选择动作。最常见的策略表达方式是一个条件概率分布 $\pi(a|s)$ ，即在状态 s 时采取动作 a 的概率。即 $\pi(a|s) = P(A_t = a | S_t = s)$ 。此时概率大的动作被个体选择的概率较高。

第五个是个体在策略 π 和状态 s 时，采取行动后的价值（value），一般用 $v_\pi(s)$ 表示。这个价值一般是一个期望函数。虽然当前动作会给一个延时奖励 R_{t+1} ，但是光看这个延时奖励是不行的，因为当前的延时奖励高，不代表到了 $t+1, t+2, \dots$ 时刻的后续奖励也高。比如下象棋，我们可以某个动作可以吃掉对方的车，这个延时奖励是很高，但是接着后面我们输棋了。此时吃车的动作奖励值高但是价值并不高。因此我们的价值要综合考虑当前的延时奖励和后续的延时奖励。价值函数 $v_\pi(s)$ 一般可以表示为下式，不同的算法会有对应的一些价值函数变种，但思路相同：

$$v_\pi(s) = E_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s) \quad (1)$$

其中， γ 是第六个模型要素，即奖励衰减因子，在 $[0, 1]$ 之间。如果为0，则是贪婪法，即价值只由当前延时奖励决定，如果是1，则所有的后续状态奖励和当前奖励一视同仁。大多数时候， γ 会取一个0到1之间的数字，即当前延时奖励的权重比后续奖励的权重大。

第七个是环境的状态转化模型，可以理解为一个概率状态机，它可以表示为一个概率模型，即在状态 s 下采取动作 a ，转到下一个状态 s' 的概率，表示为 $P_{ss'}^a$ 。

第八个是探索率 ϵ ，这个比率主要用在强化学习训练迭代过程中，由于我们一般会选择使当前轮迭代价值最大的动作，但是这会导致一些较好的但我们没有执行过的动作被错过。因此我们在训练选择最优动作时，会有一定的概率 ϵ 不选择使当前轮迭代价值最大的动作，而选择其他的动作。

1.3 Q – Learning

Q -Learning是强化学习算法中value-based的算法， Q 即为 $Q(s, a)$ ，就是在某一时刻的 s 状态下($s \in S$)，采取动作 a ($a \in A$)动作能够获得收益的期望，环境会根据agent的动作反馈相应的回报reward，所以算法的主要思想就是将State与Action构建一张Q-table和R-table来更新Q值，然后根据 Q 值来选取能够获得最大收益动作。

两个表记录的都是各个状态下采取各个动作的得分：

Q-Table	a1	a2
s1	q(s1,a1)	q(s1,a2)
s2	q(s2,a1)	q(s2,a2)
s3	q(s3,a1)	q(s3,a2)

R-Table	a1	a2
s1	r(s1,a1)	r(s1,a2)
s2	r(s2,a1)	r(s2,a2)
s3	r(s3,a1)	r(s3,a2)

注意：R-table是固定不变的，是先验知识；Q-table是要不断学习更新的，所以初始化为零矩阵。

Q-learning算法的状态转移规则：

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (2)$$

Q-learning伪代码如下：

```

1 Initialize Q(s,a)
2 Repeat (for each episode):
3   Initialize s
4   Repeat (for each step of episode):
5     Choose a from s using policy derived from Q(e.g. ε-greedy)
6     Take action a, observe r, s'
7     Q(s,a) ← Q(s,a) + α[r+γmaxQ(s',a')-Q(s,a)]
8     s ← s';
9   until s is terminal

```

1.4 DQN

如果我们遇到复杂的状态集合，如状态是连续的，则就算离散化后，集合也很大，此时若使用传统方法Q-Learning，无法在内存中维护这么大的一张Q表。这时就引入了Deep Q_Learning(DQN)。

Deep Q-Learning算法的基本思路来源于Q-Learning。但是和Q-Learning不同的地方在于，它的Q值的计算不是直接通过状态值 s 和动作 a 来计算，而是通过神经网络来计算的。在本次实验中，DQN的输入是我们的状态 s 对应的状态向量 $\phi(s)$ ，输出是 $1 \times 65 \times 1$ 的三维向量，表示该状态下每个动作对应的得分。

DQN主要使用的技巧是经验回放（experience replay），即将每次和环境交互得到的奖励与状态更新情况都保存起来，用于后面目标Q值的更新。为什么需要经验回放呢？我们回忆一下Q-Learning，它是有一张Q表来保存所有的Q值的当前结果的，但是DQN是没有的，那么在做动作价值函数更新的时候，就需要经验回放。

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

1.5 nature DQN

在DQN中，我们使用的目标Q值的计算方法是：

$$y_i = \begin{cases} R_j & \text{is_end}_j \text{ is true} \\ R_j + \gamma \max_{a'} Q(\phi(S'_j), A'_j, \omega) & \text{is_end}_j \text{ is false} \end{cases}$$

这里目标Q值的计算使用到了当前要训练的 Q 网络参数来计算 $Q(\phi(S'_j), A'_j, \omega)$ ，而实际上，我们又希望通过 y_j 来后续更新 Q 网络参数。两者循环依赖，迭代起来两者的相关性太强，不利于算法的收敛。

