中山大學 SUN YAT-SEN UNIVERSITY 实验报告

深度强化学习（DRL）

一． 算法原理

1. 强化学习介绍

1) 强化学习的本质是一种决策问题，并且可以做连续决策。
2) 它有四个基本元素，分别为 agent、state、action、reward，agent 可以当做是一个进行学习的一个对象，state 是 agent 处于的状态，action 是 agent 采取的动作，reward 是 agent 采取某个 action 所获得的回报。
3) 在某种 state 状态下采取某种 action 动作称作策略。
4) 每一个状态都可以利用某种指标去衡量它的价值，这个指标称作评估方法。
5) 强化学习就是没有标签，通过不断尝试，然后反馈调节，从而学习到在什么样的状态下选择什么样的动作可以获得最好的结果。
6) 强化学习与监督学习的区别
   A) 有监督学习是通过已有的数据和数据对应的正确标签，学习数据和标签对应逻辑。
   B) 若将状态当作数据属性，动作当作标签，监督学习和强化学习都是在试图寻找一个映射，从已知属性/状态推断出标签/动作，强化学习中的策略相当于有监督学习中的分类/回归器。
   C) 强化学习刚开始并没有标签，它是在尝试动作后才能获得结果，通过反馈的结果信息不断调整之前的策略。
   D) 有监督学习的结果反馈是实时的，而强化学习的结果反馈有延时。
   E) 有监督学习的输入独立同分布，强化学习的输入总在变化，每个状态下的动作影响下个状态
7) 常见的强化学习方法有 Q-learning、Policy Gradient 等。

2. 深度学习介绍

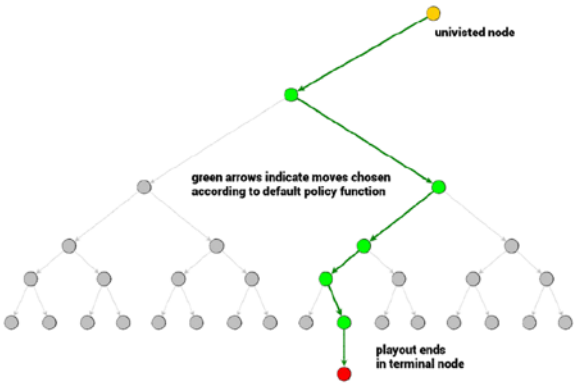
深度学习是借鉴人脑的机制提出的一个概率，通过模拟人脑运作的机制来解释数据。深度学习的本质是利用神经网络拟合一个高阶函数。由于期中大作业中已详细介绍了深度学习以及常见的模型，这里就不再赘述。

3. Monte Carlo Tree Search (MCTS)

1) 介绍：

MCTS 用于解决探索空间巨大的问题，即每一个节点都有很多子节点，那么随着探索深度的加深，那么探索空间会指数级增长，使得搜索时间非常长。但是显然有些节点是不用去探索，如果不去探索这些节点，那么搜索空间以及探索时间能得到大大的缓解。那么如何去选择哪些节点需要去探索呢，一般有α－β剪枝、MCTS 等方法解决。MCTS 不要求任何关于给定的领域策略或者具体实践知识来做出合理的决策。这个算法可以在没有任何关于博弈游戏除基本规则外的知识的情况下进行有效工作；这意味着一个简单的 MCTS 实现可以重用在很多的博弈游戏中，只需要进行微小的调整，所以这也使得 MCTS 是对于一般的博弈游戏的很好的方法。MCTS 执行一种非对称的树的适应搜索空间拓扑结构的增长。这个算法会更频繁地访问更加有趣的节点，并聚焦其搜索时间在更加相关的树的部分。这使得 MCTS 更加适

合那些有着更大的分支因子的博弈游戏，比如说 19X19 的围棋，这么大的组合空间会给标准的基于深度或者宽度的搜索方法带来问题，所以 MCTS 的适应性说明它最终可以找到那些更加优化的行动，并将搜索的工作聚焦在这些部分。



2） 描述：

对于树中的每一个节点，记录两个值，该节点的收益值Q，该节点的访问次数N。

该算法有四个步骤：选择、扩展、模拟、反向传播。

选择：如果当前节点是是完全扩展节点，即子节点都访问过，那么根据置信度上界(Upper Confidence Bound applied to Trees, UCT)选择要扩展的节点，置信度上界的公式为：$\mathrm{UCT}(u,v)=\frac{Q(v)}{N(v)}+c\sqrt{\frac{\log N(u)}{N(v)}}$；否则选择当前节点的未访问过的子节点进行扩展；

扩展：当要扩展的节点是完全扩展节点，返回选择步骤继续选择要扩展的节点；如果该节点不是终止节点，则在要扩展的节点上进行模拟，该节点也称为叶节点；

模拟：从该节点开始模拟至游戏结束，即分出胜负，最终在终止接节点获得收益值；

反向传播：沿着探索的路径反向传播，更新每个节点的Q和N，一般的更新的方式为：

$$Q_{new}=Q_{old}+q,$$

$$where\ q\ denotes\ the\ benefit\ value\ from\ the\ terminal\ node$$

$$N_{new}=N_{old}+1$$

4. DQN(Deep Q-learning Network)

1） 介绍：

在 Q-learning 中，每个状态 s 以及做出的相应的动作 a 和 Q 值是一一对应的，这就导致如果状态 s 和动作 a 太多种的话，将无法储存这样大的矩阵。所以为了解决这个问题，有一个方法就是找到一个函数表达式去表达这样一个矩阵，这样就不用去储存那么大的矩阵。那么为了拟合一个函数，最先想到就是深度神经网络，输入的是当前棋盘的某种向量表达，输出的是每个动作 a 对应的 Q 值。这就将深度学习 DL 和强化学习 RL 结合起来。但是 DL 与 RL 结合存在以下问题 ：

