本科生《人工智能基础》课程实践练习题

姓名：王欣哲

学号：1120182955

班级：07111809

时间：2021 年 1 月 12 日

问题一

一、问题分析

在对问题进行简要分析后，对于本问落子位置识别的任务，提出以下两种解决思路：

1、 根据落子前后两张棋盘图片的差别，即棋盘中棋子布局情况的差别检测落子位置

2、 根据当前的棋盘图片，准确识别棋盘中每一颗棋子的位置，得到棋子分布矩阵。通过程序比对前后两张图片所识别到的棋子分布矩阵，即可判定当前落子位置。

从实现方面看，方法 1 需要使用落子前后两张图片。如果直接将两张图片重叠，做异或操作，即可得到棋子的改变情况。但若考虑使用监督学习算法，则涉及到两张图片比对的操作。而方法 2 仅使用一张图片，只要保证对单张图片中棋子的识别准确率，也可以准确度判定出落子位置。因此，方法 2 相较于方法 1 较简单，且易于实现。

二、算法设计

1、 基本思路

利用 Tensorflow 框架，使用卷积神经网络模型进行训练。在定义好的网络上，通过定义交叉熵损失函数，评估网络对的预测效果，使用反向传播及梯度下降算法，对网络权值进行更新训练。神经网络以灰度图片作输入，以每个格点棋子状态的预测结果作输出。在查阅资料后，如文献[1]中所使用的方法，使用了目标检测的方法，将棋子识别至线框内。这种方法技术要求较高，且对于本简单任务来说较为复杂。因此最终确定使用简单的神经网络模型。

2、 网络设计

以经过粗略裁剪、大小为 96*96 像素的灰度棋盘图片作为输入，使用两层卷积神经网络、一层全连接层构建网络结构。输出层经全连接后，进行 softmax 并产生最后的输出。输出的形状为：[15*15，3]。按第一维度所取的每一个三元组，代表对应棋盘坐标位置上该棋子取黑、白、无三种情况的评估参数。取评估参数最大的参数所在的位置（即 0、1、2）代表此处棋子的颜色。

```
######################
#     卷积神经网络    #
######################
# acc=1
xs = tf.placeholder(tf.float32,shape=(None,96*96),name = "xs")
ys = tf.placeholder(tf.float32,shape=(None,15*15,3),name = "ys")
keep_prob = tf.placeholder(tf.float32,name = "keep_prob")
# ylab = tf.placeholder(tf.int64,shape=(None,15*15))

y0 = tf.reshape(xs,[-1,96,96,1])
# 第一层 卷积
w1 = tf.Variable(tf.truncated_normal(shape=[3,3,1,8], stddev=0.1))
b1 = tf.Variable(tf.constant(0.1,shape=[8]))
h1 =tf.nn.relu(conv2d(y0, w1) + b1)
y1 = max_pool_2_2(h1)

# 第二层 卷积
w2 = tf.Variable(tf.truncated_normal(shape=[3,3,8,16], stddev=0.1))
b2 = tf.Variable(tf.constant(0.1,shape=[1, 16]))
h2 =tf.nn.relu(conv2d(y1, w2) + b2)
y2 = max_pool_2_2(h2)

# 第三层 全连接
w3 = tf.Variable(tf.truncated_normal(shape=[24*24*16, 15*15*3],stddev=0.1))
b3 = tf.Variable(tf.constant(0.1,shape=[1, 15*15*3]))
h3 = tf.reshape(y2,[-1,24*24*16])
y3 = tf.nn.sigmoid(tf.matmul(h3, w3) + b3)

# Softmax
y =tf.nn.softmax(tf.reshape(y3,[-1,15*15,3]),name = "y")
```

第一卷积层，以(1,96,96)作输入，采用 3*3 的卷积核，共 8 个；池化使用 max_pool，大小为 2*2。输出为(8,48,48)。

第二卷积层，以(8,48,48)作输入，采用 3*3 的卷积核，共 16 个；池化层使用 max_pool，大小为 2*2。输出为(16,24,24)，经 reshape 为(16*24*24)后作为全连接层输入。

全连接层，以(16*24*24)作输入，输出为(1，15*15*3)的行向量，reshape 为(15*15,3)后作为 Softmax 的输入

Softmax 层，使用 Softmax 函数对数据作归一化处理得到最终输出，形状为(15*15,3)。

3、 损失函数与优化器

由于网络输出层使用了归一化的 Softmax，在预测中将棋子归一化的三个参数理解为该格点上三种棋子状态的概率，因此在查阅相关资料并结合相关实验后，最终确定使用交叉熵函数作为损失函数。损失函数值越小，网络的预测效果越好。

正确率函数用于评估棋盘格点的棋子状态预测正确率。按照输出的设计，取输出层每个三元组中最大的数的索引作为棋子状态与标签比对，并计算预测正确的格点的比例。

优化器采用 tensorflow 提供的 AdamOptimizer 优化器进行网络训练。

```
# 损失函数
cross_entropy = -tf.reduce_sum(ys*tf.log(y),reduction_indices=[1,2])
#cross_entropy = -tf.reduce_sum(ys*tf.log(y))
loss = tf.reduce_mean(cross_entropy)
#loss = tf.reduce_sum(tf.square(ys - y))
# 正确率
correct = tf.equal(tf.argmax(y,2),tf.argmax(ys,2))
acc = tf.reduce_mean(tf.cast(correct,tf.float32))

# 优化函数
train = tf.train.AdamOptimizer(1e-3).minimize(loss)
```

4、 训练方式

采用小批量梯度下降的方式对网络进行训练。从效果上看，每个 BATCH 的图片数量越大，网络的收敛速度越快，效果越好。BATCH 的图片数过少则会使得网络收敛效果变差。结合考虑个人机器的情况，确定 BATCH 为 200。

在训练集上训练一定的周期后，使用测试集对训练的效果作检测。这是为了防止网络对训练集的过拟合，在测试集上的效果可以反应网络模型的真实预测水平。

```
# 训练
BATCH_SIZE=256
batch_test = get_test_batch(1000)
for i in range(3000):
    # 生成一个Batch 的数据
    batch = get_train_batch(BATCH_SIZE)
    # 训练
    sess.run(train, feed_dict={xs:batch[0], ys:batch[1],keep_prob:1.0})

    if i%5==0:
        l = sess.run(loss, feed_dict={xs:batch[0], ys: batch[1],keep_prob:1.0})
        ac = sess.run(acc,feed_dict={xs:batch[0], ys: batch[1],keep_prob:1.0})
        res_train_loss.append(l)
        res_train_acc.append(ac)
        print("  time %d loss:%.5f  acc:%f" %(i,l,ac))

    if i%50==0:

        l = sess.run(loss, feed_dict={xs:batch_test[0], ys: batch_test[1],keep_prob:1.0})
        ac = sess.run(acc,feed_dict={xs:batch_test[0], ys: batch_test[1],keep_prob:1.0})
        res_test_loss.append(l)
        res_test_acc.append(ac)
        print("test  time:%d  loss:%.5f  acc:%f" %(i,l,ac))
```

```
  time 0 loss:234.03326  acc:0.486059
test  time:0  loss:237.86504  acc:0.454867
  time 5 loss:232.14783  acc:0.498403
  time 10 loss:224.81958  acc:0.523212
  time 15 loss:215.79324  acc:0.556181
  time 20 loss:214.11888  acc:0.576042
  time 25 loss:207.96591  acc:0.591319
  time 30 loss:207.03546  acc:0.588611
```

