基于蒙特卡洛搜索树与自对弈的五子棋模型

刘畅 黄奎源 徐子航

小组分工

刘畅：训练框架修改、训练调试、GUI界面设计、报告编写

黄奎源：训练框架搭建、初步训练

徐子航：相关背景调研、汇报

概述和相关工作

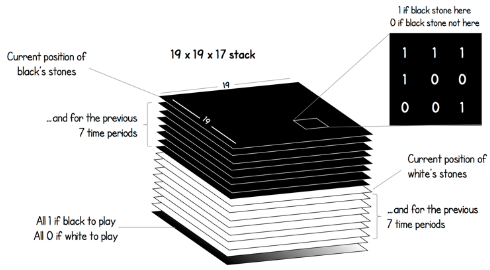
AlphaGo Zero是战胜李世石的AlphaGo的升级版本，在与其的对战中取得了100胜0负的成绩。二者都是基于蒙特卡洛搜索树进行动作的选择，AlphaGo Zero最显著的不同是它没有使用人类棋谱进行训练，而是借助蒙特卡洛搜索树，使用自对弈的方式自行产生棋谱进行训练。除此以外，在网络结构上也进行了一定的改进，在下文中会有所提及。

AlphaGo Zero被应用于围棋上，它是一个零和博弈问题，无论规则如何，一方的胜利必将导致另一方的失败，令胜利方奖励为1，则失败方为-1。此方法的一个特点是模型不需要掌握规则，只需要能够判断当前是否处于终结状态，并且能够判断获胜者即可。因此，模型框架可以广泛地应用于各类棋类游戏中。

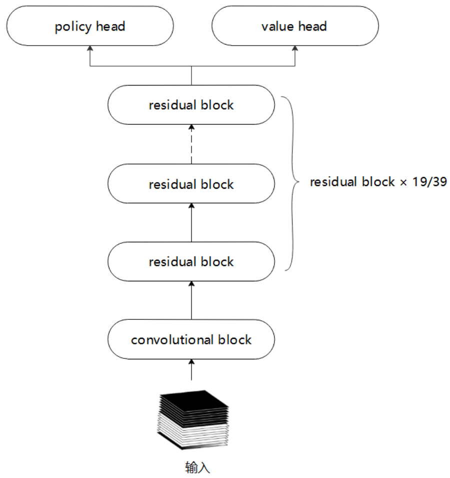
相较于围棋，五子棋的棋盘更小，具有更简单的状态空间，蒙特卡洛搜索树的深度也更小。因此我们参照AlphaGo Zero的基本原理，针对五子棋的特点进行一些修改，用来训练一个能够与玩家交互的五子棋模型。在模型训练完毕后，我们使用PyQt5搭建了一个简单的可视化界面，通过点击棋盘相应位置与模型进行交互，实现对弈。

网络结构

对于AlphaGo Zero，其输入如下图所示，尺寸为19×19×17，包含17层棋盘的数据，其中包含玩家与对手的最近8次落子状态，以及一层代表当前轮到黑子或白子，输出为362维的向量，分别表示在棋盘各处落子或不落子的可能性。针对五子棋的特点，输入尺寸为11×11×6，棋盘尺寸为11×11，记录玩家和对手前三步的状态，移除了代表先手的一层。输出也相应调整为121维，只保留了在棋盘上落子的概率。



AlphaGo Zero相比AlphaGo，除自对弈外的另一个主要改进是修改了网络结构，不仅引入残差层提高网络性能，并且将分离的policy网络与value网络进行整合，在共同的网络（称为特征提取器）上使用一个双头结构分别输出估值结果与策略选择。网络整体结构如下所示，先后经过卷积层，残差层获得特征图像，再分别使用策略网络和估值网络获得结果。



网络设置和修改如下：

- 卷积层由卷积层、批归一化层和ReLU 函数组成，通过卷积层将19×19×17的输入转化为19×19×256，其中卷积核大小为3×3。为降低网络的复杂度，我们将其中的输出通道减少为128，尺寸为11×11×128。

