中山大学数据科学与计算机学院
计算机科学与技术专业
期末proj-黑白棋AI

2019年1月2日

课程名称：**Artificial Intelligence**

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	16337281	姓名	杨志成

目录

1 实验题目

本次期末proj是lab8的延申，助教提供了遗传算法和DQN的基本思路，在本次项目中我不仅改进了lab8的评估算法，也实现了遗传算法和DQN，并做出了一定的创新。

2 基于$\alpha-\beta$剪枝的改进算法

2.1 改进评估函数

我在lab8的基础上尝试了很多操作，一开始我的lab8只是基于棋盘位置的评估函数，在这个版本中加入了**稳定子**,**行动力**,**黑白子比例**, **位置特征**等评估手段，从几个方向对棋局综合评估，进而再由$\alpha-\beta$ 剪枝算法找出房前棋局下最优(近似)走子策略。

2.1.1 稳定子

稳定子的多少显然应当成为棋盘估值的重要指标。拥有更多的稳定子，一方面能保证我方最少棋子数，另一方面，稳定子可以起到辅助翻转大量对方棋子、形成成片稳定子的作用.而稳定子又分为两类，一类是边角稳定子，一类是内部稳定子，其判断方法不一样，重要程度亦不一样。直观上说，边界稳定子应当更为重要，因为边界稳定子往往会成片形成，很容易起到翻转对方大量棋子的作用，而内部稳定子相对显得不那么重要，因为其根本不能起到协助翻转的作用（因为其所有方向均被填满）。因此边界稳定子权重较大，其计算方法为，从棋盘某一角（包括该角）开始,权重分别为：

```
static const short sideVal[9] = {0, 1, 1, 1, 2, 3, 4, 6, 7};
```

这种权重设定方法将鼓励AI占边，形成稳定的强边效应，达成对我方极为有利的局面。而内部稳定子是在eval()中调用isStable()函数实现的，其参数实际被设置得较低。注意到，稳定子计算是eval()函数中最耗时的部分之一，故在对弈中前期，SDFACTOR和STFACTOR被设置为0，因为中前期几乎不可能会产生稳定子，将参数设为零能节约时间。

实现代码如下：

- 内部稳定子——其主要依据为

一个内部子为稳定子几乎等价于其八个方向均被棋子填满

```
327       //基于边界稳定子的估值----------------------------------------------------
328       SideEval = 2.5*countSideEval(chessboard, myplayer);
329
330       //基于内部稳定子的估值----------------------------------------------------
331       for(int i = 1; i < 7; i++){
332           for(int j = 1; j < 7; j++){
333               if(isStable(chessboard, i, j)){
334                   StableEval += (chessboard[i][j] == myplayer);
335                   StableEval -= (chessboard[i][j] == opponent);
336               }
337           }
338       }
339       StableEval = 12.5*StableEval;
```

- 计算稳定子

```
499   //计算边界稳定子估值
500   double countSideEval(char chessboard[][SIZE], int myplayer){
501       double SideEval = 0;
502       int opponent = -1*myplayer;
503       //int corner[4][2] = {{0,0},{0,7},{7,0},{7,7}};
504       if(chessboard[0][0] == myplayer){
505           for(int i = 0; i < 8; i++){
506               if(chessboard[0][i] == myplayer)
507                   SideEval += sideVal[i];
508               else
509                   break;
510           }
511           for(int i = 0; i < 8; i++){
512               if(chessboard[i][0] == myplayer)
513                   SideEval += sideVal[i];
514               else
515                   break;
516           }
517       }
518       else if(chessboard[0][0] == opponent){···
531       }
532
533       if(chessboard[0][7] == myplayer){···
546       }
```

2.1.2 行动力

行动力(Mobility)，简单来说，是指棋盘上某一方的可着子位置个数。行动力维持较高水平，几乎就能保证之后若干步内都有较好着法，因此行动力是比较重要的。行动力估值采用了与黑白子比例估值类似的百分比算法，但有所不同的是，行动力估值时会处理一些特殊局面:

```
353   if (!opValidCount) //如果对方无子可下
354       MobEval = 150; //返回较高的估值
355   else if (!myValidCount) //如果我方无子可下
356       MobEval = -450; //返回很低的估值
```

我方无处着子显然是一种不利的局面，而且根据经验，无处着子的情况往往会连续出现，形成十分糟糕的局面。在我方无处着子时返回一个较大的负值能很大程度上避免这种劣着，同理，对方无处着子是对我方有利的局面，返回一个较大的正值能使AI倾向于选择这种位置。

2.1.3 黑白子比例

从一定来说，我方棋子数比对方多能够说明我方占有优势，同时为了提高该项估值的稳定性，该项估值以百分比的形式呈现：

```
358   if (myStoneCount > opStoneCount)
359       BWRateEval = 100.0 * myStoneCount / (myStoneCount + opStoneCount);
360   else if (myStoneCount < opStoneCount)
361       BWRateEval = -100.0 * opStoneCount / (myStoneCount + opStoneCount);
362   else BWRateEval = 0;
```

这样的好处在于能够动态地反映黑白子在场上的状况，比求差或者直接用比例更合理，而且范围可预测.