• DL 是监督学习需要学习训练集，强化学习不需要训练集只通过环境进行返回奖励值 reward，同时也存在着噪声和延迟的问题，所以存在很多状态 state 的 reward 值都是 0 也就是样本稀疏

• DL 每个样本之间互相独立，而 RL 当前状态的状态值是依赖后面的状态返回值的。

• 当我们使用非线性网络来表示值函数的时候可能出现不稳定的问题

DQN 中的机制决了以上问题：

- 通过 Q-Learning 使用 reward 来构造标签
- 通过 experience replay（经验池）的方法来解决相关性及非静态分布问题
- 使用一个 MainNet 产生当前 Q 值，使用另外一个 Target 产生 Target Q

2) 描述：

初始化一个价值函数$Q(s,a,;\theta)$，s 表示当前状态，a 表示动作，θ是随机权重，令$\widehat{Q}(s,a;\theta^{-})=Q$。给定一个初始状态$s_0$，利用 Q 迭代一定次数就可以形成多个数据集$(s,a,r)$，并将这些数据集储存起来，储存的地方称作经验池，大小是限定，当数据集的数量超过经验池的大小限度后，会将最早的数据集删除，腾出空间给新的数据。重复以上步骤一定次数后，利用经验池中的数据来训练 Q，假设取出的数据为(s,a,r)，设s'为状态 s 做出动作 a 的下个状态，令$y=\begin{cases} r, & s'是终止状态 \\ r+\gamma\max_{a'}\widehat{Q}(s',a';\theta^{-}), & 否则 \end{cases}$。那么对应 Q 函数的网络的训练数据的输入为$(x,a)$，输出为 y，训练一定次数，又重新令$\widehat{Q}=Q$。不断重复以上步骤，直至Q和$\widehat{Q}$收敛在一起。

5. Policy Network

1) 介绍：

Policy Gradients 方法直接预测在某个环境下应该采取的 action，而 Q-learning 方法预测某个环境下所有 action 的期望值即 Q 值。一般来说，传统的 Q-learning 方法只适合有少量离散取值的 action 环境，而 Policy Gradient 方法适合有连续取值的 action 环境。Policy Network 与 DQN 的主要区别是，Policy Network 输出的是 action 的收益值 Q，并根据这个收益值 Q 选择 action，而 DQN 使用的是$\epsilon-greedy$算法进行选择。

2) 描述：

A. 首先构建神经网络，网络的输入为状态 s，网络的输出为 action=1（1 表示采取该动作）的收益值 Q；
B. 在一个 episode 结束时（游戏胜负已分），将状态 s 重置。在下一次循环时，输入状态 s，输出一个每种落子的效益值p，根据p选取一个 action 并应用到状态 s，获取新的状态s'以及 reward，记录[s, action, reward]作为后续训练的数据；
C. 根据上面得到 reward，将整个 episode 的 reward 放到一个序列中，计算 discount_reward；
D. 当有足够的 batch 的 episode，就对网络进行梯度下降更新；

二. 伪代码

1. Monte Carlo Tree Search (MCTS)

```
function UCTSEARCH(s_0)
    create root node v_0 with sate s_0
    while within computational budget do
        v_t ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_t))
        BACKUP(v_t, Δ)
    return a(BESTCHILD(v_0, 0)
```

function TREEPOLICY(v)
while v is nonterminal **do**
if v not fully expanded **then**
return EXPAND(v)
else
$v \leftarrow$ BESTCHILD(v, Cp)
return v
function EXPAND(v)
choose $a \in$ untried actions from $A(s(v))$
add a new child v' to v
with $s(v') = f(s(v), a)$
and $a(v') = a$
return v'

function BESTCHILD(v, c)
return $\arg\max \frac{Q(v')}{N(v')} + c\sqrt{\frac{\log N(v)}{N(v')}}, v' \in$ children of v

function DEFAULTPOLICY(s)
while s is nonterminal **do**
choose $a \in A(s)$ uniformly at random
$s \leftarrow f(s, a)$
return reward for state s
function BACKUP(v, Δ)
while v is not null **do**
$N(v) \leftarrow N(v) + 1$
$Q(v) \leftarrow Q(v) + \Delta(v, p)$
$v \leftarrow$ parent of v

2. DQN(Deep Q-learning Network)

initialize replay memory D to capacity N
initialize action-value function Q with random weights θ
initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode* = 1, M **do**
initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
for t = 1, T **do**
with probability ϵ select a random action a_t
otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
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 D
sample random minibatch of transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

set $y_j = \begin{cases} r_j, \ if\ episode\ terminates\ at\ step\ j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), \ otherwise \end{cases}$

perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameter θ

every C steps reset $\hat{Q} = Q$

end for

end for

3. Policy Network

initialize replay memory D to capacity N
initialize neutral network Q with random weights θ
for *episode* = 1, M **do**
    initialize state s
    end = False
    initialize memory D' to capacity M
    while not *end* **do**
        $s' = s$
        select $a_t = \text{argmax}_a Q(s, a; \theta)$
        execute action a_t in emulator and observe reward r_t and update state $s = f(s, a)$
        store $[s', a_t, r_t]$ in D'
        if s is terminal state **then**
            end = **True**
    calculate *discound_reward* using all records in D'
    store D' and *discound_reward* in D
    every C steps sample minibatch of training data set from D to train neural network Q

三. 实现思路

借鉴 AlphaGo 实现思路，需要一个策略网络，即计算每个动作 action 的收益值 Q，需要一个价值网络，来拟合每种情况下的效益。所以将两种网络结合起来，即我们的网络应该能计算当前棋局下每个 action 的收益值 Q 以及当前局面的效益 reward（+1 代表赢，-1 代表输）。