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels: int, out_channel: int,
kernel_size, padding=0):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channel,
                              kernel_size=kernel_size,
padding=padding)
        self.batch_norm = nn.BatchNorm2d(out_channel)

    def forward(self, x):
        return F.relu(self.batch_norm(self.conv(x)))
```

- 引入残差层是AlphaGo Zero的另一项改进，能够有效提升网络的特征提取能力并防止出现梯度消失问题。残差层由卷积层以及 19 个或 39 个Residual block组成，其中每个Residual block由2个类似于加上了跳连接的两个卷积层组成，使输入与批归一化模块的输出相加再输入ReLU 函数，最终输出 19×19×256 的特征图像。在我们的网络中，将Residual block设置为4个，并且同上输出尺寸为11×11×128。

```
class ResidueBlock(nn.Module):
    def __init__(self, in_channels=128, out_channels=128):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.batch_norm1 =
nn.BatchNorm2d(num_features=out_channels)
        self.batch_norm2 =
nn.BatchNorm2d(num_features=out_channels)

    def forward(self, x):
        out = F.relu(self.batch_norm1(self.conv1(x)))
        out = self.batch_norm2(self.conv2(out))
        return F.relu(out + x)
```

- Policy head使用残差层输出的特征图像，经过内部的卷积层、批归一化层和全连接层的处理之后，经过softmax函数得到维度为362维的概率向量，分别表示在棋盘各位置落子和不操作的概率。对于五子棋的情景，输出的维度为121，因为五子棋没有不操作的选项。

```
class PolicyHead(nn.Module):
    def __init__(self, in_channels=128, board_len):
        super().__init__()
        self.board_len = board_len
        self.in_channels = in_channels
        self.conv = ConvBlock(in_channels, 2, 1)
        self.fc = nn.Linear(2*board_len**2, board_len**2)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc(x.flatten(1))
        return F.log_softmax(x, dim=1)
```

- Value head同样使用残差层输出的特征图像，包括两个全连接层：第一个全连接层将输入映射为256维的向量，第二个全连接层再将 256 维的向量变为标量，最后经过tanh函数映射得到[-1, 1]区间内的值。在我们的网络中，第一个全连接层将输入映射为128维。

```
class ValueHead(nn.Module):
    def __init__(self, in_channels=128, board_len):
        super().__init__()
        self.in_channels = in_channels
        self.board_len = board_len
        self.conv = ConvBlock(in_channels, 1, kernel_size=1)
        self.fc = nn.Sequential(
            nn.Linear(board_len**2, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.conv(x)
        x = self.fc(x.flatten(1))
        return x
```

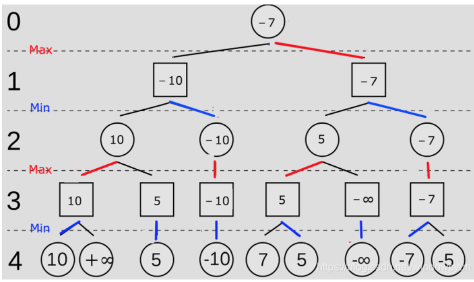
网络的实现如下：

```
class PolicyValueNet(nn.Module):
    def __init__(self, board_len=common.size, n_feature_planes=11):
        super().__init__()
        self.board_len = board_len
        self.n_feature_planes = n_feature_planes
        self.conv = ConvBlock(n_feature_planes, 128, 3, padding=1)
        self.residues = nn.Sequential(*[ResidueBlock(128, 128) for
i in range(4)])
        self.policy_head = PolicyHead(128, board_len)
        self.value_head = ValueHead(128, board_len)