因此，一个改进版的DQN: **Nature DQN**尝试用两个Q网络来减少目标Q值计算和要更新 Q 网络参数之间的依赖关系。Nature DQN使用了两个Q网络，一个当前Q网络 Q 用来选择动作，更新模型参数；另一个目标Q网络 Q' 用于计算目标Q值。目标Q网络 Q' 的网络参数不需要迭代更新，而是每隔一段时间从当前Q网络 Q 复制过来，即延时更新，这样可以减少目标 Q' 值和当前的Q值相关性。且两个Q网络的结构是一模一样的，这才可以复制网络参数。

1.6 Dueling DQN

Dueling DQN是DQN的又一个变种，其通过优化神经网络的结构来优化算法。

在DQN算法中，神经网络输出的 Q 值代表动作价值，那么单纯的动作价值评估会不会不准确？我们知道， $Q(s, a)$ 的值既和 s 有关，又和 a 有关，但是这两种“有关”的程度不一样，或者说影响力不一样。因此，有论文提出通过优化神经网络结构，来反映两个方面的差异。

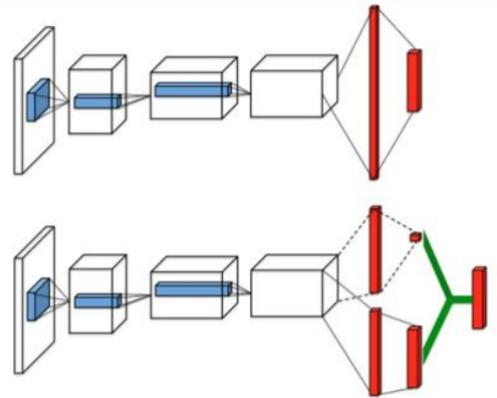
Dueling DQN考虑将Q网络分成两部分，第一部分是仅仅与状态 S 有关，与具体要采用的动作 A 无关，这部分我们叫做**价值函数**，记做 $V(S, \omega, \alpha)$ 。第二部分同时与状态 S 和动作 A 有关，这部分叫做**优势函数(Advantage Function)**部分，记为 $A(S, A, \omega, \beta)$ 。那么最终的价值函数可以重新表示为：

$$Q(S, A, \omega, \alpha, \beta) = V(S, \omega, \alpha) + A(S, A, \omega, \beta)$$

其中， ω 是公共部分的网络参数，而 α 是价值函数独有部分的网络参数，而 β 是优势函数独有部分的网络参数。

具体来讲，状态价值函数就等于在该状态下所有可能动作所对应的动作值乘以采取该动作的概率的和。更通俗的讲，状态价值函数 $V(s)$ 是该状态下所有动作值函数关于动作概率的平均值；而动作价值函数 $q(s, a)$ 表示在状态 s 下选取动作 a 所能获得的价值。而优势函数 $A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$ 是当前动作价值相对于平均价值的大小，即动作价值相比于当前状态的值的优势。如果优势大于零，则说明该动作比平均动作好，如果优势小于零，则说明当前动作还不如平均动作好。这样那些比平均动作好的动作将会有更大的输出，从而加速网络收敛过程。

其网络结构图如下：



- 其中，上面的是 DQN ，其实就是 $CNN+全连接$ ；
- 下面是 $Dueling DQN$ 的结构，只是把最后一个隐藏层拷贝一份，并分别全连接成两部分，即添加一个隐藏层，对应上面提到的价格函数网络部分和优势函数网络部分，分别输出 V 和 A ，其中 V 只有一维，表示该状态的得分， A 和动作的维度是一样的，表示执行某个动作相对于该状态可以获得的额外得分。然后再由 V 和 A 通过线性组合得到 Q 值。
- 注意：倒数第二层隐藏层的大小和原来 DQN 对应的隐藏层大小是一样的，上图只不过画小了）。

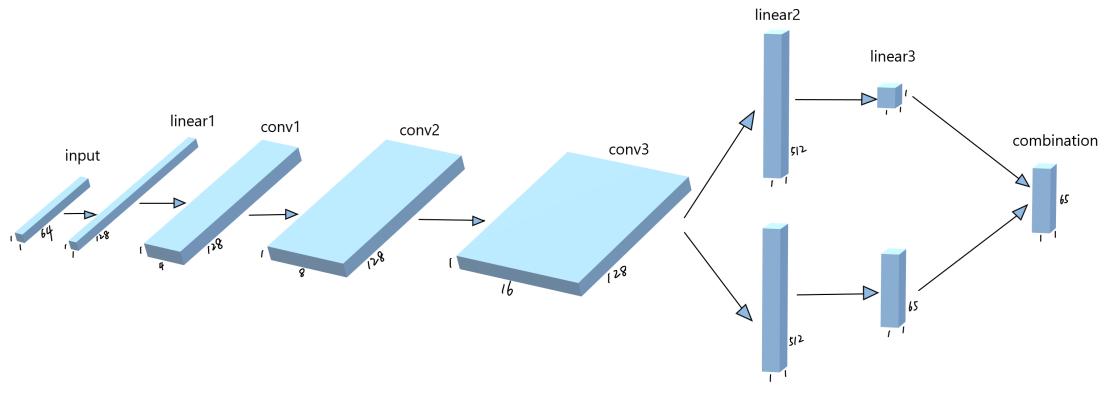
对优势函数做进一步的中心化处理，可以得到最终的线性组合公式为：

$$Q(S, A, \omega, \alpha, \beta) = V(S, \omega, \alpha) + (A(S, A, \omega, \beta) - \frac{1}{\mathcal{A}} \sum_{a' \in \mathcal{A}} A(S, a', \omega, \beta))$$