设计好了网络的结构后，需要考虑的问题就是如何获得训练数据？利用 self-play 的思想，让两个网络进行对弈，并收集对弈过程中局面信息以及动作等信息，进而利用这些数据集去训练网络。一开始两个网络是一样的，所有的权重都是随机初始化的，训练的时候只训练其中一个，暂且称该网络是新网络，没有被训练的网络称作旧网络，然后在新的一轮收集训练信息的循环中，记录新网络的胜利次数，当胜利次数达到一定阈值，可以相信新网络是优于旧网络的，就将新网络复制到旧网络，并继续训练新网络。这就是训练的方法。

以上是实现黑白棋 AI 的大致思路，接下来详细介绍其中的重点。

1. 基础网络结构

网络的输入是棋局的情况，即每个位置的落子情况，正负一表示黑白棋，零表示没有棋子，网络的输出是在每个位置下子的收益值 Q，以及当前局面的效益，+1 代表网络预测会赢，-1 代表网络预测会输。网络的中间结构为：四层卷积层，每层卷积后都会正则化和经过激活函数，最后展开到二维并经过两层全连接层，每层全连接也会正则化和经过激活函数，最后通过不同维的全连接层以及不同的激活函数获得预测每个位置的落子的收益值 Q 和当前局面的效益。

网络的损失函数是预测的收益值 Q 与真实的收益值 Q 之间的交叉熵加上预测的效益与真实的效益之间的平方差

2. MCTS

MCTS 以上述网络为基础搜索的，首先判断初始节点是不是终止节点，即游戏已分胜负，如果是，返回当前输赢状态（+1 代表赢，-1 代表输）；如果不是终止节点，再查看是不是叶节点，即是否被访问过，如果没有，返回网络预测的效益值，如果访问过，在其子节点中选择 UCT 最大的，继续往下搜索。在反向传播中，利用子节点返回效益值更新每个节点的 Q 值和 N 值。

以上是一次搜索的步骤，经过一定次数的 MCTS，就可以得到当前节点到每个子节点的 Q 值，可以做个归一化的处理，并将 Q 值列表当作每个动作的概率，并在这个概率下随机选择下一个动作，这就是网络+MCTS 做出的决策。

3. self-play 以及训练数据

新旧网络分别以先后手的身份进行 n 局对弈，再交换身份进行 n 局对弈，最后如果新网络的胜率超过一定数值，可以认定新网络比旧网络好，就将旧网络的权重置为新网络的权重，然后继续训练新网络，不断重复该步骤。训练数据也是从对弈中得来的，每次落子的选择是通过 MCTS 利用网络的输出搜索出来的（如何搜索出来可见**（三. 2. MCTS）**），训练数据即是[当前棋盘状态、MCTS 搜索得出的子节点 Q 值结果、这局的输赢情况（下这步棋的一方的角度）]。每次对弈完 2n 局后，就可以继续训练新网络。

四． 关键代码展示

1. 神经网络代码：

1） 基础网络结构（具体可见**三． 1. 基础网络结构**）：输入是当前棋局，有两个输出，一个输出是每个可下位置的概率值或者归一化后的收益值，另一个输出是+1 或-1，表示输赢。中间有三层卷积（每层之后都会有归一层），然后是两个层全连接层（每层之后都会有归一层），最后通过不同维度的全连接层获得不同的输出。

```python
class OthelloNNet(nn.Module):
    def __init__(self, game, args):
        # game params
        self.board_x, self.board_y = game.getBoardSize()
        self.action_size = game.getActionSize()
        self.args = args

        # 卷积
        super(OthelloNNet, self).__init__()
        self.conv1 = nn.Conv2d(1, args.num_channels, 3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(args.num_channels, args.num_channels, 3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(args.num_channels, args.num_channels, 3, stride=1)
        self.conv4 = nn.Conv2d(args.num_channels, args.num_channels, 3, stride=1)

        # 正则化
        self.bn1 = nn.BatchNorm2d(args.num_channels)
        self.bn2 = nn.BatchNorm2d(args.num_channels)
        self.bn3 = nn.BatchNorm2d(args.num_channels)
        self.bn4 = nn.BatchNorm2d(args.num_channels)
```

```python
        # 全连接
        self.fc1 = nn.Linear(args.num_channels*(self.board_x-4)*(self.board_y-4), 1024)
        self.fc_bn1 = nn.BatchNorm1d(1024)

        self.fc2 = nn.Linear(1024, 512)
        self.fc_bn2 = nn.BatchNorm1d(512)

        self.fc3 = nn.Linear(512, self.action_size)

        self.fc4 = nn.Linear(512, 1)
```

```python
    def forward(self, s):
        #                                                           s: batch_size x board_x x board_y
        s = s.view(-1, 1, self.board_x, self.board_y)               # batch_size x 1 x board_x x board_y
        s = F.relu(self.bn1(self.conv1(s)))                         # batch_size x num_channels x board_x x board_y
        s = F.relu(self.bn2(self.conv2(s)))                         # batch_size x num_channels x board_x x board_y
        s = F.relu(self.bn3(self.conv3(s)))                         # batch_size x num_channels x (board_x-2) x (board_y-2)
        s = F.relu(self.bn4(self.conv4(s)))                         # batch_size x num_channels x (board_x-4) x (board_y-4)
        s = s.view(-1, self.args.num_channels*(self.board_x-4)*(self.board_y-4))

        s = F.dropout(F.relu(self.fc_bn1(self.fc1(s))), p=self.args.dropout, training=self.training)  # batch_size x 1024
        s = F.dropout(F.relu(self.fc_bn2(self.fc2(s))), p=self.args.dropout, training=self.training)  # batch_size x 512

        pi = self.fc3(s)                                                                              # batch_size x action_size
        v = self.fc4(s)                                                                               # batch_size x 1

        return F.log_softmax(pi, dim=1), torch.tanh(v)
```