三、实际训练

1、 实验环境

语言：Python

神经网络框架：Tensorflow

开发环境：Anaconda、jupyter notebook、visual studio code

运行设备：联想小新 AIR15 IKBR 个人笔记本电脑

硬件：CPU Intel Core i5-8250U

GPU NVIDIA GeForce MX150

RAM 8G

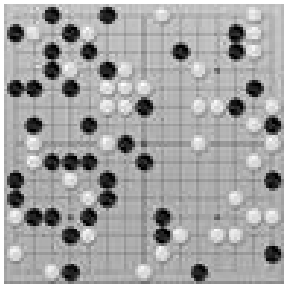
2、 数据集准备

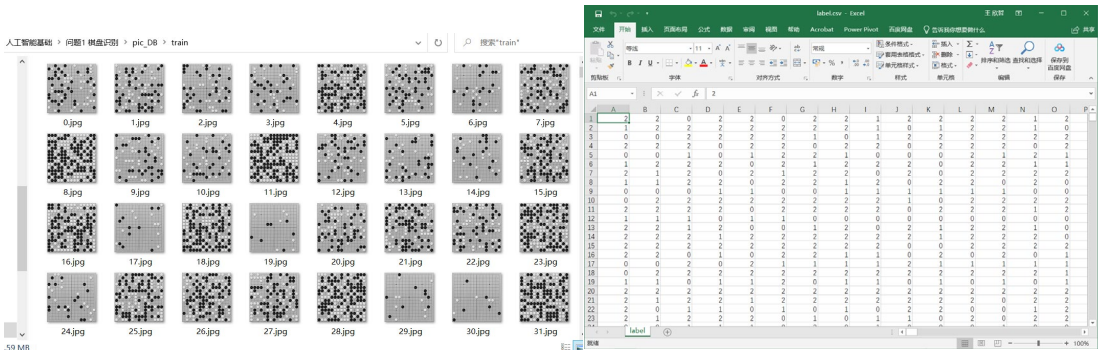
本任务使用的数据集图片，训练集 10000 张、测试集 1000 张。

由于在训练中需要使用大量的（棋盘图片，标签）数据对进行网络的训练，因此在训练前先完成对训练数据集的构建工作。使用网络上寻找到的五子棋图片素材，使用 Python 的 PIL 库将棋子图片准确叠加到棋盘上，并生成响应的数据标签文件。由于训练所用的图片输入为 96*96 像素的灰度图像，因此数据集中图片均设置为此大小。图片标签存储在图片数据的同一文件夹下，使用.csv 文件将标签按照行向量的方式存储，标签行号与图片的命名向对应。

由于棋子预测，需要保证对所有的格点进行充分的训练，因此棋盘中的棋子使用均匀随机分布的方式生成。

下图展示了数据集中图片的样例。数据集生成代码在程序文件夹中给出。





3、 模型训练

在模型训练时，需要将数据集的数据加载到程序中并组织成相应的格式，产生 BATCH，并送入神经网络进行训练。这里定义了两个函数，用于抓取训练集、测试集的数据。

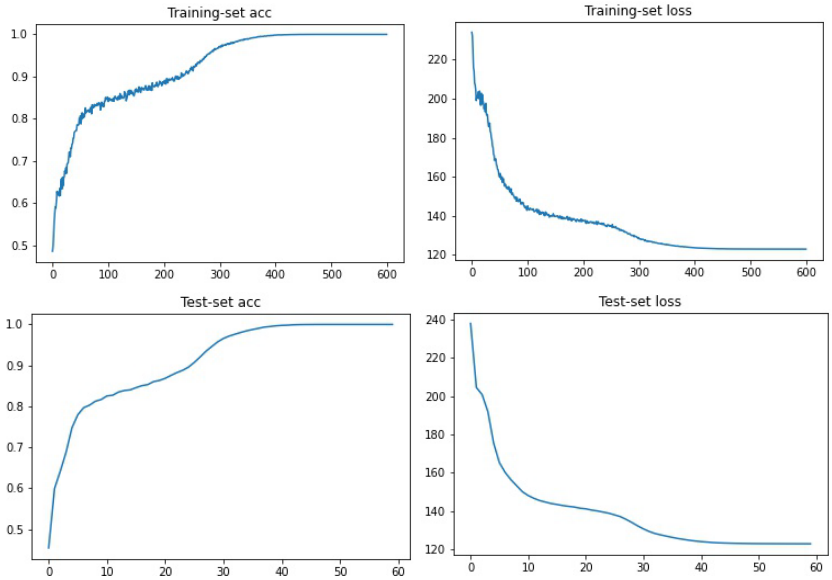
对于网络送入数据的格式，也需要按照一定的格式对数据标签进行重新构建。这里需要

将标签构建为(15*15,3)的标签矩阵，正确的标签数值置 0.99，其余置 0。对于图像，将 0~255 的取值范围映射到 0~1 的浮点数区间内。参考相关文献及代码资料中的做法，将颜色为 0 的像素点置 0.01。

同时，还需要记录网络训练过程的相关参数情况。除了打印的训练日志外，这里主要以网络训练过程中的 loss、acc 两个函数值作为监控指标。每 5 个周期抓取一次训练集的评估参数，每 50 给周期抓取一次测试集的评估参数。

为了提高模型的训练效率，使用 GPU 版本的 tensorflow 进行训练。在实际训练过程中，最终确定的网络模型可以在 3000 个训练周期中将正确率收缩至百分之百。

在训练完成后，使用 tensorflow 提供的参数保存接口，将模型保存，以供之后的调用。



从图中可以看到，网络在训练集、测试集上均有较好的效果。在训练的开始阶段，网络可以快速收敛；而在正确率 0.8 左右时，网络的收敛速度放缓；在达到 0.9 左右时，网络快速收敛至 0.99 的正确率，并继续收敛直至正确率为 1。在多次实验中，本网络的在训练集、测试集上的预测效果均可以达到 1。

```
 time 40 loss:198.99777  acc:0.627813
 time 45 loss:201.40103  acc:0.620729
 time 50 loss:200.42195  acc:0.626580
test  time:50  loss:204.46304  acc:0.599444

 time 2145 loss:123.16602  acc:0.999687
 time 2150 loss:123.12407  acc:0.999601
test  time:2150  loss:123.45884  acc:0.999547
```

这里注意到，loss 数值并未收敛至 1，而是稳定在了 120 左右。上图展示了模型第 50 周期、第 2150 周期在测试集上的表现。确定的网络的 loss 数值发生了异常情况，对这一情况将在第四部分中进行分析说明。