1.7 DQN 网络结构

在本次实验中，棋盘大小是 8×8 的，即状态 $state$ 可以用 1×64 大小的向量来表示棋盘，当对应位置取 1 表示黑棋，取 -1 表示白棋，取 0 表示空。而动作 $action$ 则表示在某个位置落子，因此也可以用 1×64 的向量表示 $action$ ，但考虑到有可能没有位置可下，所以增加一个表示跳过的 $action$ ，即用 1×65 的向量表示 $action$ ，即在此状态 $state$ 下各个 $action$ 的得分。

因此，模型的结构如下：



二、实现过程

在本次实验中，我们总共实现了三个模块：*reverse.py*, *run.py* 和 *DQN.py*

2.1 *reverse.py*

这一模块主要是游戏界面的初始化，游戏中的一些判断函数，具体分析如下：

```

1 class Game(object):
2     def __init__(self):
3         self.board_size = BOARD_SIZE
4         self.black_chess = set()
5         self.white_chess = set()
6         self.board=[[0 for j in range(self.board_size)] for i in
7             range(self.board_size)]
8
9         self.black_chess.add((BOARD_SIZE // 2 - 1, BOARD_SIZE // 2))
10        self.black_chess.add((BOARD_SIZE // 2, BOARD_SIZE // 2 - 1))
11        self.white_chess.add((BOARD_SIZE // 2 - 1, BOARD_SIZE // 2 - 1))
12        self.white_chess.add((BOARD_SIZE // 2, BOARD_SIZE // 2))
13 #这个部分是对棋盘的初始化，首先将所有位置置为0（表示此时棋盘为空）
14 #接着将board的指定位置放上黑白棋子，add()函数会在之后分析

```

. Get_Valid_Pos函数：

```

1 def Get_Valid_Pos(self, my_chess, oppo_chess):
2     #找到此时所有可以下棋的position
3     valid_pos = set()
4     for (x, y) in my_chess:
5         temp = (x, y)
6         while (x - 1, y) in oppo_chess:
7             x -= 1
8         if (x, y) != temp and x - 1 >= 0 and (x - 1, y) not in my_chess:
9             valid_pos.add((x - 1, y))
10    #这部分代码较长，因此截取一部分
11    #其实找到有效位置的核心就是所有my_chess的点出发，分别向左/右/上/下/左上/左下/右上/右下遍历
12    #所示代码就是向左遍历直至找不到对方的棋，则将最后位置的左边设为valid_pos

```

. Game_over函数：

- 这里涉及到reward的设计：当如果是黑色胜利则返回100，如果是白色胜利则返回-100作为reward。
- 其余情况返回“黑色棋子数-白色棋子数”作为reward（可能是游戏过程中的reward也可能是平局）

```

1 def Game_over(self):
2 #判断游戏是否结束
3 #黑白棋结束的评价标准有两个：①棋盘下满了，这时统计哪一方剩下的棋子比较多；②棋盘上只剩下了一种颜色
4 #如果是黑色胜利则返回100，如果是白色胜利则返回-100作为reward
5 #其余情况返回“黑色棋子数-白色棋子数”作为reward（可能是游戏过程中的reward也可能是平局）

```

• Reverse函数：

```

1 def Reverse(self, last_pos, my_chess, oppo_chess, my_color):
2 #在每次下完一步棋之后将对方棋子颜色翻转

```

• Get_state函数：

```

1 def Get_State(self):
2 #返回当前棋盘状态，为一个一维数组包含64个元素
3     return np.array(self.board, dtype=np.int).flatten()

```

• Add函数：

```

1 def Add(self, my_color, pos):
2 #在pos位置放置相对应颜色的棋子
3     if pos != 64:
4         (x, y) = (pos // self.board_size, pos % self.board_size)
5         #pos为一个在区间[0, 63]的整数，我们需要将其转换为横纵坐标
6         if my_color == 1:
7             self.black_chess.add((x, y))
8             self.board[x][y] = 1
9         elif my_color == -1:
10            self.white_chess.add((x, y))
11            self.board[x][y] = -1
12
13     #在加入棋子之后，相当于人/AI下了一步棋，我们需要调用Reverse函数来翻转棋子颜色
14     if my_color == 1:
15         self.Reverse((x, y), self.black_chess, self.white_chess, my_color)
16     elif my_color == -1:
17         self.Reverse((x, y), self.white_chess, self.black_chess, my_color)