2） 定义网络的损失函数以及训练过程：网络的损失函数是 pi_loss + v_loss，pi_loss 是通过交叉熵算出来，v_loss 是通过均方误差算出来的；优化器是 Adam Optimizer。

```python
class NNetWrapper:
    def __init__(self, game):
        self.nnet = onnet(game, args)
        self.board_x, self.board_y = game.getBoardSize()
        self.action_size = game.getActionSize()

        if args.cuda:
            self.nnet.cuda()

    def train(self, examples):
        # examples: list of examples, each example is of form (board, pi, v)
        optimizer = optim.Adam(self.nnet.parameters())

        for epoch in range(args.epochs):
            print('EPOCH ::: ' + str(epoch+1))
            self.nnet.train()
            data_time = AverageMeter()
            batch_time = AverageMeter()
            pi_losses = AverageMeter()
            v_losses = AverageMeter()
            end = time.time()
```

```python
            bar = Bar('Training Net', max=int(len(examples)/args.batch_size))
            batch_idx = 0

            while batch_idx < int(len(examples)/args.batch_size):
                sample_ids = np.random.randint(len(examples), size=args.batch_size)
                boards, pis, vs = list(zip(*[examples[i] for i in sample_ids]))
                boards = torch.FloatTensor(np.array(boards).astype(np.float64))
                target_pis = torch.FloatTensor(np.array(pis))
                target_vs = torch.FloatTensor(np.array(vs).astype(np.float64))

                # predict
                if args.cuda:
                    boards, target_pis, target_vs = boards.contiguous().cuda(), target_pis.contiguous().cuda(), \
                                                    target_vs.contiguous().cuda()

                # measure data loading time
                data_time.update(time.time() - end)

                # compute output
                out_pi, out_v = self.nnet(boards)
                l_pi = self.loss_pi(target_pis, out_pi)
                l_v = self.loss_v(target_vs, out_v)
                total_loss = l_pi + l_v
```

```python
                # record loss
                pi_losses.update(l_pi.item(), boards.size(0))
                v_losses.update(l_v.item(), boards.size(0))

                # compute gradient and do SGD step
                optimizer.zero_grad()
                total_loss.backward()
                optimizer.step()

                # measure elapsed time
                batch_time.update(time.time() - end)
                end = time.time()
                batch_idx += 1

                # plot progress
                bar.suffix = '({batch}/{size}) Data: {data:.3f}s | Batch: {bt:.3f}s | ' \
                             'Total: {total:} | ETA: {eta:} | Loss_pi: {lpi:.4f} | Loss_v: {lv:.3f}'.format(
                             batch=batch_idx,
                             size=int(len(examples)/args.batch_size),
                             data=data_time.avg,
                             bt=batch_time.avg,
                             total=bar.elapsed_td,
                             eta=bar.eta_td,
```

```
    def loss_pi(self, targets, outputs):
        return -torch.sum(targets*outputs)/targets.size()[0]

    def loss_v(self, targets, outputs):
        return torch.sum((targets-outputs.view(-1))**2)/targets.size()[0]
```

2. 训练相关代码

1) 定义两个网络以及一个经验池：两个网络初始的权值是一样的，经验池是用来储存训练样本。

```
class Coach:
    # class executes the self-play + learning. It uses the functions defined
    # in Game and NeuralNet.

    def __init__(self, game, nnet, args):
        self.game = game
        self.nnet = nnet
        self.pnet = self.nnet.__class__(self.game)  # the competitor network
        self.args = args
        self.mcts = MCTS(self.game, self.nnet, self.args)
        self.trainExamplesHistory = []    # history of examples from args.numItersForTrainExamplesHistory latest iterations
        self.skipFirstSelfPlay = False    # can be overriden in loadTrainExamples()
```

2) 两个网络利用预测值以及 MCTS 进行对弈，以获得训练数据：为两个网络分别创建一棵蒙特卡洛树，每一次对弈过程，两个网络的蒙特卡洛树利用网络的预测值进行搜索，然后利用搜索结果选择落子。每次落子前的局面以及蒙特卡洛树的搜索结果将成为训练样本，用来训练新网络。其中，落子前的局面可以通过旋转等操作获得镜像的局面，用来增强数据。

```
    def executeEpisode(self):
        # This function executes one episode of self-play, starting with player 1.
        # As the game is played, each turn is added as a training example to
        # trainExamples. The game is played till the game ends. After the game
        # ends, the outcome of the game is used to assign values to each example
        # in trainExamples.

        # Return trainExamples: a list of examples of the form (canonicalBoard,pi,v)
        # pi is the MCTS informed policy vector, v is +1 if
        # the player eventually won the game, else -1.
        trainExamples = []
        board = self.game.getInitBoard()
        self.curPlayer = 1
        episodeStep = 0
```

```
        while True:
            episodeStep += 1
            canonicalBoard = self.game.getCanonicalForm(board,self.curPlayer)
            temp = int(episodeStep < self.args.tempThreshold)

            pi = self.mcts.getActionProb(canonicalBoard, temp=temp)
            sym = self.game.getSymmetries(canonicalBoard, pi)
            for b,p in sym:
                trainExamples.append([b, self.curPlayer, p, None])

            action = np.random.choice(len(pi), p=pi)
            board, self.curPlayer = self.game.getNextState(board, self.curPlayer, action)

            r = self.game.getGameEnded(board, self.curPlayer)

            if r != 0:
                return [(x[0], x[2], r*((-1)**(x[1] != self.curPlayer))) for x in trainExamples]
```