2.1.4 位置特征

使用根据经验总结出的一张估值表:

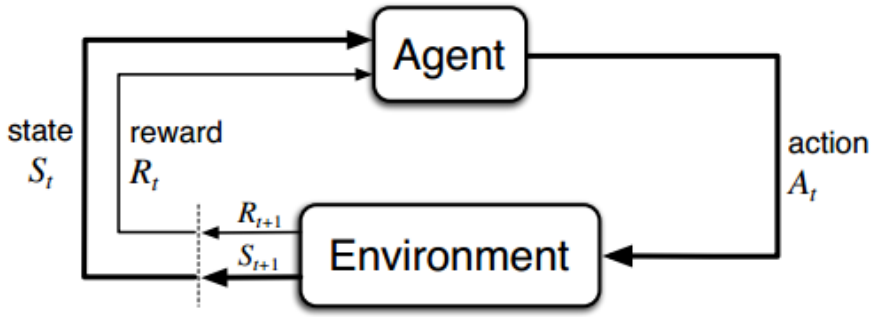
```
271       int weight[SIZE][SIZE] =
272       {
273           {20,-3,11, 8, 8,11,-3,20},
274           {-3,-7,-4, 1, 1,-4,-7,-3},
275           {11,-4, 2, 2, 2, 2,-4,11},
276           { 8, 1, 2,-3,-3, 2, 1, 8},
277           { 8, 1, 2,-3,-3, 2, 1, 8},
278           {11,-4, 2, 2, 2, 2,-4,11},
279           {-3,-7,-4, 1, 1,-4,-7,-3},
280           {20,-3,11, 8, 8,11,-3,20}
281       };
```

可以看到角位被赋予了最高的估值，角邻近位（紧靠每个角的两个边上的位置, 紧靠每个角的对角线上的位置,被认为是不好的，因为这些位置较容易让对方占角或者被对方翻转大量棋子,边上的其他位置都被赋予了很高的估值，这是基于迅速占边容易获得边角稳定子优势的考虑。最中心的位置被赋予较低的值。

3 强化学习

强化学习是智能体（Agent）以“试错”的方式进行学习，通过与环境进行交互获得的奖赏指导行为，目标是使智能体获得最大的奖赏，强化

学习不同于连接主义学习中的监督学习，主要表现在强化信号上，强化学习中由环境提供的强化信号是对产生动作的好坏作一种评价(通常为标量信号)，而不是告诉强化学习系统RLS如何去产生正确的动作。由于外部环境提供的信息很少，RLS必须靠自身的经历进行学习。通过这种方式，RLS在行动-评价的环境中获得知识，改进行动方案以适应环境。



3.1 从Q-learning理解强化学习

Q-Learning是一项无模型的增强学习技术，它可以在MDP问题中寻找一个最优的动作选择策略。它通过一个动作-价值函数来进行学习，并且最终能够根据当前状态及最优策略给出期望的动作。它的一个优点就是它不需要知道某个环境的模型也可以对动作进行期望值比较，这就是为什么它被称作无模型的。

Q–learning中，每个Q(s，a)对应一个相应的Q值，在学习过程中根据Q值，选择动作。Q值的定义是如果执行当前相关的动作并且按照某一个策略执行下去，将得到的回报的总和。最优Q值可表示为Q+，其定义是执行相关的动作并按照最优策略执行下去，将得到的回报的总和.

- **Q-learning算法:**

初始化$Q(s,a), \forall s \in S, a \in A(s)$,任意的数值，并且$Q(terminal-state,\cdot)=0$

重复（对每一节episode）：

初始化 状态S

重复（对episode中的每一步）：

使用某一个policy比如（$\epsilon-greedy$)根据状态S选取一个动作执行

执行完动作后，观察reward和新的状态S'

$Q(S_t,A_t) \leftarrow Q(S_t,A_t)+\alpha(R_{t+1}+\lambda \max_a Q(S_{t+1},a)-Q(S_t,A_t))$

$S \leftarrow S'$

循环直到S终止

对于Q-Learning，首先就是要确定如何存储Q值，最简单的想法就是用矩阵，一个s一个a对应一个Q值，所以可以把Q值想象为一个很大的表格，横列代表s，纵列代表a，里面的数字代表Q值.如下表示：

	a1	a2	a3	a4
s1	Q(1,1)	Q(1,2)	Q(1,3)	Q(1,4)
s2	Q(2,1)	Q(2,2)	Q(2,3)	Q(2,4)
s3	Q(3,1)	Q(3,2)	Q(3,3)	Q(3,4)
s4	Q(4,1)	Q(4,2)	Q(4,3)	Q(4,4)

虽然Q-learning算法根据value iteration计算出target Q值，但是这里并没有直接将这个Q值（是估计值）直接赋予新的Q，而是采用渐进的方式类似梯度下降，朝target迈近一小步，取决于α,这就能够减少估计误差造成的影响。类似随机梯度下降，最后可以收敛到最优的Q值。