```

2.2 Run.py

这一部分主要是通过 `pygame` 模块为游戏提供可视化界面，运行这一部分即可开始游戏，具体分析如下：

```

1 | for i in range(N - 1):
2 |     pygame.draw.line(surf, (0, 0, 0), (BOX*(2+i), BOX), (BOX*(2+i), WIDTH-BOX))
3 |     pygame.draw.line(surf, (0, 0, 0), (BOX, BOX*(2+i)), (WIDTH-BOX, BOX*(2+i)))
4 | #在空白棋盘上画线

```

```

1 | for i in range(len(board)):
2 |     for j in range(len(board[0])):
3 |         if board[i][j] == 1:
4 |             t = (int((j + 1.5) * BOX), int((i + 1.5) * BOX))
5 |             pygame.draw.circle(surf, (0, 0, 0), t, int(BOX / 3))
6 |         if board[i][j] == -1:
7 |             t = (int((j + 1.5) * BOX), int((i + 1.5) * BOX))
8 |             pygame.draw.circle(surf, (255, 255, 255), t, int(BOX / 3))
9 | #画上黑白棋子, (0,0,0)/(255,255,255)均为对应颜色的RGB值
10 |
11 for (x, y) in valid:
12     t = (int((y + 1.5) * BOX), int((x + 1.5) * BOX))
13     pygame.draw.circle(surf, (0, 255, 0), t, int(BOX / 3))
14 #对valid_pos也画上相应circle, 来提醒博弈方不要破坏规则
15
16 if isinstance(last_pos, tuple):
17     (x, y) = last_pos
18     t = (int((y + 1.5) * BOX), int((x + 1.5) * BOX))
19     pygame.draw.circle(surf, (255, 0, 0), t, int(BOX / 8))
20 #在最近一次下棋的地方的棋子中心画上一个红色小circle, 在电脑互相博弈时很有用

```

博弈的时候就是人和AI交替进行的过程：

```

1 #人走
2 for event in pygame.event.get():
3     if event.type == pygame.QUIT:
4         step += 1
5         running = False
6     #如果发生的事件是鼠标点击才继续博弈, 否则退出博弈
7     elif event.type == pygame.MOUSEBUTTONDOWN:
8         grid = (int(event.pos[1]/(BOX + .0)-1), int(event.pos[0]/(BOX + .0)-1))
9         if grid[0] >= 0 and grid[0] < 8 and grid[1] >= 0 and grid[1] < 8:
10             if grid in valid_pos:
11                 #只有点击的位置在棋盘范围内并且是valid_pos才可以添加棋子
12                 a = N * grid[0] + grid[1]
13                 game.Add(human_color, a)
14                 show_board(game.board, grid)
15                 time.sleep(1)
16                 step += 1 #当step为偶数时人走, step为奇数时AI走
17                 break

```

```

1 #AI走
2 s = game.Get_State()
3 a = ai.Choose_Action_EpsilonGreedy(s, game, ai_color, 0) #找到最佳落子位置
4 game.Add(ai_color, a)
5 grid = (a // N, a % N)
6 show_board(game.board, grid)
7 step += 1

```

```

1 #判断游戏结束，刷新游戏页面
2 #以黑方赢棋为例
3 if game.Gameover() == 1:
4     #延迟两秒刷新野页面
5     time.sleep(2)
6     white = 255, 255, 255
7     pygame.init()
8     screen = pygame.display.set_mode((400, 400))
9     myfont = pygame.font.Font(None, 30)
10    textImage = myfont.render("The winner is Black", True, white)
11    while True:
12        screen.blit(textImage, (100, 100))
13        pygame.display.update()

```

2.3 DQN.py

. 定义网络结构：

- 利用上面所画的Dueling_DQN网络结构（三个卷积层+2个全连接层），构建出 Net 类。
网络的输入是当前棋盘的状态，用一个 1×64 的 tensor 表示；经过此网络结构，输出是 1×65 维的 tensor，反映此状态下采取各种 action 的得分。

```

1 class NET(nn.Module):
2     def __init__(self):
3         super(NET, self).__init__()
4
5         self.linear1 = nn.Sequential(
6             nn.Linear(N_STATE, 128),
7             nn.LeakyReLU()
8         )
9         # self.linear1.weight.data.normal_(0, 0.1)
10
11        self.conv1 = nn.Sequential(
12            nn.Conv1d(1, 4, 3, 1, 1),   # in_channel=1, out_channel=4,
13            kernel_size卷积核大小=3, stride步长=1, padding=1
14            nn.LeakyReLU(inplace=True)
15        )
16        self.conv2 = nn.Sequential(
17            nn.Conv1d(4, 8, 3, 1, 1),
18            nn.LeakyReLU()
19        )
20        self.conv3 = nn.Sequential(
21            nn.Conv1d(8, 16, 3, 1, 1),
22            nn.LeakyReLU()
23        )
24        self.linear2_val = nn.Sequential(
25            nn.Linear(16 * 128, 512),
26            nn.LeakyReLU()
27        )
28        self.linear2_adv = nn.Sequential(
29            nn.Linear(16 * 128, 512),
30            nn.LeakyReLU()