3) 两个网络利用各自的蒙特卡洛树对弈，并记录新网络的胜率，以决定新网络是否优

于旧网络，若是，更新网络，否则新网络回退到旧网络，并继续训练新网络。每次对弈要重置网络的蒙特卡洛树，因为每次对弈过后，网络的权值可能会被更新。

```python
def learn(self):
    # Performs numIters iterations with numEps episodes of self-play in each
    # iteration. After every iteration, it retrains neural network with
    # examples in trainExamples (which has a maximum length of maxlenofQueue).
    # It then pits the new neural network against the old one and accepts it
    # only if it wins >= updateThreshold fraction of games.

    for i in range(1, self.args.numIters+1):
        # bookkeeping
        print('------ITER ' + str(i) + '------')
        # examples of the iteration
        if not self.skipFirstSelfPlay or i>1:
            iterationTrainExamples = deque([], maxlen=self.args.maxlenOfQueue)

            eps_time = AverageMeter()
            bar = Bar('Self Play', max=self.args.numEps)
            end = time.time()

            for eps in range(self.args.numEps):
                self.mcts = MCTS(self.game, self.nnet, self.args)   # reset search tree
                iterationTrainExamples += self.executeEpisode()
```

```python
                # bookkeeping + plot progress
                eps_time.update(time.time() - end)
                end = time.time()
                bar.suffix = '({eps}/{maxeps}) Eps Time: {et:.3f}s | ' \
                             'Total: {total:} | ETA: {eta:}'.format(eps=eps+1, maxeps=self.args.numEps,
                                                                    et=eps_time.avg, total=bar.elapsed_td,
                                                                    eta=bar.eta_td)

                bar.next()
            bar.finish()

            # save the iteration examples to the history
            self.trainExamplesHistory.append(iterationTrainExamples)

        if len(self.trainExamplesHistory) > self.args.numItersForTrainExamplesHistory:
            print("len(trainExamplesHistory) =", len(self.trainExamplesHistory), " => remove the oldest trainExamples")
            self.trainExamplesHistory.pop(0)
        # backup history to a file
        # NB! the examples were collected using the model from the previous iteration, so (i-1)
        self.saveTrainExamples(i-1)
```

```python
        # shuffle examples before training
        trainExamples = []
        for e in self.trainExamplesHistory:
            trainExamples.extend(e)
        shuffle(trainExamples)

        # training new network, keeping a copy of the old one
        self.nnet.save_checkpoint(folder=self.args.checkpoint, filename='temp.pth.tar')
        self.pnet.load_checkpoint(folder=self.args.checkpoint, filename='temp.pth.tar')
        pmcts = MCTS(self.game, self.pnet, self.args)

        self.nnet.train(trainExamples)
        nmcts = MCTS(self.game, self.nnet, self.args)

        print('PITTING AGAINST PREVIOUS VERSION')
        arena = Arena(lambda x: np.argmax(pmcts.getActionProb(x, temp=0)),
                      lambda x: np.argmax(nmcts.getActionProb(x, temp=0)), self.game)
        pwins, nwins, draws = arena.playGames(self.args.arenaCompare)

        print('NEW/PREV WINS : %d / %d ; DRAWS : %d' % (nwins, pwins, draws))
        if pwins+nwins == 0 or float(nwins)/(pwins+nwins) < self.args.updateThreshold:
            print('REJECTING NEW MODEL')
            self.nnet.load_checkpoint(folder=self.args.checkpoint, filename='temp.pth.tar')
```

```
        else:
            print('ACCEPTING NEW MODEL')
            self.nnet.save_checkpoint(folder=self.args.checkpoint, filename=self.getCheckpointFile(i))
            self.nnet.save_checkpoint(folder=self.args.checkpoint, filename='best.pth.tar')

    def getCheckpointFile(self, iteration):
        return 'checkpoint_' + str(iteration) + '.pth.tar'
```

3. MCTS 代码

1) 所有字典类型的变量存储的都是树的每个节点的信息，Qsa 表示在状态 s 采取动作 a 所能获得收益，Nsa 表示在状态 s 采取动作 a 的次数，Ns 表示达到状态 s 的次数，Ps 表示网络输入为状态 s 时的第一个输出，Es 表示状态 s 的输赢情况，Vs 表示在状态 s 时可下子的位置。

```
class MCTS():
    """
    This class handles the MCTS tree.
    """

    def __init__(self, game, nnet, args):
        self.game = game
        self.nnet = nnet
        self.args = args
        self.Qsa = {}       # stores Q values for s,a (as defined in the paper)
        self.Nsa = {}       # stores #times edge s,a was visited
        self.Ns = {}        # stores #times board s was visited
        self.Ps = {}        # stores initial policy (returned by neural net)

        self.Es = {}        # stores game.getGameEnded ended for board s
        self.Vs = {}        # stores game.getValidMoves for board s
```

2) 通过一定次数的 MCTS 搜索，不断更新每个节点的信息，最后将根结点的子节点的收益值归一化，并返回。

```
    def getActionProb(self, canonicalBoard, temp=1):
        # This function performs numMCTSSims simulations of MCTS starting fromcanonicalBoard.
        # eturns probs: a policy vector where the probability of the ith action is
        # proportional to Nsa[(s,a)]**(1./temp)

        for i in range(self.args.numMCTSSims):
            self.search(canonicalBoard)

        s = self.game.stringRepresentation(canonicalBoard)
        counts = [self.Nsa[(s,a)] if (s,a) in self.Nsa else 0 for a in range(self.game.getActionSize())]

        if temp == 0:
            bestA = np.argmax(counts)
            probs = [0]*len(counts)
            probs[bestA] = 1
            return probs

        counts = [x**(1./temp) for x in counts]
        counts_sum = float(sum(counts))
        probs = [x/counts_sum for x in counts]
        return probs
```