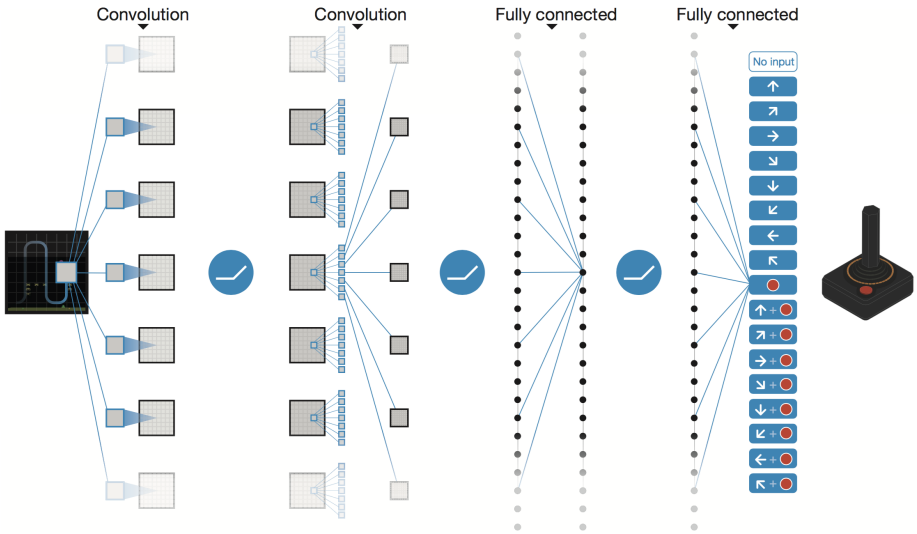
- **Q值更新公式:**

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t)+\alpha(R_{t+1}+\lambda \max_a Q(S_{t+1},a)-Q(S_t,A_t))$$

3.2 DQN(Deep Q-learning network)

在介绍DQN之前，我想先说说DQN的由来:2015年2月，Google研究员黄士杰博士在Nature上发表了一篇论文：Human-level control through deep

reinforcement learning。文章描述了如何让电脑自己学会打Atari 2600电子游戏。



文章主要描述了使用CNN(卷积神经网络)对游戏从像素级别上进行处理，运用强化学习的思想实现让电脑学会自己打游戏。黄士杰博士在论文中提到，他用这个模板实现的 AI已经在几十种游戏中超越了人类的排行榜。

3.2.1 Neural Network与强化学习

论文主要的贡献在于，首次提出将强化学习的思想于神经网络相结合。那么为什么需要引入神经网络，而不是直接使用Q-learning进行强化学习呢？

- **维度爆炸:**
  在Q-learning算法中我们使用矩阵来表示Q(s,a)，但是这个在现实的很多问题上是几乎不可行的，因为状态实在是太多。使用表格的方式根本存不下。比如在 Atari游戏中：计算机玩Atari游戏的要求是输入原始图像数据，也就是210x160 像素的图片，然后输出几个按键动作。总之就是和人类的要求一样，纯视觉输入，然后让计算机自己玩游戏。那么这种情况下，理论上看，如果每一个像素都有256种选择，那么就有$256^{210x160}$个矩阵元素。这在计算机中是绝对无法承受的空间复杂度。

- **解决方法：Q值神经网络化：**
  为了解决维度爆炸的问题，黄士杰博士提出了Q值神经网络化的方法。我们都知道神经网络的本质，其实就是在拟合函数，并且多层的神经网络理论上可以拟合任何形式的函数。因此使用神经网络来拟合Q(s,a)是完全可行的。用一个深度神经网络来作为Q值的网络，参数为w：

$$Q(s,a,w) \approx Q^{\pi}(s,a)$$

3.2.2 训练DQN

神经网络的训练是一个最优化问题，最优化一个损失函数loss function，也就是标签和网络输出的偏差，目标是让损失函数最小化。为此，需要巨量的有标签数据，然后通过反向传播使用梯度下降的方法来更新神经网络的参数。

- **如何为DQN提供有标签样本数据?**
  由于DQN训练的目的，是为了让Q网络的输出值与使用reward与Q值近似，即让神经网络的输出值近似于Q-learning中收敛的Q值。因此可以使用以下函数作为loss function：

$$L(w) = E[(\underbrace{r + \gamma \max_{a'} Q(s',a',w)}_{Target} - Q(s,a,w))^2]$$

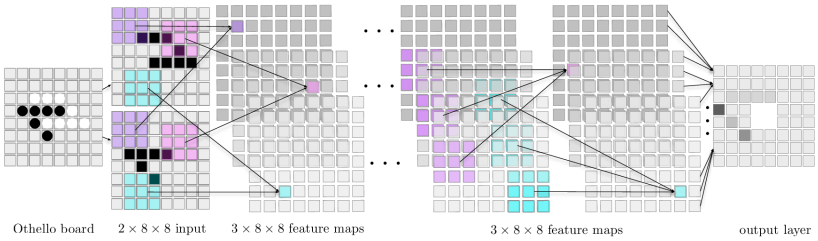
- **计算参数ww关于loss function的梯度**

$$\frac{\partial L(w)}{\partial w} = E[(r + \gamma \max_{a'} Q(s',a',w) - Q(s,a,w)) \frac{\partial Q(s,a,w)}{\partial w}]$$

- **使用SGD实现End-to-end的优化目标**
  有了上面的梯度，而$\frac{\partial Q(s,a,w)}{\partial w}$可以从深度神经网络中进行计算，因此，就可以使用SGD 随机梯度下降来更新参数，从而得到最优的Q值。

3.2.3 DQN网络结构

2017年11月，有学者在IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES 上发表了一篇有关于Othello游戏与深度学习相结合的论文，他使用的网络结构如下:



由于其网络层数过深，我无法在自己的笔记本电脑上训练，因此我使用了更加浅层的神经网络去训练。我自己的神经网络有四层：第一层将8x8x1的棋盘输入并卷积；第二层与第三层都是全连接层，size为8x8x2，激活函数使用的是softmax；第四层是输出层，size为8x8x1，使用了softmax作为输出。

3.2.4 DQN代码实现

主要使用了tensorflow实现神经网络的结构框架并进行训练，将棋盘类型封装成一个类。由于整体代码较多，下面只给出重要部分的代码，棋盘类实现暂不给出。

- **网络结构构建：**

```
39      def init_model(self): # 定义网络结构
40          # input 输入层
41          self.x = tf.placeholder(tf.float32, [None, self.rows, self.cols])
42          # 平坦输入层
43          size = self.rows * self.cols
44          x_flat = tf.reshape(self.x, [-1, size]) # 一行就是一个棋盘
45
46          # 第一层
47          W_fc1 = tf.Variable(tf.zeros([size, 200]))
48          b_fc1 = tf.Variable(tf.zeros([200]))
49          h_fc1 = tf.nn.relu(tf.matmul(x_flat, W_fc1) + b_fc1)
50
51          # 第二层
52          W_fc2 = tf.Variable(tf.zeros([200, 200]))
53          b_fc2 = tf.Variable(tf.zeros([200]))
54          h_fc2 = tf.nn.relu(tf.matmul(h_fc1, W_fc2) + b_fc2)
55
56          # 第三层
57          W_fc3 = tf.Variable(tf.zeros([200, 200]))
58          b_fc3 = tf.Variable(tf.zeros([200]))
59          h_fc3 = tf.nn.relu(tf.matmul(h_fc2, W_fc3) + b_fc3)
60
61          # 输出层
62          W_out = tf.Variable(tf.truncated_normal([200, self.n_actions], stddev
63          b_out_init = tf.zeros([self.n_actions])
64          b_out_init = b_out_init
65          b_out = tf.Variable(b_out_init)
66          self.y = tf.matmul(h_fc3, W_out) + b_out
```

- Q值计算：

```
79          # session计算图
80          self.sess = tf.Session()
81          self.sess.run(tf.global_variables_initializer())
82
83      # Q(s,a) 定义Q值， 其实就是返回网络计算结果
84      def Q_values(self, state):
85          return self.sess.run(self.y, feed_dict={self.x: [state]})[0]
```

- 损失函数：

```
68          # loss function
69          self.y_ = tf.placeholder(tf.float32, [None, self.n_actions])
70          self.loss = tf.reduce_mean(tf.square(self.y_ - self.y))
```

- 选择策略落子：

```
    # Q(s,a) 定义Q值， 其实就是返回网络计算结果
    def Q_values(self, state):
        return self.sess.run(self.y, feed_dict={self.x: [state]})[0]

    # 从输出Q值中选择策略
    def select_action(self, state, targets, epsilon):
        if np.random.rand() <= epsilon:
            return np.random.choice(targets)
        else:
            # 选择最大的Q(state, action)
            qvalue, action = self.select_enable_action(state, targets)
            return action
    # 选择可行解的最优策略
    def select_enable_action(self, state, targets):
        Qs = self.Q_values(state)
        index = np.argsort(Qs)
        for action in reversed(index):
            if action in targets:
                break
        # max_action Q(state, action)
        qvalue = Qs[action]
        return qvalue, action
```

- 网络自我博弈：

  胜者reward给100，其他步骤默认给0

```
            # 互相博弈
            for j in range(0, len(players)):
                reword = 0
                if end == True:
                    if win == playerID[j]:
                        # 获胜者奖励100点
                        reword = 100.0
                players[j].store_experience(state, targets, tr, \
                reword, state_X, target_X, end)
```

- 反向传播更新网络：

```
        if terminal:
            y_j[action_j_index] = reward_j
        else:
            qvalue, action = \
            self.select_enable_action(state_j_1, targets_j_1)
            y_j[action_j_index] = reward_j + self.discount_factor * qvalue

        state_minibatch.append(state_j)
        y_minibatch.append(y_j)

    # training
    self.sess.run(self.training, feed_dict\
    ={self.x: state_minibatch, self.y_: y_minibatch})
    # for log
    self.current_loss = self.sess.run(self.loss, feed_dict\
    ={self.x: state_minibatch, self.y_: y_minibatch})
```

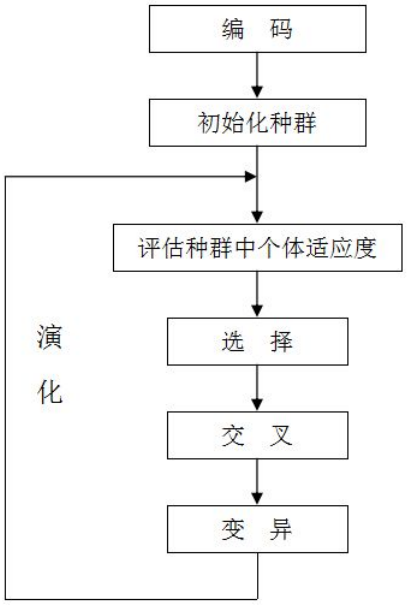
3.3 神经网络+遗传算法

先介绍遗传算法的主要思想：遗传算法(Genetic Algorithm)是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法。