    def forward(self, x):
        x = self.conv(x)
        x = self.residues(x)
        p_hat = self.policy_head(x)
        value = self.value_head(x)
        return p_hat, value
```

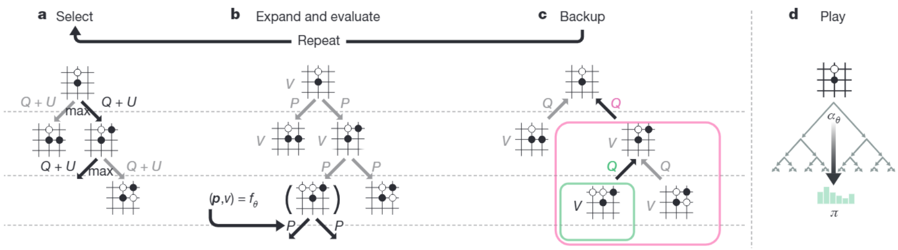
蒙特卡洛搜索树

在较为简单的零和博弈游戏中，常使用的方法是极小极大算法(Minimax)，其基本假设是玩家希望自身得到最大效益，而对手希望玩家得到最小的效益。示意图如下，从当前状态建立决策树，根据胜负情况对叶节点估值。在对手层(Min)，选择效用最小的节点动作；在玩家层(Max)，选择效用最大的节点动作，从而递归获得最佳动作。由于Minimax状态需要遍历所有节点，在深度较大时效率较低，因此提出了alpha-beta剪枝，在确认节点不会被选中后，跳过尚未遍历的子节点，从而提高了效率。



极大极小值算法在象棋、黑白棋等传统游戏中取得了巨大的成功，游戏本身的探索空间相对较小，使得探索能够到达较深层甚至底层。除此以外，这些游戏的指向性较为明显，可以设计出具有指向性的评价函数，从而引导局面向预期的方向发展。但是Minimax算法在围棋上难以应用，原因与上面相对应，首先围棋状态较为复杂，每一步都有362-N（N为棋盘上的棋子）种可能，搜索空间很大，限制了搜索的深度。另外，对围棋中的每一步难以设定明确的价值，从而无法设计较好的评价函数。为了解决极大极小算法的难题，蒙特卡洛搜索树被提出。

在 19×19 的棋盘上，要穷举出接下来的所有走法是不太现实的一件事，所以 AlphaGo 系列都使用了蒙特卡洛树搜索（MCTS）算法。如下图所示，AlphaGo Zero 的 MCTS 包含四个步骤，分别是：选择、拓展与评估、反向传播和演绎，如下图所示。



详细的步骤如下：

- 选择

  搜索树上的的节点表示棋盘状态，节点间的边包含以下的数据：

  - P(s,a)代表从父节点s进行动作a后到达子节点的先验概率；
  - N(s,a)代表对子节点 的访问次数；
  - Q(s,a)代表子节点上的累计平均奖赏；
  - U(s,a)代表在子节点上应用上限置信区间算法得到的值，公式如下

$$U(s,a) = c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

所示，其中Cpuct为探索常数，它的值越大，就越有可能探索未被访问或者访问次数较少的子节点；

在蒙特卡洛搜索树的每一轮搜索中，搜索都从根节点出发，根据U+Q的最大化选择动作到达子节点，接着重复上述步骤直到到达叶节点或游戏结束。

- 拓展与评估

当在选择过程中遇到叶节点，但节点对应的状态不是终止状态时，叶节点对应的棋盘状态输入策略-价值网络，神经网络对棋局进行评估后得到移动概率向量p和当前玩家获胜的概率v。移动概率向量p将用来拓展叶节点，p中的每一个元素分别对应一个子节点的先验概率 P(s,a)，并且将所有新节点的访问次数初始化为0。

- 反向传播

在拓展与评估步骤中我们得到了叶节点对应的玩家的获胜概率v，反向传播就是指将这个v传播到从叶节点向上传播到根节点，我们可以使用递归做到这一点，在路径上的每一个节点上，执行一次更新。

- 演绎

当我们完成n次搜索后，接下来根据根节点的各个子节点的访问次数 N(s,a)，计算选择动作a的概率：

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

其中τ为温度常数，最后根据每个节点的π来随机选择一种动作并在棋盘上执行。温度常数越小，越有趋向于选择π最大的动作，即越趋近于贪婪；而温度常数越大，越趋近于探索。

```
class Node:
    def __init__(self, prior_prob, c_puct, parent):
        self.Q = 0
        self.U = 0
        self.N = 0
        self.score = 0
        self.P = prior_prob
        self.c_puct = c_puct
        self.parent = parent
        self.children = {}

    def select(self):
        return max(self.children.items(), key=lambda item:
item[1].get_score())
```