```

```

31         self.linear3_adv = nn.Sequential(
32             nn.Linear(512, N_ACTION)
33         )
34         self.linear3_val = nn.Sequential(
35             nn.Linear(512, 1)
36     )

```

- ϵ – greedy 算法选择 action:

- ϵ – greedy 算法选择下一个 action。以 ϵ 概率随机选择一个 action，否则按网络的输出结果来选择最优动作；
- 在黑白棋游戏中，因为每一步的合法动作是有限的，随机选择动作只能在合法动作集合里选择。且 ϵ – greedy 算法保留了一定几率选择随机动作，所以一定程度上削弱了网络对游戏状态的控制，让网络有一定概率去探索新的局面，避免了造成过拟合的情况。
- 函数的输入是当前状态，在 Q-Learning 中，选择下一个动作应该是查表得到的；但在 DQN 中没有这个表，所以要先经过 Q 网络得到一个状态的 Q 值，然后选择这向量里概率最大的 action。输出是得分最大的 action。

```

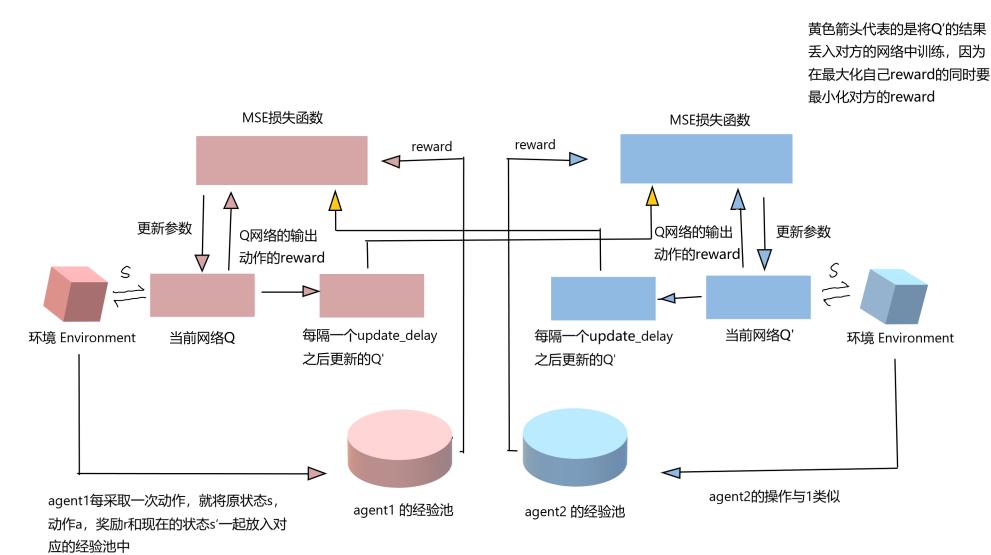
1 def Choose_Action_EpsilonGreedy(self, x, game_state, color, Epsilon=0.1):
2     if color == 1:
3         available_pos = game_state.Get_Valid_Pos(game_state.black_chess,
4                                         game_state.white_chess)
4     elif color == -1:
5         available_pos = game_state.Get_Valid_Pos(game_state.white_chess,
6                                         game_state.black_chess)
6
7     available_pos = list(map(lambda a: game_state.board_size * a[0] +
8                             a[1], available_pos)) # 列表, 表明合法位置
9     if len(available_pos) == 0:
10        return 64 # 表示这一步只能跳过
11
12        if np.random.uniform() < Epsilon: # random choose an action
13            action = np.random.choice(available_pos, 1)[0] # 从
14            available_pos里面抽取1个数字，并返回数组
15        else: # choose the max Q-value action
16            x = torch.tensor(x, dtype=torch.float)
17            x = x.view(1, -1)
18            actions_values = self.Q(x)[0] # 65维tensor, 各个action在各个位置的值
19            # (1*65维, 经过NET的结果)
20
21            ava_actions = torch.tensor(actions_values[available_pos])
22            _, action_ind = torch.max(ava_actions, 0)
23            action = available_pos[action_ind]
24
25        return action

```

- Learn 函数：更新网络参数

- 在本次的黑白棋实验中，训练过程实际上是两个agent博弈的过程。它们的目的都是最大化自己的reward，并相应的减小对方的reward。且连续两个状态是交替执行的，即连续两个状态的执行者是不一样的（除去无法落子的情况），此时agent切换情况不同于DQN的原始模型，因此需要修改DQN为“对抗性DQN”。

- 网络图如下：



- 对抗性DQN模型由两个DQN组成，分别对应与 $agent_1$ 和 $agent_2$ ，每个DQN包含上面提到的两个Q网络以提高稳定性。每个agent的经验池只记录属于自己的经验，即 $(s, a, r, s'), s \in agent_i$ 。训练的时候， s 和 s' 分别对应不同的agent的状态，因此，Q网络的更新公式为：