其主要特点是直接对结构对象进行操作，不存在求导和函数连续性的限定；具有内在的隐并行性和更好的**全局寻优能力**；采用概率化的寻优方法，不需要确定的规则就能自动获取和指导优化的搜索空间，**自适应地调整搜索方向**。

遗传算法以一种群体中的所有个体为对象，并利用随机化技术指导对一个被编码的参数空间进行高效搜索。其中，选择、交叉和变异构成了遗传算法的遗传操作；参数编码、初始群体的设定、适应度函数的设计、遗传操作设计、控制参数设定五个要素组成了遗传算法的核心内容。

- **遗传算法过程**



初代种群产生之后，按照适者生存和优胜劣汰的原理，逐代(generation)演化产生出越来越好的近似解，在每一代，根据问题域中个体的适应度(fitness)大小选择 (selection）个体，并借助于自然遗传学的遗传算子(genetic operators)进行组合交叉(crossover)和变异(mutation)，产生出代表新的解集的种

群

这个过程将导致种群像自然进化一样的后生代种群比前代更加适应于环境，末代种群中的最优个体经过解码(decoding)，可以作为问题近似最优解。

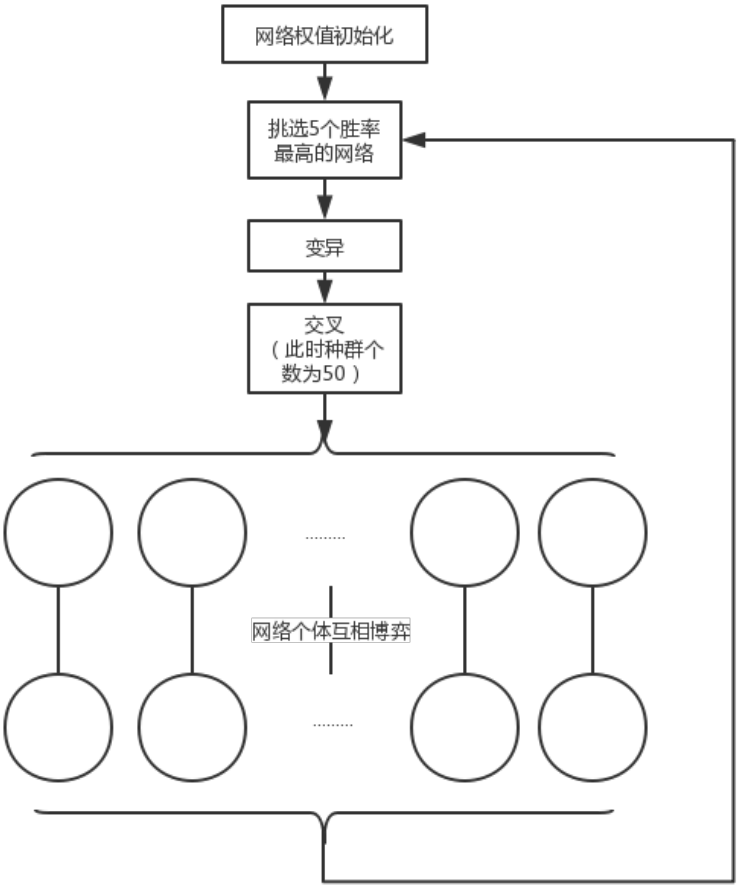
3.3.1 遗传算法在黑白棋中的应用

遗传算法其实可以应用在$\alpha-\beta$剪枝算法中对评估函数，各个权值系数的优化。可以将**稳定子**,**行动力**,**黑白子比例**,**位置特征** 四个重要因素的权值系数通过遗传算法，进行种群迭代，选取自我博弈过程中，获胜次数较多的权值系数，再通过这些权值进行交叉变异生成新的种群，进入新的迭代过程。末代种群中的最优个体可以视为最优权值解。

3.3.2 遗传算法与神经网络的结合

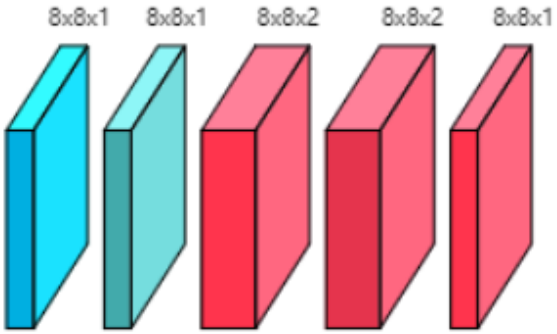
在遗传算法和DQN的启发下，我想到可以用神经网络当作用于决策的Q函数，利用遗传算法对神经网络的权值进行交叉变异，让多个神经网络不断自我博弈，挑选最优的几个网络生成下一代种群，开启新一轮的交叉变异。末代种群中的最优网络的参数可以视为网路权值的近似最优解。

- **遗传算法与神经网络迭代过程**



- **神经网络结构**

  神经网络的结构对于训练结果同样重要，在这里为了方便直接对神经网路权重进行遗传变异，我使用了python包numpy实现神经网络。网络共有四层，第一层将8x8x1的棋盘输入并卷积；第二层与第三层都是全连接层，size为8x8x2，激活函数使用的是softmax；第四层是输出层， size为8x8x1，使用了softmax作为输出。