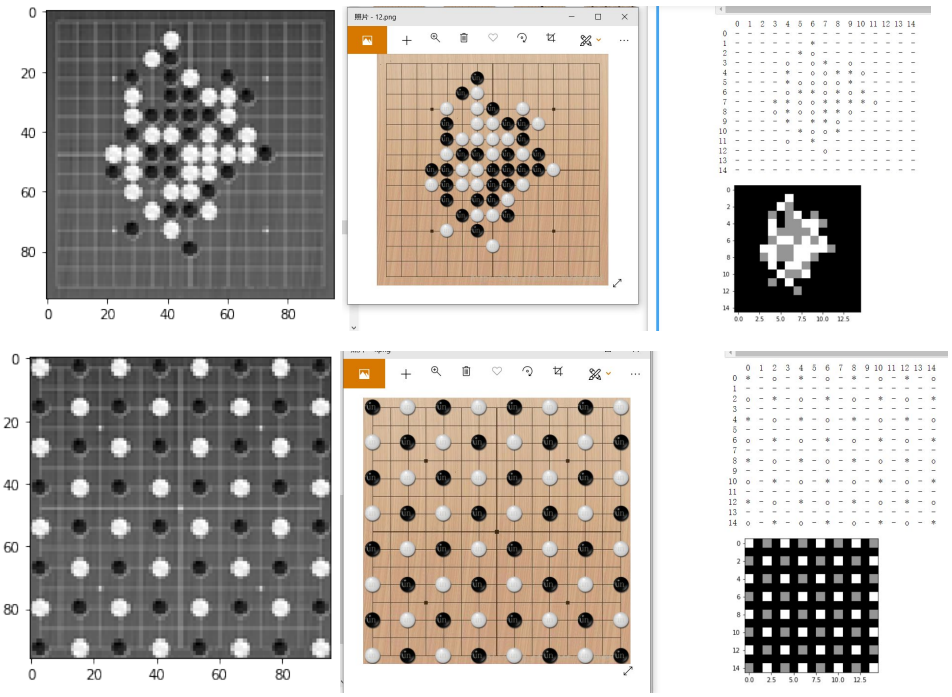
4、 模型使用

通过加载已经训练好的网络，在实际的情景下对网络的效果进行检验与使用。这部分由于不需要较高的计算能力，使用 tensorflow 的 CPU 版本进行预测。

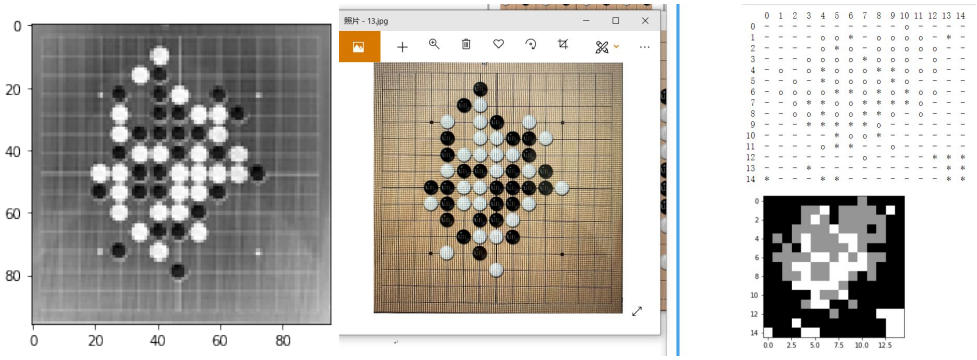
由于网络对于输入的数据格式有一定的要求，因此对任意输入的彩色图片，需要对其进行必要的处理。预测结果以数组进行输出。下图展示了图片加载、图片预处理、预测的代码。同时还计算了图片的预测时间等信息。

```
#加载图片，图像预处理为灰度图片
img = Image.open("./img/1.png").convert('L').resize((96, 96), Image.ANTIALIAS)
# 处理为np矩阵
img = np.asfarray((np.array(img)/255.0*0.99+0.01).reshape(1,96*96))

time_start = time.time()
infer_y = sess.run(y,feed_dict={xs:img,keep_prob:1.0})
time_end = time.time()

print("time = ",time_end-time_start)
print("FPS = ",1.0/(time_end-time_start))
plt.imshow(img.reshape(96,96),cmap='Greys',interpolation='None')
```

```
time =  0.003953695297241211
FPS =  252.92793825001507
```



上图是对电脑截屏的预测。从左至右一次为：预处理后图片、待识别原图、识别结果。经对比，神经网络对图片中的各个棋子均能实现正确的识别。对于集中分布的棋盘、随机散点分布的棋盘，均有较好的识别正确率。



上图是使用手机对电脑屏幕中的棋盘进行拍照后的识别效果。预处理后的图片中可以看件右较为明显的人影，这对模型产生了一定的干扰，使得预测结果中出现了大量噪点。这说明模型网络的泛化能力并不是很强。

由于选取的数据集为单一的、干净的图片，因此泛化能力不强是合理的。在进一步的改进中，可以使用带有噪声与干扰的数据图片对网络进行训练，以增强网络的泛化能力。

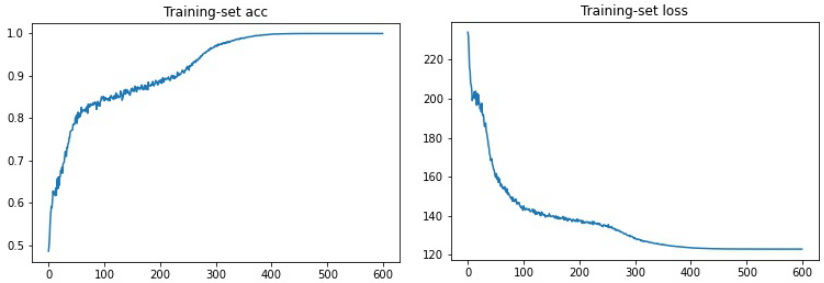
对于图片的边缘对齐，在多次实验后发现神经网络对边缘的对齐具有一定程度的容错。

四、实验结果分析

在本问题中，我先后尝试了多种不同的神经网络，包括单层感知机、多层感知机、卷积神经网络等。单层、多层感知机在实际的训练中，在较长时间的训练中，其预测准确率至多提升至约 60%，便不再提升。同时，这两种网络的参数两较为巨大，耗费了大量的存储。而卷积神经网络，在较短的训练时间后，其识别准确率就可以提升至 99%左右，甚至在对网络参数进行优化调整后训练得到了在训练、测试集上正确率为 1 的网络。同时，卷积神经网络的大小也较小。在优化保存为.pb 文件后，其大小可以压缩到 30MB 左右。