- 其中 Q_1 、 \widehat{Q}_1 、 Q_2 、 \widehat{Q}_2 分别对应于两个DQN网络的当前值网络和目标值网络。
对 Q_1 进行更新时，把后继状态 s' 输入对方的目标值网络，而不是输入到自身的目
标值网络。

$$Q_1(s, a) = Q_1(s, a) - \alpha[r + \gamma \max_{a'} \widehat{Q}_2(s', a') - Q_1(s, a)]$$

$$Q_2(s, a) = Q_2(s, a) - \alpha[r + \gamma \max_{a'} \widehat{Q}_1(s', a') - Q_2(s, a)]$$

- 每次训练时从经验池中挑选 minibatch 来训练样本，用这些样本得到的梯度来更新网
络。把后继状态 s' 输入对方的 Q' 网络得到的输出与自身网络的输出做一个均方误差，用
这个误差来反向传播更新参数。

```

1 def Learn(self, oppo_Q_):
2     for step in range(10):
3         # update parameters of Q_ 每隔一段时间将Q的参数直接给到Q_
4         if self.learn_iter % UPDATE_DELAY == 0:
5             self.Q_.load_state_dict(self.Q.state_dict())
6             self.learn_iter += 1
7         # randomly choose BATCH_SIZE samples to learn 从经验池中随机选取进行训
    练, 是数组
8         sample_index = np.random.choice(TRANSITIONS_CAPACITY, BATCH_SIZE)
9
10        batch_tran = self.transitions[sample_index, :]
11        batch_s = batch_tran[:, :N_STATE]
12        batch_a = batch_tran[:, N_STATE: N_STATE + 1]
13        batch_r = batch_tran[:, N_STATE + 1: N_STATE + 2]
14        batch_s_ = batch_tran[:, N_STATE + 2:]

```

```

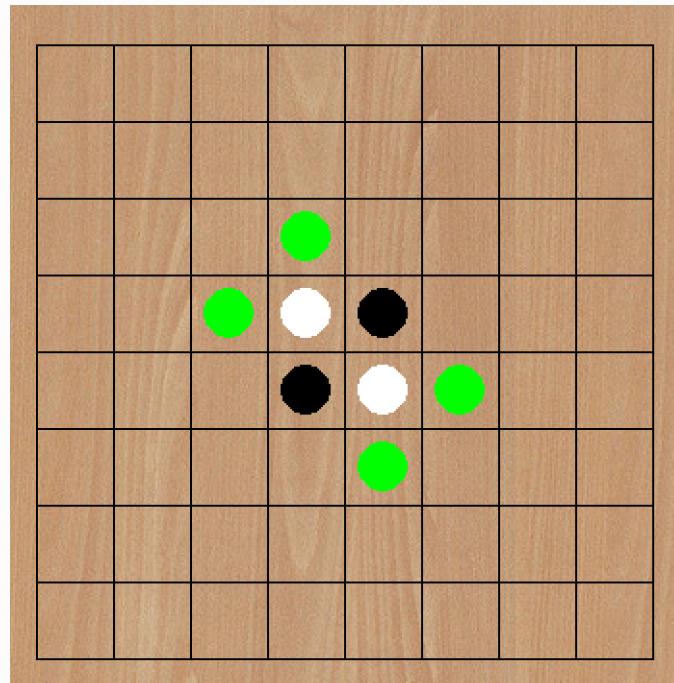
14
15     batch_s = torch.tensor(batch_s, dtype=torch.float)
16     batch_s_ = torch.tensor(batch_s_, dtype=torch.float)
17     batch_a = torch.tensor(batch_a, dtype=int)
18     batch_r = torch.tensor(batch_r, dtype=torch.float)
19
20     # gather函数:相当于从第一维取第batch_a位置的值
21     batch_y = self.Q(batch_s).gather(1,batch_a)  #
22     # 禁止梯度更新,在计算loss的时候会更新梯度,因为Q_不用更新,因此禁止梯度。
23     batch_y_ = oppo_Q_(batch_s_).detach_()
24     # max(1) return (value,index) for each row
25     batch_y_ = batch_r - GAMMA * torch.max(batch_y_, 1)[0].view(-1,1)
26
27     loss = self.criteria(batch_y, batch_y_)
28     self.optimizer.zero_grad()
29     loss.backward()
30     self.optimizer.step()