3.3.3 ANN+GA代码实现

正如之前所说，为了方便对网络参数直接变异遗传，我使用了python库numpy实现神经网络。棋盘类单独实现，此处不再赘述。

- 神经网络结构

```
29  structure = [n * n, 2 * n * n, 2 * n * n, n * n] # 网络结构
30  # 定义神经网络
31  def evaluate(board, structure, weights, turn):
32      layer = np.array(sum(board, [])) # 第一层
33      negativeForArray = np.vectorize(negative)
34      if turn == w: # 默认为白色
35          layer = negativeForArray(layer)
36      for i in range(1, len(structure)):
37          layer = np.append(layer, 1) # 添加偏置b
38          layer = np.dot(weights[i - 1], layer) # 下一层
39          sigmoidForArray = np.vectorize(sigmoid)
40          layer = sigmoidForArray(layer) # 应用sigmoid函数
41      return layer
```

- 落子选择

```
42  # 选择最优落子策略
43  def chooseMove(board, sturcture, weights, turn):
44      # 神经网络输出结果
45      results = evaluate(board, structure, weights, turn)
46      # 可落子列表
47      validMoves = generateValidMovesList(board, turn)
48      bestMove = invalidMove
49      highestValue = -1
50      for coordinates in validMoves: # choose best move
51          valueOfCoordinates = results[n * coordinates[0] + coordinates[1]]
52          if valueOfCoordinates > highestValue:
53              highestValue = valueOfCoordinates
54              bestMove = coordinates
55      return bestMove
```

- 网络权值变异

```
    # mutation变异
    for i in range(bestNetworkPoplation):
        for _ in range(5): # do 5 mutations for each neural network
            newNetwork = deepcopy(networks[i])
            for p in range(len(newNetwork)):
                for q in range(len(newNetwork[p])):
                    for r in range(len(newNetwork[p][q])):
                        newNetwork[p][q][r] \
                        += random.uniform(-mutationAmount, mutationAmount)
            networks.append(newNetwork)
    #print("mutation: ", len(networks))
```

- 网络权值交叉

```
    for i in range(bestNetworkPoplation):
        for j in range(bestNetworkPoplation):
            if i != j:
                for _ in range(1): # do 1 crossovers for each neural network
                    newNetwork = deepcopy(networks[i])
                    useNetworkJWeight = True if random.randrange(0, 1) < 0.5 \
                    else False
                    for p in range(len(newNetwork)):
                        for q in range(len(newNetwork[p])):
                            for r in range(len(newNetwork[p][q])):
                                useNetworkJWeight = not useNetworkJWeight \
                                if random.randrange(0, 1) < 30/(8*n*n*n*n) \
                                    else useNetworkJWeight
                                if useNetworkJWeight:
                                    newNetwork[p][q][r] \
                                    = networks[j][p][q][r]
                    networks.append(newNetwork)
```

- 种群自我博弈：

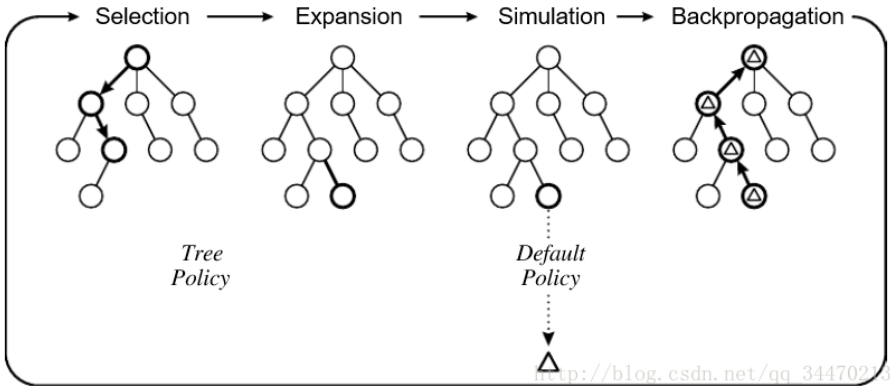
```
        fitnesses = [0 for _ in range(population)]
        for i in range(population): # black
            for j in range(population): # white
                if i != j:
                    turn = b
                    board = generateBoard()
                    whiteCanPlay = True # 若白色能下子
                    blackCanPlay = True # 若黑色能下子
                    while whiteCanPlay or blackCanPlay:
                        move = chooseMove(board, structure, \
                        networks[i if turn == b else j], turn)
                        if move == invalidMove:
                            if turn == w:
                                whiteCanPlay = False
                            if turn == b:
                                blackCanPlay = False
                        else :
                            if turn == w:
                                whiteCanPlay = True
                            if turn == b:
                                blackCanPlay = True
                            board = makeMove(board, move[0], move[1], turn)
                        # 交换下棋方
                        turn = opposite(turn)
                    if countChess(board, b) > countChess(board, w):
                        fitnesses[i] = fitnesses[i] + 1
                    elif countChess(board, b) < countChess(board, w):
                        fitnesses[j] = fitnesses[j] + 1
```