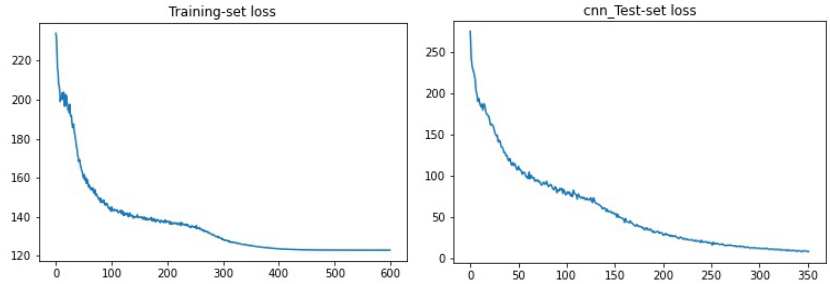
同时，在本次训练中，我也发现了一些有趣的问题。

1、 网络收敛速度



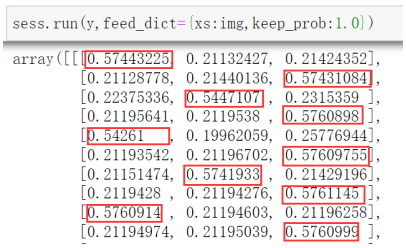
上图展示了训练过程的 acc 曲线及 loss 曲线。可以明显看到，acc 及 loss 曲线的变化可以划分为多个阶段。在 acc 为 0.5~0.8 时，acc 的提升速度较快，而进入 0.8~0.9 时，其提升速度变得缓慢，并且出现了明显的锯齿状波动。这说明网络可能进入了一个较为平缓的收敛区域，并进行较大范围内的搜索。而进入了 0.9~0.99 后，网络的收敛速度又出现了明显的加快，在接近 1 后收敛速度减小并最终稳定。

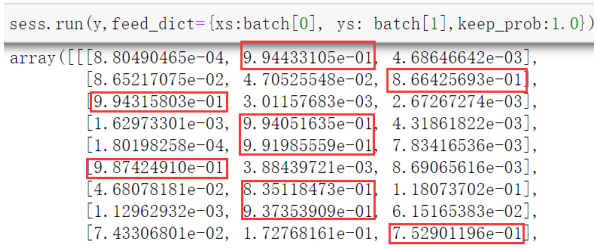
2、 loss 的稳定值



由于按照数据标签设定，每个棋子状态的三元组标签应为：(0.99,0,0)。因此，网络的预测结果也应尽量使得三元组中的一个数值更接近 1。在这种情况下，loss 应不断减小，直至逼近 0 为止。

在实验中，经过对卷积网络参数及网络觉狗不断调整，发现绝大多网络都符合这一特征，同时这些网络的预测准确率均在 99.8%左右。但有一组参数下的网络——即最终确定使用的网络，其训练的准确率可以达到 100%，而它的最终 loss 值稳定在 120 左右。通过对比其最终预测结果，其最终预测的三元组值稳定在(0.57,0.21,0.21)左右。下图左侧为 loss 异常的网络，右侧为正常的网络。





这种 loss 值较大且不再收敛，但预测结果却十分准确的现象较为奇特。异常的网络中最后两层的 sigmoid 激活函数与 softmax 函数相邻，因此有可能是这两种指数型函数相互作用产生了此种现象。具体的产生机理还需研究。

五、总结与改进

在本问题中，使用卷积神经网络可以对棋局识别任务达到较好的效果。对于干净的棋盘截图、不同风格的五子棋棋盘均能有较好的识别检测效果，具有一定程度的泛化能力。对于棋盘边界对齐的要求不高，有一定边界偏移的容错能力。

在网络方面，全连接结构网络对于此任务的效果欠佳，在长时间的训练后最终的预测准确率仅为 60%左右。而卷积神经网络，对此任务有极好的效果。由于本问题并不是复杂，因此在对网络的参数做不断的调整后，较小的网络的预测准确率也可以达到 99%。

而目前存在的问题是，对于拍照生成的，背景有噪声的图片，神经网络的识别准确率较低。这主要是由于训练数据集内容不够丰富造成的。因此，进一步改进措施在于丰富数据集的内容，使用有噪声的图片训练网络。同时本网络只能识别边界大致对齐的棋盘图像，对于边界偏移较大的图像，则可以通过结合其它边界裁剪的算法，完成对更一般的棋盘图像的识别与检测。

训练过程中出现的 loss 异常现象较为奇特，它的发生机制可以作为进一步研究的问题。

问题二

一、问题分析

本问使用一种博弈搜索算法，实现五子棋博弈程序。可以选择极大极小搜索结合 α - β 剪枝或蒙特卡洛搜索树。最终确定使用极大极小搜索结合 α - β 剪枝的方法来实现。棋局状态的判断函数，是五子棋棋力提升的关键。根据相关资料中的介绍，使用扫描法来编写器具状态判断函数。

二、算法设计

1、 极大极小搜索与 α - β 剪枝

在整个博弈程序中，博弈树搜索算法主要使用搜索过程中收集到的启发信息，来完成对棋局搜索的剪枝，并决定搜索的方向。极大极小搜索假定每轮双方落子时均会选择最利于自己的落子位置，并不断的交换落子。在搜索过程中，如果一种走法的最大估值小于已经发现的最大估值，则剪枝；对于另一方，如果一种走法的最小估值大于已经发现的最大估值则剪枝。这种搜索方式使得在搜索过程中可以省去很多不必要的搜索。

```
switch (board_cur.turn) {
case BLACK:
    if (val > alpha) {
        alpha = val;
    }
    if (alpha >= beta) {
        return alpha;
    }
    break;
case WHITE:
    if (val < beta) {
        beta = val;
    }
    if (alpha >= beta) {
        return beta;
    }
    break;
}
```

对同一棋盘不同招法的搜索顺序会使得剪枝的效果产生较大的差异。考虑五子棋周围棋盘对棋局贡献度较低的特点，确定搜索的顺序从棋盘中心点开始，向外围旋转展开。在实际实验过程中，此种搜索顺序可以使程序搜索的时间大大减少。以同样的棋局为例，若按照从左至右、从上到下顺序搜索的策略，在 10 秒内程序最多可以搜索 2 层；而采用此种中心展开的搜索方式，10 秒内程序可以搜索 4 层。

```
for (int i = 3; i <= 15; i += 2) {  //每一圈
    move_temp.x++, move_temp.y++;
    for (int dir = 0; dir < 4; dir++) { //四个方向
        for (int k = 0; k < i - 1; k++) {   //每颗棋子
            if (board_cur.board[move_temp.x][move_temp.y] == EMPTY) {
                Board board_new(board_cur); //拷贝新棋盘
                board_new.move(move_temp);  //走一步
                val = Alpha_Beta(board_new, alpha, beta, depth + 1);    //当前局面的搜索估值
```

2、 棋局状态判断函数

此函数主要用于为棋局进行估值。不妨设置评分为正黑方占优势，评分为负白方占优势。根据五子棋的特点，活二、活三、活四、死四对于棋局优势的影响依次递增。而有胜方的棋局，其估值的绝对值应尽可能大。根据五子棋的规则，需判断棋局内所有这些状态的数量并