```

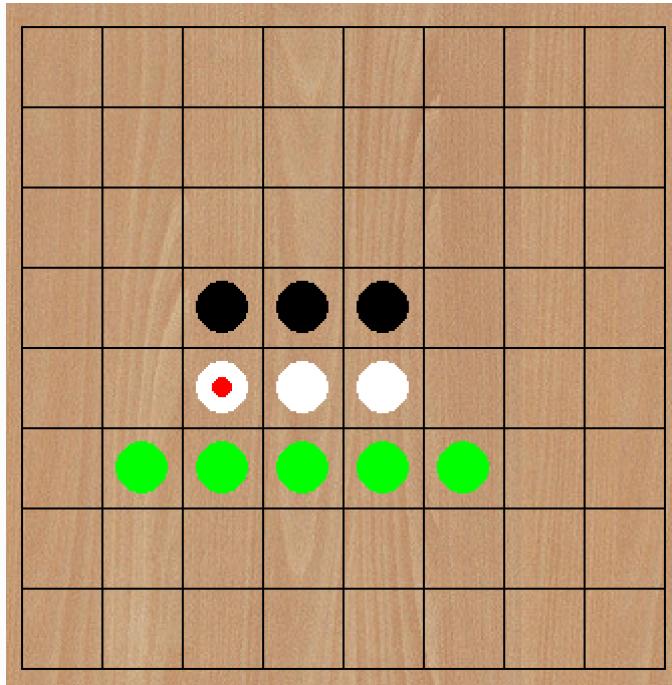
三、实验结果分析

强化学习不同于有监督学习，因为数据最开始并不存在标签，所以我们没办法采用准确率/召回率/F1值来评价模型好坏，只能通过模型结果来进行人和电脑的对弈，在过程中感受AI的下法是否准确合适。

由于篇幅限制，我们只选取了几个中间过程（假设人先手，此时人执黑棋）。

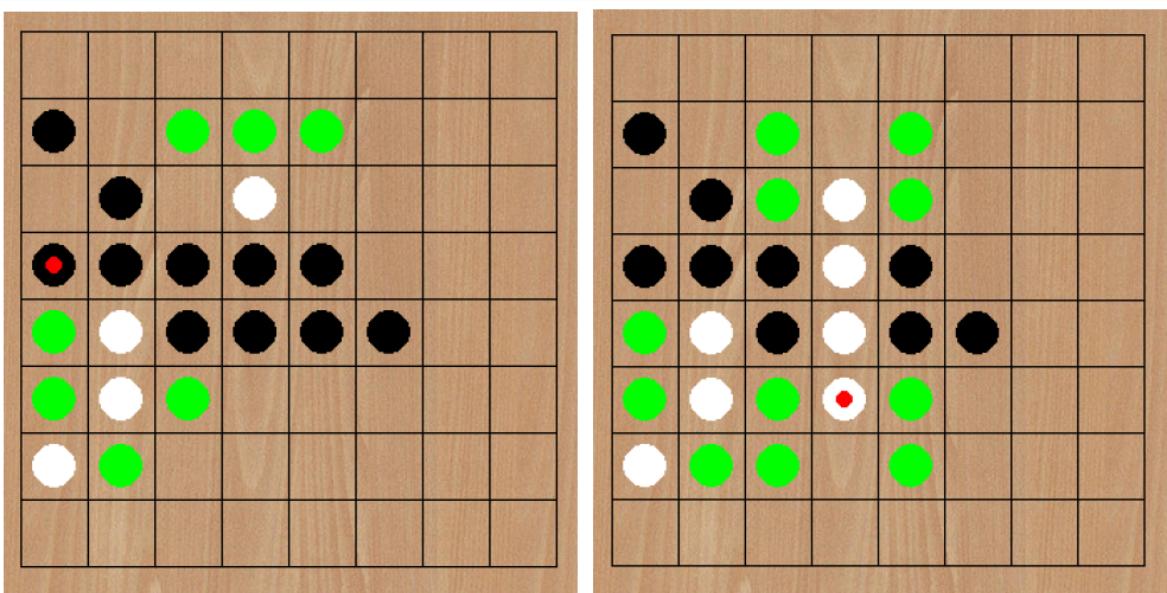


棋盘的初始状态



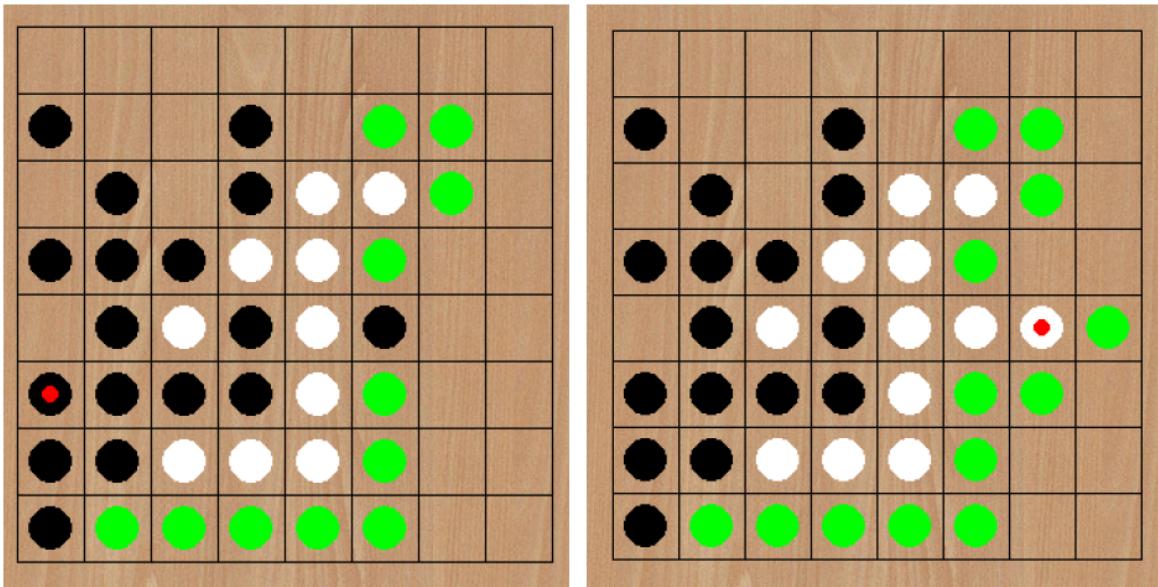
我们先将黑棋下在了(4,3)的位置，可以看到电脑将白棋下在了(5,3)。因为我们的评价函数是黑棋个数减白棋个数的差值，而我们遍历整个棋盘中白棋可以下的位置会发现，(5,3)是使reward最小的点（reward为0）。