4 其他算法

除了上面那些我已经实现的算法，我还特意了解了一些其他的算法，但是迫于时间关系，并没有实现。这里就简单介绍一下这些算法，就当学习了一些新的知识。

4.1 MCTS蒙特卡洛树搜索

蒙特卡洛数搜索也是搜索算法的一个变种,但最近它在博弈系统中使用的非常广泛。



MCTS的主要步骤分为四个:

- **选择(Selection)**

  即找一个最好的值得探索的结点，通常是先选择没有探索过的结点，如果都探索过了，再选择UCB值最大的进行选择（UCB是由一系列算法计算得到的值，这里先不详细讲，可以简单视为value）

- **扩展(Expansion)**

  已经选择好了需要进行扩展的结点，那么就对其进行扩展，即对其一个子节点最为下一步棋的假设，一般为随机取一个可选的节点进行扩展。

- **模拟(Simulation)**

  扩展出了子节点，就可以根据该子节点继续进行模拟了，我们随机选择一个可选的位置作为模拟下一步的落子，将其作为子节点，然后依据该子节点，继续寻找可选的位置作为子节点，依次类推，直到博弈已经判断出了胜负，将胜负信息作为最终得分。

- **回溯更新(Backpropagation)**

  将最终的得分累加到父节点，不断从下向上累加更新。

4.2 AlphaZero使用的算法

AlphaZero使用的算法是基于MCTS的，它主要体现在以下三点:

- 单个神经网络收集棋局特征，在末端分支输出策略和棋局终止时的奖励

- 自我对弈的强化学习的蒙特卡罗树搜索(MCTS)
- 探索与利用的结合（Q+U 置信区间上限）

针对描述当前棋盘的一个状态(位置)s，执行一个由神经网络f_θ 指导的MCTS搜索，MCTS搜索输出每一步行为（在某个位置落子）的概率。MCTS搜索给出的概率通常会选择那些比由神经网络 $f_\theta(s)$ 给出的执行某一行为的概率要更强大。从这个角度看，MCTS搜索可以被认为是一个强大的策略优化工具。通过搜索来进行自我对弈——使用改善了的基于MCTS的策略来指导行为选择，然后使用棋局结果（哪一方获胜，用-1和1分别表示白方和黑方获胜）来作为标签数据——可以被认为是一个强大的策略评估工具。

Alpha Zero算法主体思想就是在策略迭代过程中重复使用上述两个工具：神经网络的参数得以更新，这样可以使得神经网络的输出 $(p, v) = f_\theta(s)$：移动概率和获胜奖励更接近与经过改善了的搜索得到的概率以及通过自我对弈得到的棋局结果，后者用 (π, z) 表示。得到的新参数可以在下一次自我对弈的迭代过程中让搜索变得更加强大。

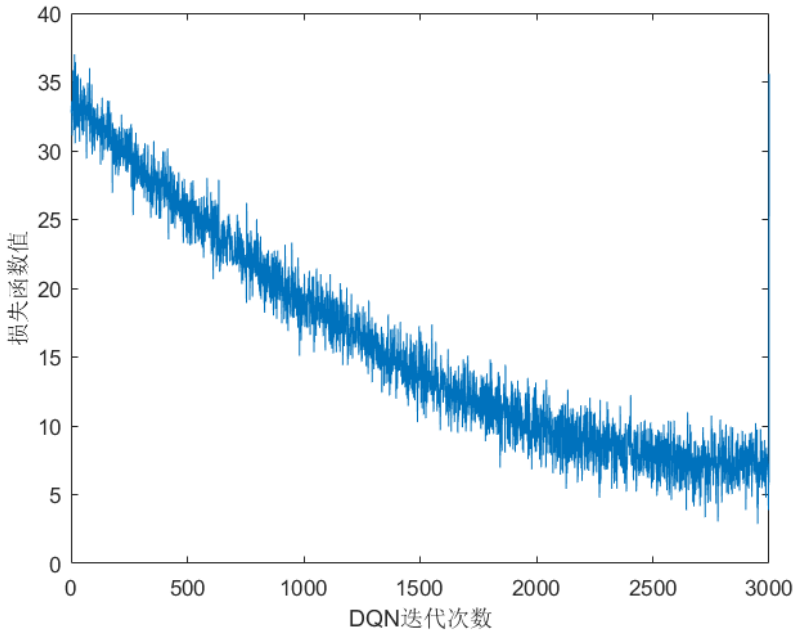
虽然我对$\alpha - zero$的实现很感兴趣，但由于期末压力，没有实现，以后有时间了一定尝试一下。