对齐进行估值。

使用六元组搭配哈希估值表的方式进行估值。扫描棋局内所有的棋子，若非空，则对其所在的横、竖、左斜、右斜四个方向截取多个六元组，用六元组中对应的棋子状态在哈希表中映射取值，找到最大的一个。将四个方向取值最大的数的绝对值求和，作为估值。同时结合当前落子方的颜色，对六元组估值进行翻倍处理。这是为了增强落子方对棋局估值的影响。

```
//遍历每颗棋子
for (int i = 0; i < 15; i++) {
    for (int j = 0; j < 15; j++) {
        //空的点不搜索
        if (board_cur.board[i][j] == EMPTY) {
            continue;
        }
        //比遍历4个方向
        for (int d = 0; d < 4; d++) {
            //遍历5组连续的7颗棋子
            for (int k = -5; k <= -1; k++) {
                part = 0, part_real = 0, part_abs = 0;
```

```
int part_real = SCOREHASH[color[0]][color[1]][color[2]][color[3]][color[4]][color[5]];
//评分相加
if ((part_real > 0 &&
    part_real *= 2;
}
double abso = abs(part_real);
if (part_abs < abso) {
    part = part_real;
    part_abs = abs(part);
}
```

(字段) int ab::SCOREHASH[4][4][4][4][4][4]
棋形评分矩阵
联机搜索

rt_real < 0 && board_cur.turn == WHITE)

对于估值的哈希表，提前使用相应的算法构建，并存储在.txt 文件中。

```
//00000_    活五
if (a == 0 && b == 0 && c == 0 && d == 0 && e == 0) {   //黑棋
    SCOREHASH[a][b][c][d][e][f] = 1000000;
    continue;
}
//11111_    活五
else if (a == 1 && b == 1 && c == 1 && d == 1 && e == 1) {  //白棋
    SCOREHASH[a][b][c][d][e][f] = -1000000;
    continue;
}
//_0000_ 活四
else if (a == 2 && b == 0 && c == 0 && d == 0 && e == 0 && f == 2) {    //黑棋
    SCOREHASH[a][b][c][d][e][f] = 300000;
    continue;
}
```

三、实验

1、 实验环境

语言：c++

集成开发环境：Visual Studio 2019

运行设备：联想小新 AIR15 IKBR 个人笔记本电脑

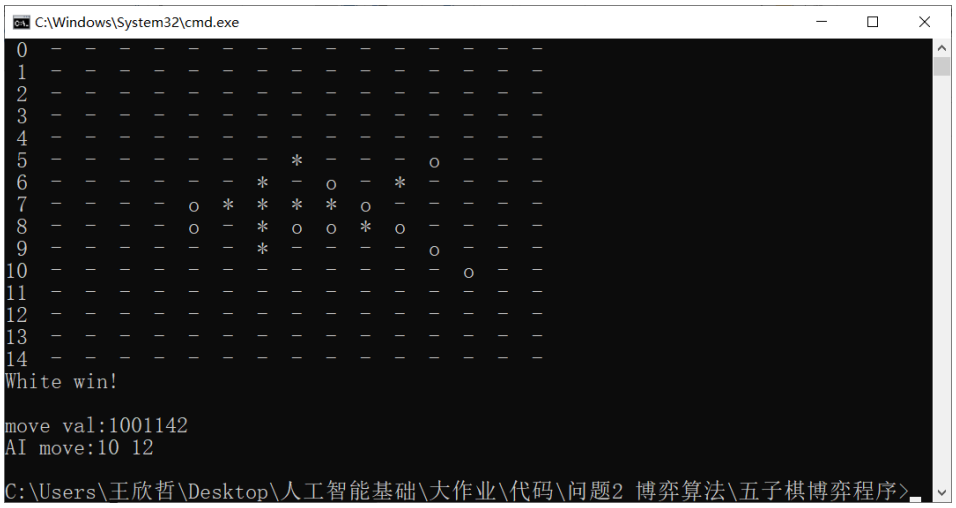
硬件：CPU Intel Core i5-8250U

GPU NVIDIA GrForce MX150

RAM 8G

2、 实验结果

经过对评估函数的若干个版本的迭代，五子棋的棋力得到了较大幅度的提升。对于活三、死四等必赌的情况，程序均能够及时封堵，并且具有一定的攻击性，能够主动落子出现活三定棋况。



四、实验结果分析

实验中所编写的五子棋博弈程序具有一定的棋力水平，在与人的对战中，取胜的概率较大。拿它和与手机上的五子棋程序对战，也能有不分伯仲的对战效果。因此综合来说，博弈程序取得了较好的效果。

在实际的对战测试中，我发现哈希估值表对于程序棋力的影响非常大。在不对参数做精确调整的情况下，程序不能很好的平衡活三、活四、攻击、防守之间的关系，经常犯错。而在精确调参后，程序能够很好的平衡各种棋况下的落子，展现出不易犯错、发挥稳定的特点。

从时间复杂度上来讲，随着棋局中棋子数目的增多，搜索的时间也会不断增长。使用 VS 性能分析器可以看到，程序的主要性能瓶颈在于评估函数。

五、总结与改进

极大极小搜索结合 α - β 剪枝与棋局估值函数的技术方案，对于五子棋博弈问题有着较好的实战效果。实验中所编写的博弈程序具有较高的棋力水平，并展现出不易犯错、棋力稳定的特点。

为了使得剪枝的效率更高，我采用中心展开式的招法生成策略，使得剪枝效率大幅提升，搜索深度可以达到 4 层。

在实验中我发现，棋局估值函数是提升程序棋力的关键。同时，棋局估值函数也是程序的性能瓶颈所在。因此进一步改进的方案主要在于对评估函数的改进。一方面是估值的准确度，对于哈希估值表，可以采用模拟退火、机器学习、进化学习等多种学习策略，对估值表的数值进行更新训练，使得估值更加准确，从而提高程序的棋力；另一方面是性能，可以进一步优化算法为多线程，使用 CPU 的多颗核心并行计算，从而使得搜索的深度更深，达到提升棋力的效果。也可以使用神经网络的方式实现估值函数。

问题三

一、问题分析

本问使用进化计算的方法对一个五子棋的估值网络进行训练。由于五子棋主要涉及到棋子之间的连续分布状态，即五子、四子、三子的线性特征，因此考虑使用卷积神经网络作为估值方法。通过对网络的不断重组和变异，以优胜略汰的方式训练网络，从而可能进化出能够准确评估棋局的网络模型。网络以 15*15 的棋盘做输入，输出为一个浮点数。

```
def conv_net():
    xs = tf.placeholder(tf.float32, shape=(None, 15*15), name="xs")
    ys = tf.placeholder(tf.float32, shape=(None, 1), name="ys")
    y0 = tf.reshape(xs, [-1, 15, 15, 1])

    # 第一层 卷积
    w1 = tf.Variable(tf.truncated_normal(
        shape=[3, 3, 1, 8], stddev=1),name="w1")
    b1 = tf.Variable(tf.constant(0.1, shape=[8]),name="b1")
    h1 = tf.nn.relu(conv2d(y0,  w1) + b1)
    y1 = max_pool_3_3(h1)   # 5*5*8

    # 第二层 卷积
    w2 = tf.Variable(tf.truncated_normal(
        shape=[3, 3, 8, 16], stddev=1),name="w2")
    b2 = tf.Variable(tf.constant(0.1, shape=[1, 16]),name="b2")
    y2 = tf.nn.relu(conv2d(y1,  w2) + b2)   # 5*5*16

    # 第三层 输出层
    w3 = tf.Variable(tf.truncated_normal(
        shape=[5*5*16, 1], stddev=1),name="w3")
    b3 = tf.Variable(tf.constant(0.1, shape=[1, 1]),name="b3")
    h3 = tf.reshape(y2, [-1, 5 * 5 * 16])

    y = tf.add(tf.matmul(h3, w3), b3, name="y")
    return xs, ys, y
```