3) MCTS 搜索过程（具体可见**三.1.MCTS**），首先判断初始节点是不是终止节点，即游戏已分胜负，如果是，返回当前输赢状态（+1 代表赢，-1 代表输）；如果不是终止节点，再查看是不是叶节点，即是否被访问过，如果没有，返回网络预测的效益值，如果访问过，在其子节点中选择 UCT 最大的，继续往下搜索。在反向传播中，利用子节点返回效益值更新每个节点的 Q 值和 N 值。

UCT 公式为：

$$\mathrm{UCT} = \mathrm{Qsa} + c * \mathrm{Ps} * \sqrt{\frac{\mathrm{Ns}}{1 + \mathrm{Nsa}}}$$

更新公式为：

$$Qsa_{new} = \begin{cases} \frac{Nsa * Qsa_{old} + v}{Nsa + 1}, & sa\ 已被探索过 \\ v, & sa\ 未被探索过 \end{cases}$$

$$Nsa_{new} = \begin{cases} Nsa_{old} + 1, & sa\ 已被探索过 \\ 1, & sa\ 未被探索过 \end{cases}$$

```
def search(self, canonicalBoard):
    # one iteration of MCTS. It is recursively called
    # till a leaf node is found.
    # The action chosen at each node is one that
    # has the maximum upper confidence bound as in the paper.

    # Returns v: the negative of the value of the current canonicalBoard

    s = self.game.stringRepresentation(canonicalBoard)

    if s not in self.Es:
        self.Es[s] = self.game.getGameEnded(canonicalBoard, 1)
    if self.Es[s] != 0:
        # terminal node
        return -self.Es[s]
```

```
    if s not in self.Ps:
        # leaf node
        self.Ps[s], v = self.nnet.predict(canonicalBoard)
        valids = self.game.getValidMoves(canonicalBoard, 1)
        self.Ps[s] = self.Ps[s]*valids      # masking invalid moves
        sum_Ps_s = np.sum(self.Ps[s])
        if sum_Ps_s > 0:
            self.Ps[s] /= sum_Ps_s    # renormalize
        else:
            print("All valid moves were masked, do workaround.")
            self.Ps[s] = self.Ps[s] + valids
            self.Ps[s] /= np.sum(self.Ps[s])

        self.Vs[s] = valids
        self.Ns[s] = 0
        return -v

    valids = self.Vs[s]
    cur_best = -float('inf')
    best_act = -1
```

```
    # pick the action with the highest upper confidence bound
    for a in range(self.game.getActionSize()):
        if valids[a]:
            if (s,a) in self.Qsa:
                u = self.Qsa[(s,a)] + self.args.cpuct*self.Ps[s][a]*math.sqrt(self.Ns[s])/(1+self.Nsa[(s,a)])
            else:
                u = self.args.cpuct*self.Ps[s][a]*math.sqrt(self.Ns[s] + EPS)     # Q = 0 ?

            if u > cur_best:
                cur_best = u
                best_act = a

    a = best_act
    next_s, next_player = self.game.getNextState(canonicalBoard, 1, a)
    next_s = self.game.getCanonicalForm(next_s, next_player)

    v = self.search(next_s)
```

```
        v = self.search(next_s)

        if (s, a) in self.Qsa:
            self.Qsa[(s, a)] = (self.Nsa[(s, a)]*self.Qsa[(s, a)] + v)/(self.Nsa[(s, a)]+1)
            self.Nsa[(s, a)] += 1

        else:
            self.Qsa[(s, a)] = v
            self.Nsa[(s, a)] = 1

        self.Ns[s] += 1
        return -v
```