5 实验结果与分析

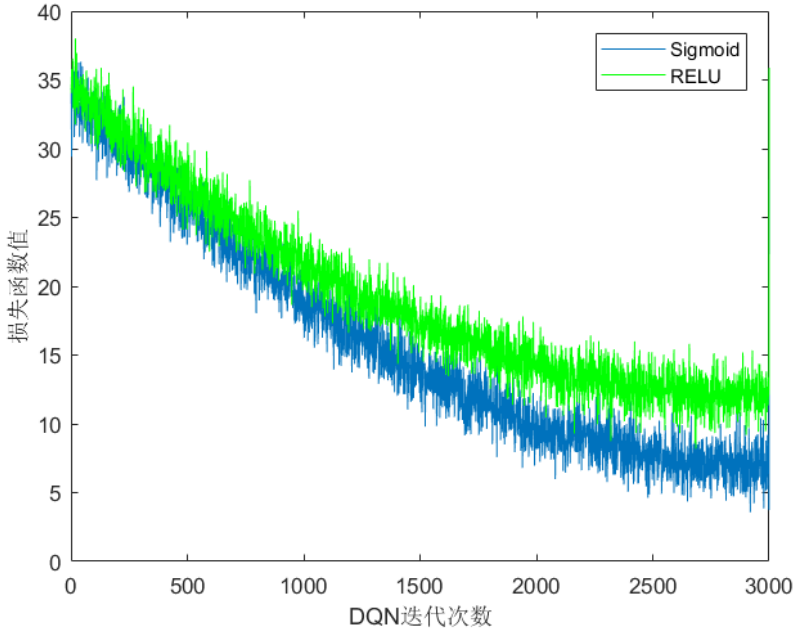
现将本次项目中得到得结果贴在下面：

5.1 DQN损失函数值

在3000轮的DQN迭代内，损失函数是在不断降低的，它运行了7小时跑出了3000 轮的数据，每一轮是根据当前agent对棋盘模拟棋局所对应的结果，进行梯度下降的。

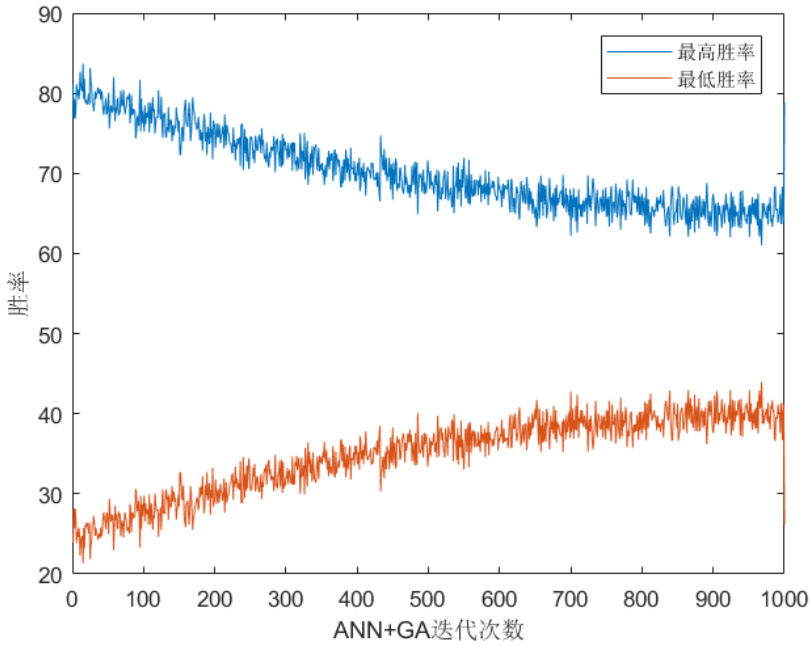


后来我也尝试使用了不同的激活函数，比如将激活函数替代为RELU，我们可以看出激活函数对于收敛结果的影响.其结果如下:



5.2 ANN+GA胜率变化

我也尝试了较多不同的遗传变异的策略，胜率最终基本都集中在65%左右，下面直接给出最优结果：



6 创新点

- **更新实验8评估函数**

  运用自己的理解，更新加强了实验8的评估函数，并取得了不错的效果：可以下赢中等难度的电脑(7k7k)

- **实现DQN**

  对于很少用神经网络的我来说，这次试验也加强了我对tensorflow 的理解

- **提出了ANN与GA的结合***

  在DQN的启发下，我将神经网络与遗传算法结合起来，使用遗传算法去更新神经网络的权值。这是一个很不错的创新点!

7 课程感想

本学期的课程基本到此为止了。作为计算机科学里较为有趣的一个分支科学，近年来随着深度学习的热度逐渐升高，人工智能夺得了很多人的关注。本学期的实验课从难度上讲并不很高，但是有些项目的实现还是需要非一些精力。作为本学期的最后一次实验报告，我特地写了几点感想：

- **从算法角度理解智能**

  在期中项目中，我主要实现了SVM算法，作为80-90年代的极为流行的算法，甚至如今在工业界也广为使用的SVM还是有必要了解一下的。SVM的数学原理实际上超出了我们本科两年的所学内容，想要彻底看懂需要一定的精力。但我还是抱着极大的兴趣将整个SVM不调用skitlearn的情况下运行了起来，并在二分类任务中取得了0.88的准确率.

  在当前的人工智能里，算法也许算是人工智能的基石，我认为了解SVM这样的算法也是比较重要的。

- **人类思维与算法融和**

  $\alpha - \beta$剪枝算法使用了博弈的思想，是受人类思维启发而提出的一种算法；此外最近流行的GAN生成对抗神经网络的思想，也是受了类似的启发而实现的，这或许体现了一种人与智能的联系。期末项目所用到的强化学习也是一种类人的方法，奖励函数机制的引入与生活中的奖励何其相似。这种‘仿生’ 思维的引入或许对人工智能起到了一定的影响。

人工智能学科的精髓或许是将人的思维机械化数据化，进而运用到实际生活中来，提升生产生活的效率。但前路也许是极其漫长的，费曼曾说“虽然人类以不同的方式切割自然，我们在不同的学科有不同的课程，我们在所能到达的地方享受乐趣就行了。”