二、算法设计

1、 网络设计

五子棋本身的判胜方法为同一直线上的棋子连续分布状态，而不同棋子在交叉线上相互作用对棋局优势方也会产生较大影响。卷积神经网络可以较好的提取方阵中的连续线性特征，因此最终使用卷积神经网络作为网络模型。

2、 进化算法设计

进化算法主要涉及到变异、重组、环境反馈三个部分。对于变异算子，通过对目标网络的网络参数叠加一层随机数值来实现；重组算子通过交换两个网络间部分网络层的参数来实现；环境反馈则通过网络之间的相互对局来实现。

在查阅相关资料后，决定使用进化规划策略。初始一定规模的种群，种群内进行重组、变异，生成若干新个体。通过网络间对局，来产生对网络的估值，并将评分低的网络删除。反复执行此过程，达到提升种群的环境适应水平

三、实验

1、 实验环境

语言：c++

集成开发环境：Visual Studio 2019

运行设备：联想小新 AIR15 IKBR 个人笔记本电脑

硬件：CPU Intel Core i5-8250U

GPU NVIDIA GeForce MX150

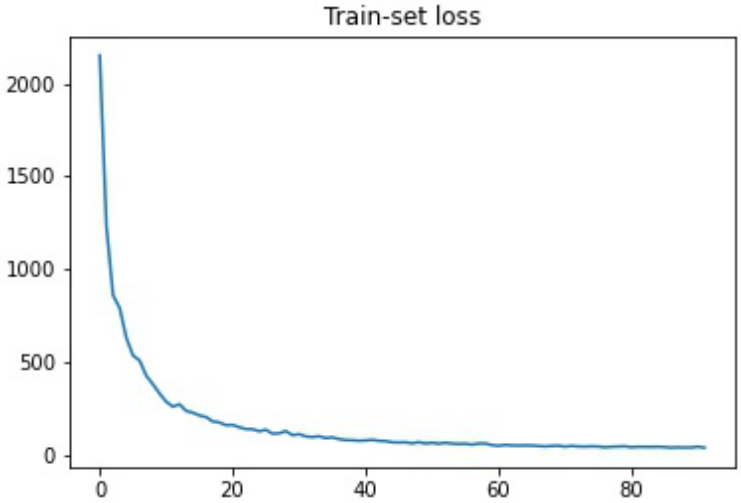
RAM 8G

2、 网络预训练

在经过简单的实验后发现，初始状态下，由于群体均没有任何知识，因此整个过程中均处于盲目落子的状态。在长时间的训练后，种群的棋力也没有任何的提升。在第 2 问中，我已经实现了一个具有一定水平的棋局评估函数，因此使用此评估函数对神经网络进行预训练。

使用大量随机生成的棋盘及评估函数，生成对棋盘的估值，并生成数据集。由于对局中，棋子应尽量靠近棋盘的中间，因此采用二项分布的随机数来生成棋盘。

使用反向传播和梯度下降的方法对网络进行预训练。这里同样使用 Tensorflow 框架进行实现。可以看到，在 4500 个训练周期后，网络较好的收敛。（这里每 50 个周期记录一个 loss）



3、 进化训练

首先定义变异、重组、环境反馈三个函数。

```
# 网络参数大致
# i 网络序号  sigma 变异程度
def variation(i,sigma=0.002):
    with graph[i].as_default():
        #卷积层1
        sess[i].run(tf.assign(w1[i],sess[i].run(w1[i])+tf.truncated_normal(shape=[3,3,1,8], stddev=sigma)))
        sess[i].run(tf.assign(b1[i],sess[i].run(b1[i])+tf.truncated_normal(shape=[8], stddev=0.1)))
        #卷积层2
        sess[i].run(tf.assign(w2[i],sess[i].run(w2[i])+tf.truncated_normal(shape=[3,3,8,16], stddev=sigma)))
        sess[i].run(tf.assign(b2[i],sess[i].run(b2[i])+tf.truncated_normal(shape=[16], stddev=0.1)))
        #全连接层
        sess[i].run(tf.assign(w3[i],sess[i].run(w3[i])+tf.truncated_normal(shape=[5*5*16, 1],stddev=sigma)))
        sess[i].run(tf.assign(b3[i],sess[i].run(b3[i])+tf.truncated_normal(shape=[1,1], stddev=0.1)))
```

```
# 环境反馈函数
board = ChessBoard()

def battle(i, j):
    print("battle i, j")
    board.reset()

    while True:
        board.judge_Win()
        if board.winner != ChessBoard.EMPTY:
            break
        if board.num >= 225:
            break
        pair = solve(board, i)
        print(pair)
        board.draw_XY(pair[0][0], pair[0][1])

        os.system("cls")
        printChess(board.board)
        print(pair[0][0], pair[0][1])

    if board.winner ==ChessBoard.BLACK:
        return 1, -1
    elif board.winner == ChessBoard.WHITE:
        return -2, 2
    else:
        return 0, 0
```

由于 tensorflow 本身不支持跨图的张量拷贝，因此通过将网络保存至外存后再加载的方式完成遗传算子。

4、 进化规划训练

使用进化规划策略对种群进行训练。在实际的训练过程中，程序的性能表现较差，群体并不能在长时间的训练中达到整体性能的提升。同时，受限于 tensorflow 无法高效拷贝网络的原因，网络的迭代次数较为缓慢，但周期的运行时间过长。

```
3 14
([4, 0], -0.5973872)
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 0  *  o  *  o  *  o  *  o  *  o  *  o  *  o  *
 1  o  *  o  *  o  *  o  *  o  *  o  *  o  *  o
 2  *  o  *  o  *  o  *  o  *  o  *  o  *  o  *
 3  o  *  o  *  o  *  o  *  o  *  o  *  o  *  o
 4  *  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 5  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 6  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 7  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 8  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 9  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
10  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
11  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
12  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
13  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
14  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
4 0
```

所有的个体均停止在初始状态无法前行。

四、实验结果分析

本问题中，使用进化计算的方法对人工网络进行训练。但是训练的效果并不好。即使在对网络进行预训练的情况下，种群依然始终停留在初始状态不产生进一步进化。这主要可能有四方面原因造成。

一是网络预训练的方法不科学，使用随机产生的棋盘-估值数据对对网络进行训练无法使得网络的水平提升至可以产生自举进化的程度。

二是算法涉及不合理，在算法训练过程中没有引入一定的随机过程，使得种群停留在初

始的局部最优状态，无法进一步运行。

三是算法运行低效，由于种群遗传需要使用外存进行交换，导致算法运行的效率过低，单批次迭代时间过长。

四是重组算子设计不合理，神经网络不像TSP问题，易于设计适合问题本身的重组算子。通过对网络参数简单交换来实现重组算子可能是不科学的。由于神经网络层级之间可能会有网络参数之间的依赖关系，这样盲目的交换可能使得网络无法进一步进化。