4. 黑白棋规则代码

1） 记录棋盘信息、获得其中一方可落子的位置、执行一步棋后棋局状态

```
class Board:
    # x is col, y is row in board
    # list of all 8 directions on the board, as (x,y) offsets
    __directions = [(1, 1), (1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1)]

    def __init__(self, n):
        # Set up initial board configuration

        self.n = n
        # Create the empty board array.
        self.pieces = [None]*self.n
        for i in range(self.n):
            self.pieces[i] = [0]*self.n

        # Set up the initial 4 pieces.
        self.pieces[int(self.n/2)-1][int(self.n/2)] = 1     # 后手
        self.pieces[int(self.n/2)][int(self.n/2)-1] = 1
        self.pieces[int(self.n/2)-1][int(self.n/2)-1] = -1  # 先手
        self.pieces[int(self.n/2)][int(self.n/2)] = -1
```

```
    # add [][] indexer syntax to the Board
    def __getitem__(self, index):
        return self.pieces[index]

    def count(self, color):
        count = 0
        for y in range(self.n):
            for x in range(self.n):
                if self[x][y] == color:
                    count += 1
        return count

    def countDiff(self, color):
        # Return the number of self's chees - opponent's
        # 1:white  -1:black 0:empty
        count = 0
        for y in range(self.n):
            for x in range(self.n):
                if self[x][y] == color:
                    count += 1
                if self[x][y] == -color:
                    count -= 1
        return count
```

```python
def get_legal_moves(self, color):
    # return valid move for next step
    moves = set()  # stores the legal moves.

    # Get all the squares with pieces of the given color.
    for y in range(self.n):
        for x in range(self.n):
            if self[x][y] == color:
                newmoves = self.get_moves_for_square((x, y))
                moves.update(newmoves)
    return list(moves)

def has_legal_moves(self, color):
    for y in range(self.n):
        for x in range(self.n):
            if self[x][y]==color:
                newmoves = self.get_moves_for_square((x, y))
                if len(newmoves) > 0:
                    return True
    return False
```

```python
def get_moves_for_square(self, square):
    # from x,y to discover all possible position
    (x, y) = square

    # determine the color of the piece.
    color = self[x][y]

    # skip empty source squares.
    if color == 0:
        return None

    # search all possible directions.
    moves = []
    for direction in self.__directions:
        move = self._discover_move(square, direction)
        if move:
            # print(square,move,direction)
            moves.append(move)

    # return the generated move list
    return moves
```

```
def execute_move(self, move, color):
    # Add the piece to the empty square.
    # print(move)
    flips = [flip for direction in self.__directions
                  for flip in self._get_flips(move, direction, color)]
    if len(list(flips)) > 0:
        for x, y in flips:
            #print(self[x][y],color)
            self[x][y] = color

def _discover_move(self, origin, direction):
    # Accroading to the rule of Othelogic to
    # get next steps
    x, y = origin
    color = self[x][y]
    flips = []

    for x, y in Board._increment_move(origin, direction, self.n):
        if self[x][y] == 0:
            if flips:
                # print("Found", x,y)
                return (x, y)
            else:
                return None
        elif self[x][y] == color:
            return None
        elif self[x][y] == -color:
            # print("Flip",x,y)
            flips.append((x, y))
```

```
def _get_flips(self, origin, direction, color):
    # return a position to drop in direction
    # initialize variables
    flips = [origin]

    for x, y in Board._increment_move(origin, direction, self.n):
        #print(x,y)
        if self[x][y] == 0:
            return []
        if self[x][y] == -color:
            flips.append((x, y))
        elif self[x][y] == color and len(flips) > 0:
            #print(flips)
            return flips

    return []
```

```
@staticmethod
def _increment_move(move, direction, n):
    # Generate move
    moves = []
    move = (move[0]+direction[0], move[1]+direction[1])
    while 0 <= move[0] < n and 0 <= move[1] < n:
        moves.append(move)
        move = list(map(sum, zip(move, direction)))
    return moves
```

5. UI 界面以及游戏主函数

1) 画棋局（显示已下棋子、可下的位置以及 AI 上一步棋的位置）

```
def display(g, board, cur_action=None, next_steps=None):
    surf.blit(background, (0, 0))
    bound_lines = [((GRID, GRID), (GRID, HEIGHT - GRID)),
                   ((GRID, GRID), (WIDTH - 4 * GRID, GRID)),
                   ((WIDTH - 4 * GRID, GRID), (WIDTH - 4 * GRID, HEIGHT - GRID)),
                   ((GRID, HEIGHT - GRID), (WIDTH - 4 * GRID, HEIGHT-GRID))]
    for line in bound_lines:
        pygame.draw.line(surf, BLACK, line[0], line[1], 2)

    for i in range(N - 1):
        pygame.draw.line(surf, BLACK,
                         (GRID * (i + 2), GRID),
                         (GRID * (i + 2), HEIGHT - GRID))
        pygame.draw.line(surf, BLACK,
                         (GRID, GRID * (i + 2)),
                         (WIDTH - 4 * GRID, GRID * (i + 2)))
```

```
    for i in range(N):
        for j in range(N):
            pos = (int((j + 1.5) * GRID), int((i + 1.5) * GRID))
            if cur_action is not None and (i, j) == cur_action:
                if board[i][j] == 1:
                    pygame.draw.circle(surf, WHITE, pos, int(GRID / 2))
                else:
                    pygame.draw.circle(surf, BLACK, pos, int(GRID / 2))
                pygame.draw.circle(surf, BLUE, pos, int(GRID / 6))
            else:
                if board[i][j] == 1:
                    color = WHITE
                elif board[i][j] == -1:
                    color = BLACK
                if board[i][j]:
                    pygame.draw.circle(surf, color, pos, int(GRID / 2))
```

```
    if next_steps is not None:
        for i in range(len(next_steps)):
            if next_steps[i] == 1:
                pos = (int((i % 8 + 1.5) * GRID), int((int(i / 8) + 1.5) * GRID))
                pygame.draw.circle(surf, GREEN, pos, int(GRID * 3/8))

    black_num = g.count(board, -1)
    white_num = g.count(board, 1)
    b_pos = (int(10.5 * GRID), int(GRID * 1.5))
    w_pos = (int(10.5 * GRID), int(GRID * 2.5))
    pygame.draw.circle(surf, BLACK, b_pos, int(GRID / 2))
    pygame.draw.circle(surf, WHITE, w_pos, int(GRID / 2))
    draw_text(str(black_num), 32, (GRID * 10 + (3 * GRID) // 2, 98), BLACK)
    draw_text(str(white_num), 32, (GRID * 10 + (3 * GRID) // 2, 170), BLACK)

    pygame.display.flip()
```

2) 显示信息

```
def draw_text(text, size, pos, color):
    font = pygame.font.Font(font_name, size)
    text_surface = font.render(text, True, color)
    text_rect = text_surface.get_rect()
    text_rect.midtop = pos
    surf.blit(text_surface, text_rect)
```

3) 游戏结束的提示信息

```
def game_end(winner):
    if winner == 1:
        draw_text('You win !', 64, (WIDTH // 2 - 72, HEIGHT // 2), RED)
    elif winner == 0:
        draw_text('Draw !', 64, (WIDTH // 2 - 72, HEIGHT // 2), RED)
    else:
        draw_text('You lose !', 64, (WIDTH // 2 - 72, HEIGHT // 2), RED)
    draw_text('Press any key to close', 64, (WIDTH // 2, HEIGHT // 2 + 64), BLUE)

    waiting = True
    while waiting:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.QUIT
                pygame.quit()
                return
            elif event.type == pygame.KEYUP:
                # waiting = False
                pass
        pygame.display.flip()
```