注：此时人先走所以以为offensive方，目的是最大化reward；电脑则恰好相反。

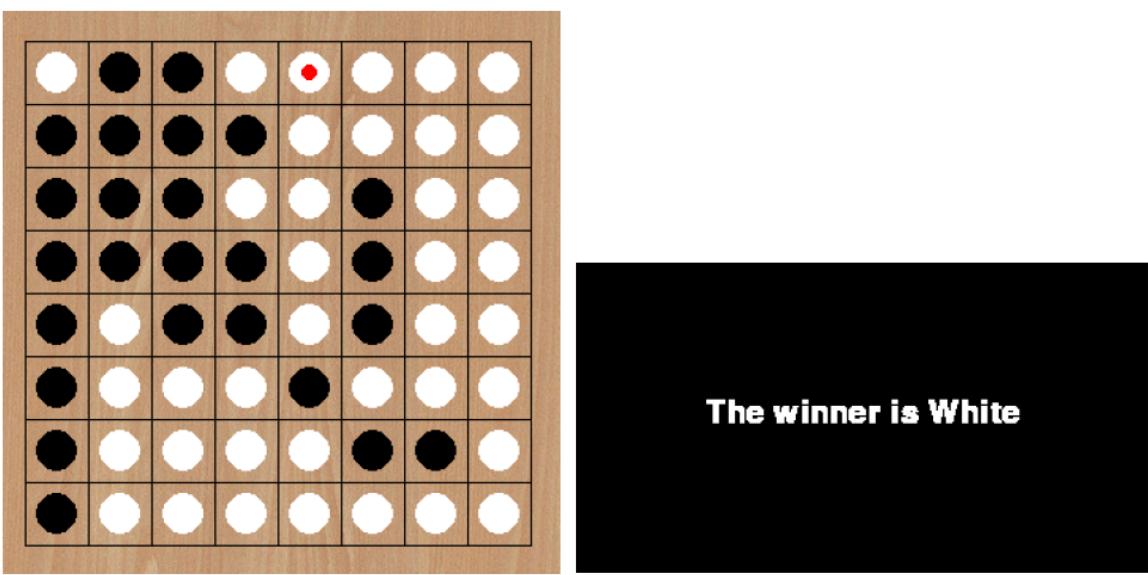


注：左图中的绿色并不是白方可以走的 `valid_pos`，而是此时黑方的有效位置，所以没有参考价值。

在某一时刻我们将黑棋落在了(4,1)，之后白棋落在了(6,4)上。对于左图中白方的每一个 `valid_pos`，我们均计算一下如果电脑下在这里所产生的reward是多少。（通俗来说就是下之后能翻转多少对方的棋子）其实当前状态的最佳下法是下在(5,7)，我觉得是模型的思考方式其实不同于我们人类。我们在下棋的时候通常只关注这一步的利益最大化，但是电脑采用的模型是从空棋盘到分出胜负的所有采样点，所以AI在下棋的时候不会只关注这一步，它当前下的这一步可能不是现在最有利，但一定是它采用的模型中的数据训练之后得到的结果。（也就是走一步看几步）



在某一时刻我们将黑棋落在了(6,1)，之后白棋落在了(5,7)上。此时对于所有白棋的valid_pos，落子之后只能最多翻转一个黑子。其实不从评价函数的角度出发，单看黑白棋的规则，我们就知道棋子尽量下在四周才更有机会赢，显然我们的模型也知道这一点，所以下在了离边界很近的点上。



最终是我们的模型取得了胜利。其实在中途有一些步骤内棋盘上几乎全是黑棋了，但是经过几个步骤后白棋又逐渐占领上风。这也是AI和人类不一样的思考方式：人类喜欢将此时的reward最大化，而AI习惯将所有可能的情况考虑在内，更在意*long-term benefit*。或许人类的顶尖棋手也是这样思考的，但是对我们这些业余爱好者而言确实更在乎这一步能翻转多少颜色。

四、创新点

1. 使用了DQN模型搭建模型的训练网络，解决了Q-learning的内存过大问题。
2. 引入了Nature DQN的两个Q网络（当前Q网络和目标Q网络）作为优化，增加模型稳定性。
3. 引入了Dueling DQN模型，优化了神经网络结构，加速模型收敛速度。
4. 自定义了reward函数，棋局中途时为“黑棋数量 - 白棋数量”，棋局结束时返回100或-100。