五、总结与改进

使用遗传算法对神经网络的训练效果较差。从算法设计角度看，本算法的设计有较多的不合理之处。而在进一步改进的方面，对于预训练，可以使用网上的公开的五子棋对局数据集对网络进行训练，以使网络获得正确有效的基本网络参数。对于随机过程的引入，可以在训练中引入经验池，随机抽取对局，在两个网络对战的过程中以一定概率随机落子。对于算法运行低效的方面，可以进一步优化算法，将各个神经网络组织在一张计算图内。而对于重组算子设计不合理，可以尝试使用网络参数均值或其它方法，进一步的改进需要基于实验做具体的尝试与改进。

问题四

一、问题分析

使用强化学习算法对神经网络模型进行学习。由于问题三中网络的实验效果欠佳，因此本问不在问题三的网络基础上做训练，使用强化学习 DQN 算法训练神经网络。同样使用卷积神经网络作为估值函数。通过引入随机过程和经验池，可以一定程度上解决问题三中出现的陷入局部最优解的问题。通过使用主网络和目标网络两个网络交替训练的方式，完成网络的训练。而对网络参数的更新，则以棋盘-Q 值数据对对网络进行反向传播与梯度下降，达到训练的效果。

二、算法设计

1、 网络设计

为了适合 DQN 算法，重新设计神经网络。以 15*15 的棋盘作为输入，通过一个 5*5 的卷积层、一层全连接层，得到最终的输出，输出层为 1*15*15。主网络与目标网络完全相同，目标网络是主网络若干代之前的版本。

```
def create_Q(self):
    # 网络权值
    W1 = self.weight_variable([5, 5, 1, 16])
    b1 = self.bias_variable([16])  # 5*5*16
    W2 = self.weight_variable([5*5*16+1, 225])
    b2 = self.bias_variable([1, 225])

    # 输入层
    self.state_input = tf.placeholder("float", [None, self.state_dim]
    self.turn = tf.placeholder("float", [None, 1])

    y0 = tf.reshape(self.state_input, [-1, 15, 15, 1])
    # 第一卷积层
    h1 = tf.nn.relu(self.conv2d(y0, W1) + b1)
    y1 = self.max_pool_3_3(h1)  # 5*5*16

    # 第二全连接层tf.concat([t1, t2], 0)
    h2 = tf.concat([tf.reshape(y1, [-1, 5 * 5 * 16]), self.turn], 1)
    self.Q_value = tf.matmul(h2, W2)+b2
    # 保存权重
    self.Q_weihgts = [W1, b1, W2, b2]
```

2、 DQN 类设计

为了使得训练过程中算法更加清晰，使用面向对象的方式，定义了 DQN 类。在类内封装了训练过程中所包含的各个算法。DQN 算法所使用的代码参考了相关 CSDN 博客中的代码流程与结构，网页链接在参考文献[2]中给出。对于网络部分，不再赘述，这里主要介绍所使用的 Q 学习策略。Q-learning 是在状态转移模型和已知收益函数未知时学习最优策略的模型无关方法。它的核心在于使用 maxQ(st + 1, at + 1)代替 V*(st + 1, at + 1)。对于状态和策略较为简单的问题，Q 学习能够很好的找出最优策略。在本问题中，正是使用神经网络作为 Q 函数，利用神经网络的强大你和功能，不断学习成为 Q 函数的替代品，从而实现对棋盘的准确估值。这里主要使用到的公式是：

$$Q(s,a) = r(s,a) + \Gamma \max(Q'(s',a'))$$

在训练中，目标网络进行的估值正是Q′。而 r(s,a)则是棋局胜利、和棋、非法落子所产生的反馈值。

除此之外，DQN 还使用经验回播技术来减少反馈的相关性。通过维护一个大小固定的经验池，将对局过程中的状态节点进行存储，并在其中进行随机选择生成 BATCH 的方式对

网络进行训练更新。这样可以减少反馈的相关性。

```
def perceive(self, state, action, reward, next_state, done):
    """添加经验池"""
    one_hot_action = np.zeros(self.action_dim)
    one_hot_action[action] = 1
    self.replay_buffer.append(
        [state, one_hot_action, reward, next_state, done])
    # 经验池满了
    if len(self.replay_buffer) > REPLAY_SIZE:
        self.replay_buffer.popleft()
    # 一个batch够了
    if len(self.replay_buffer) > BATCH_SIZE:
        self.train_Q_network()
```

3、 模型训练

对于网络，执行若干次迭代。每次迭代进行一局棋局的对战，在每一局的对战过程中，将每一步的落子所产生的棋盘及下一状态，作为一个状态节点投入经验池进行保存。当经验池中的数据量达到一定程度后，便在每次落子后抽取 BATCH 组数据，对网络进行训练。而每训练 100 轮次，将主网络参数拷贝至目标网络，进行参数的更新。

```
# 训练
for step in range(STEP):
    # 自己下一步棋
    action_1d = agent.egreedy_action(state)  # 有随机概率的走一步
    action_2d = [math.floor(action_1d / ChessBoard.SIZE), action_1d %
                 ChessBoard.SIZE, camp]  # 转化为二维棋盘坐标
    # 在模拟棋盘上落子
    next_state_2d, reward, done, _ = chess.draw_XY(
        action_2d[0], action_2d[1])
    next_state = np.reshape(next_state_2d, [-1])
    # 构造数据
    if step % 2 == 0:
        camp[0] = 1
    else:
        camp[0] = -1
    next_state = [next_state, camp]
    # 定义奖励
    reward_agent = reward
    # 丢入经验池  执行训练
    agent.perceive(state, action_1d, reward, next_state, done)
    state = next_state
```

三、实验

1、 实验环境

语言：c++

集成开发环境：Visual Studio 2019

运行设备：联想小新 AIR15 IKBR 个人笔记本电脑

硬件：CPU Intel Core i5-8250U

GPU NVIDIA GeForce MX150

GPU NVIDIA RTX2070

RAM 8G

2、 预训练

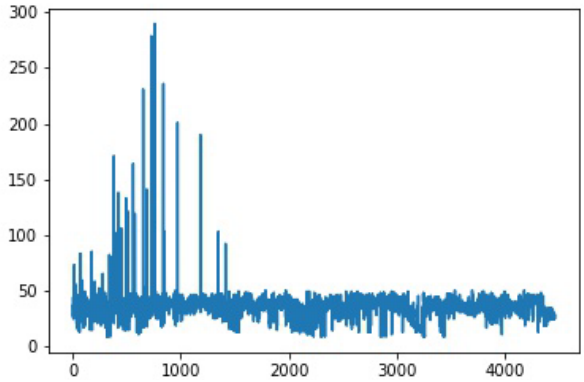
为了检验算法的正确性，我首先在 7*7 的棋盘上进行了小规模的实验。

```
   0  1  2  3  4  5  6
0  -  -  -  *  -  -  -
1  *  -  -  -  -  -  -
2  o  -  -  -  -  -  -
3  o  *  -  -  -  -  -
4  o  -  -  -  -  -  -
5  o  -  *  -  -  -  -
6  o  -  *  -  -  -  -

 WHITE 6   0
done step:9  episode:699
```