4) 主函数

```
if __name__ == "__main__":
    g = OthelloGame(8)
    n1 = NNet(g)
    n1.load_checkpoint('./models/', '8x8_100checkpoints_best.pth.tar')
    args1 = dotdict({'numMCTSSims': 50, 'cpuct': 1.0})
    mcts1 = MCTS(g, n1, args1)
    n1p = lambda x: np.argmax(mcts1.getActionProb(x, temp=0))

    running = True
    Choosing = True
    Dropping = False
    User = 1
    curplayer = -1      # 先手为黑
    cur_action = None

    board = g.getInitBoard()
    display(g, board)
```

```
    # 玩家是否选择先行
    draw_text('Othello Game', 64, ((WIDTH - 3 * GRID) // 2, 10 + HEIGHT // 4), BLACK)
    draw_text('Press \'F\' to be on the offensive or others to be on the defensive',
              22, ((WIDTH - 3 * GRID) // 2, 10 + HEIGHT // 2), BLUE)
    pygame.display.flip()

    while Choosing:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_f:
                    User = -1
                    Dropping = True
                Choosing = False

    while running:
        if g.getGameEnded(board, 1) != 0 and g.getGameEnded(board, -1) != 0:
            break
```

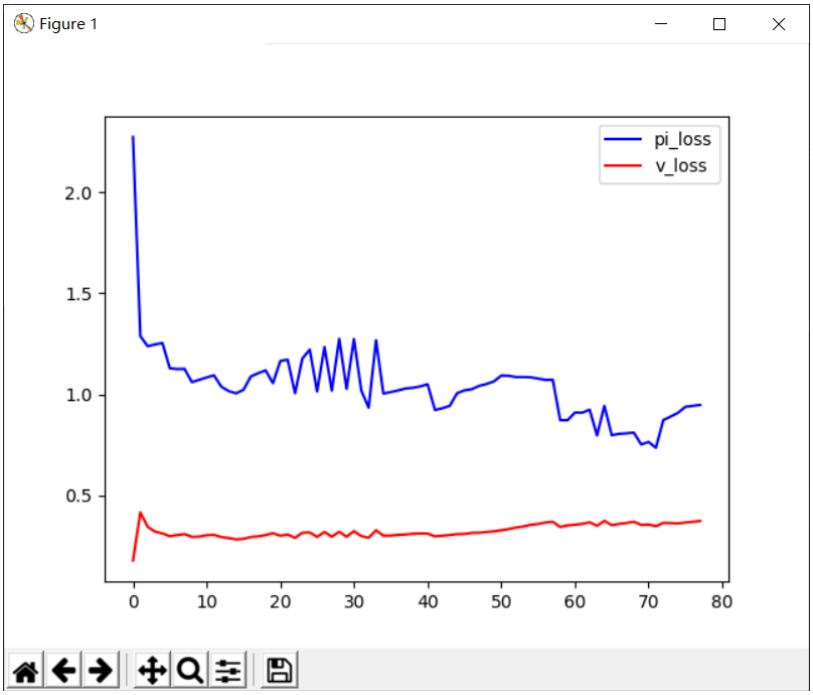
```
        next_steps = g.getValidMoves(g.getCanonicalForm(board, curplayer), 1)
        if next_steps[-1] == 1:
            if curplayer == User:
                print(curplayer)
                print('没得下')
                print()
                t.sleep(3)
            curplayer = -curplayer
            if curplayer == User:
                Dropping = True
            else:
                Dropping = False

        # 人
        while Dropping:
            next_steps = g.getValidMoves(g.getCanonicalForm(board, curplayer), 1)
            display(g, board, cur_action=cur_action, next_steps=next_steps)
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False
                    Dropping = False
                    pygame.quit()
                elif event.type == pygame.MOUSEBUTTONDOWN:
```

```
                    grid = (int(event.pos[1] / (GRID + .0)-1), int(event.pos[0] / (GRID + .0) - 1))
                    if 0 <= grid[0] < 8 and 0 <= grid[1] < 8:
                        index = g.n * grid[0] + grid[1]
                        if next_steps[index]:
                            board, curplayer = g.getNextState(board, curplayer, index)
                            display(g, board)
                            Dropping = False

        Dropping = True
        if not running:
            break

        # 机
        action = n1p(g.getCanonicalForm(board, curplayer))
        board, curplayer = g.getNextState(board, curplayer, action)
        cur_action = (int(action / N), action % N)
        display(g, board, cur_action=cur_action)

    if g.getGameEnded(board, User) == 1:
        game_end(1)
    elif g.getGameEnded(board, 1) == -1 and g.getGameEnded(board, -1) == -1:
        game_end(0)
    else:
```

```
        game_end(-1)
    pygame.quit()
```

五． 实验结果展示

1. 训练结果（只展示最新的）



pi_loss 是预测每个位置落子的概率与真实值之间的交叉熵，v_loss 是预测输赢与真实值之间的均方误差，整个网络的损失就是 pi_loss 与 v_loss 的和。在训练过程中，尽管 v_loss 呈现略微上升的趋势，但总体误差还是处于下降的趋势，说明训练还是有效果的。由于训练时间太长，所以迭代了接近 80 次。

2. **游戏过程**

通过测试，基本可以与 4399 小游戏中的高阶黑白棋下的有来有回，甚至可以下赢 4399，说明模型的效果挺好的，如果有时间还能继续训练下去，模型的效果可能会更好。

六. 实验分工

郑康泽：实验构思，代码结构安排，实现 DQN，Policy Network 及对应部分报告

朱绪思：实现 MCTS，UI 界面设计及对应部分报告