上图使第 699 个训练周期后，网络的自我对局情况。可以看到，白棋很快即可在对局中找到获胜的方法：连续下五颗棋子；而黑棋则处于随机落子的状态。

```
   0  1  2  3  4  5  6
0  -  -  -  *  -  -  -
1  o  -  -  -  -  -  *
2  *  -  -  -  -  -  -
3  o  *  -  -  -  -  -
4  o  -  -  -  -  -  -
5  o  o  *  -  -  -  -
6  o  -  *  -  -  -  -

 WHITE 5   1
```

上图是第 899 个训练周期后，网络的自我对局。此时，黑棋已经发现，左下角白棋的连续落子会使自己输掉，因此在(3,0)的位置落子。



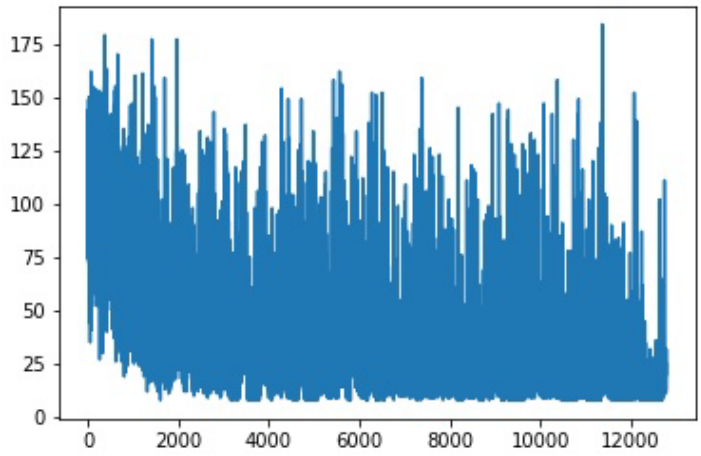
上图是 4000 个训练周期中，双方对局的步数情况。在前期，出现了落子数显著多于棋盘棋子数的情况。这说明程序中出现了 BUG。进一步分析，在超过 225 的步数中,有一定概率是因为棋盘没满，但是发生了碰撞也有一定情况是棋盘已满。对这种情况，可以在 action 函数中添加判断语句，即在所有 Q<0 时，判别棋盘已满跳出；也可以不做处理，处理至 300 步满，第二种选择应该更好，因为它可以强化模型对每一个棋盘的有子、无子的判断。另外，通过对步数曲线的观测，还可以判断模型非法落子发生的情况。因此最终选择不做任何处理。

2、 正式训练

在经过预实验后，在 15*15 的棋盘上进行正是训练。

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 0  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 1  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 2  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 3  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 4  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
 5  -  -  -  -  -  -  *  -  -  -  -  -  -  -  -
 6  -  -  -  -  -  -  *  -  -  -  o  -  -  -  -
 7  -  -  -  -  -  -  *  o  -  o  -  -  -  -  -
 8  -  -  -  -  -  -  *  -  o  *  -  -  -  -  -
 9  -  -  -  -  -  -  *  o  -  o  -  -  -  -  -
10  -  -  -  -  -  -  -  -  -  *  -  -  -  -  -
11  -  -  -  -  -  o  -  o  -  *  -  -  -  -  -
12  -  -  -  -  -  -  -  -  -  *  -  -  -  -  -
13  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
14  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
BLACK 5   6
```

上图使经过 12822 个训练周期后的对局情况。在对局过程中，黑棋、白棋均已发现连续落子至五颗即可获胜这一策略。可以看到黑、白双方都在尽力落满五颗子。而由于黑棋先行的优势，黑棋率先获胜。同时也看到，虽然白棋有围堵黑棋的迹象，但黑棋、白棋之间的互动仍然较少。但这仍然说明，神经网络成功学习到了简单的五子棋规则，有了一点点的棋力水平。而相信在更加充分的训练后，双方的棋力都会得到大幅度的提升。



上图是 12788 个训练周期中，双方对局的步数情况。可以看到在前 2000 局对局中，双方的对局步数不断下降，最小值稳定在 15 步左右。这说明神经网络学习到了取胜的方式，并能够在较少的步数中赢得胜利。

四、实验结果分析

本实验中，通过使用卷积神经网络和 DQN 算法，训练了一个具有初步棋力水平的局面估值网络。在其自我对局中可以发现，网络对基本的五子棋规则已经有了一定的识别能力，但是对于落子双方间棋子的相互作用并没有学到太多。从整体上来说，网络取得了一定的对局能力。如果结合第二问中的博弈搜索算法，相信网络会有更好的效果与表现。

五、总结与改进

使用 DQN 算法对五子棋局面估值神经网络进行训练是一条正确的、初步能走的通的路。在实际的实验中，训练得到的网络具有了初步的对局能力。在缺陷方面，一个显而易见的缺陷是训练速度过慢。在初步训练中，我发现自己机器的性能不足，因此借用了朋友的电脑，但在进一步训练中，其训练的迭代速度依然很慢。而在训练程序中，可以优化的点依然很多，因此要解决训练速度的问题，可以通过优化程序、调成更优的程序参数来解决。

对于收敛速度缓慢这一问题，则可以对神经网络进行预训练，使其具有一定的水平基础，

再进行强化学习。再相关资料的介绍中，AlphaGO 便是先对策略神经网络进行监督学习后，再进行强化学习。

本问题的难点主要在于对强化学习思想的理解。在理清了整个算法的流程、思路后，程序编写也会较为简单。

总结

在本次课程作业中，我使用 Python、Tensorflow 框架、C++语言，通过各种不同的技术路线实现了对五子棋博弈程序的设计。在学习和实验中，通过查阅各类资料，我学会了大量的有关人工智能的知识和技术，并从实践出发，编写了一些程序进行实际的训练。在实验中我发现，无论是监督学习、遗传学习、强化学习，它们都有自身的特点。对于特定的问题，使用特定合适的方法才能够取得最优的结果。而如果选择了不恰当的技术路线，无论功夫再大，都很难出成果。这就要求我对各类机器学习、人工智能算法有较为广泛的了解，在面对具体问题时，一定是在完善分析的基础上选择最科学、合理的方法进行问题的求解。

通过本次课程作业，我也学习到了许多的知识，我对于人工智能、机器学习也有了更加深刻的理解。能够利用技术解决一个简单的问题，便是我本次作业最大的收获。通过这次作业，我知道自己仍然有很多的知识需要学习。只有在掌握更多的方法的基础上，在面对问题时才能选择最正确的道路。

参考文献

[1]刘帅,郭滨,陈亮,张晨洁. 基于 SSD 的棋子检测算法研究[J]. 长春理工大学学报(自然科学版),2019,42(06):67-72.

[2] CSDN 博客 基于 DQN 的五子棋算法

https://blog.csdn.net/x_studying/article/details/80527498