```python
    def expand(self, action_probs):
        for action, prior_prob in action_probs:
            self.children[action] = Node(prior_prob, self.c_puct, self)

    def __update(self, value):
        self.Q = (self.N * self.Q + value)/(self.N + 1)
        self.N += 1

    def backup(self, value):
        if self.parent:
            self.parent.backup(-value)
        self.__update(value)

    def get_score(self):
        self.U = self.c_puct * self.P * sqrt(self.parent.N)/(1 + self.N)
        self.score = self.U + self.Q
        return self.score

    def is_leaf_node(self):
        return len(self.children) == 0
```

训练过程

AlphaGo Zero相比于AlphaGo的其中主要区别在于训练过程，AlphaGo需要人类棋谱作为参照进行训练，而AlphaGo Zero则从零开始，完全依靠自对弈训练模型。

在每轮训练中，我们都进行500次自对弈，每一次自对弈的过程如下：

1. 清空棋盘，初始三个列表`pi_list`、`z_list`、`feature_planes_list`，分别记录每个动作对应的概率，每个动作对应的激励，以及棋盘状态；
2. 将当前的棋盘状态加入`feature_planes_list`，并且根据前文所属的方法，进行蒙特卡洛树搜索，获得动作概率向量，加入`pi_list`；
3. 根据动作概率向量挑选动作，更新棋盘，检查是否结束，若未结束则继续执行2；
4. 根据最终的赢家，更新`z_list`中的元素，若对应步是赢家所下，则其激励值为1，若为输家所下则为-1，若是平局则值为0；

五子棋与围棋类似，都具有旋转不变性以及镜像对称性，因此对于自对弈获得的每组数据，都可以通过旋转和镜像分别获得4组数据，一共可以得到8组数据。另外为了在自对弈中兼顾探索性和后期的稳定性，自对弈前 30 步的温度常数为1，后面的温度常数趋于无穷小。参考原文，为了增加探索，在自定义时对动作概率向量添加满足狄利克雷分布的噪声。

$$P(s,a) = (1-\epsilon)p_a + \epsilon\eta_a, \eta_a \sim Dir(0.03), \epsilon = 0.25$$

在获取一定大小的数据集后，在这里设置为500组，即进行63次自对弈后，进行训练。在每一次训练之前，均通过一次自对弈产生新的数据，训练步骤如下：

1. 从数据集中随机抽出大小为`batch_size`的样本集；
2. 将样本集含有的 `feature_planes_list` 作为一个 `mini_batch` 输入策略-价值网络，输出维度为 `(batch_size, 121)` 的动作概率向量对数值和`(batch_size, 1)` 的估值；
3. 根据损失函数更新神经网络的参数，公式如下，其中c是控制 L2 权重正则化水平的参数；

$$l = (z-v)^2 - \pi^T logp + c||\theta||$$

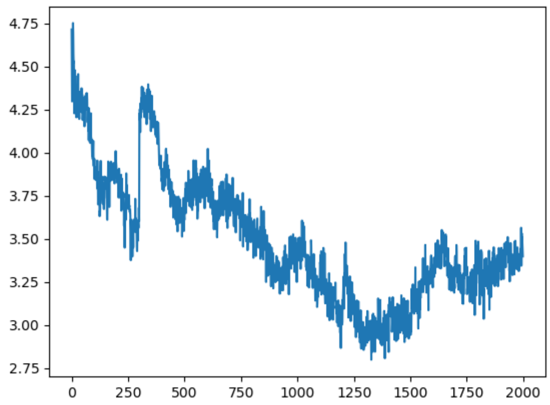
4. 每隔一定周期将当前模型与历史最佳模型进行对比；

```
    def train(self):
        for i in range(self.n_self_plays):
            self.dataset.append(self.__self_play())
            if len(self.dataset) >= self.start_train_size:
                data_loader = iter(DataLoader(self.dataset,
self.batch_size, shuffle=True, drop_last=False))
                self.policy_value_net.train()
                feature_planes, pi, z = next(data_loader)
                feature_planes = feature_planes.to(self.device)
                pi, z = pi.to(self.device), z.to(self.device)
                for _ in range(5):
                    p_hat, value =
self.policy_value_net(feature_planes)
                    self.optimizer.zero_grad()
                    loss = self.criterion(p_hat, pi,
value.flatten(), z)
                    loss.backward()
                    self.optimizer.step()
                    self.lr_scheduler.step()
                self.loss_record.append(loss.item())
            if (i+1) % self.check_frequency == 0:
```

```
                self.__test_model()
```

其中与AlphaGo Zero略有不同的方面在于，五子棋模型的自对弈数据使用最新模型产生，而AlphaGo Zero则使用历史最佳模型产生，这主要考虑到频繁比较模型优劣带来的时间损耗。另外为保存最佳模型，参考AlphaGo Zero的方法，定期对比当前模型和历史最佳模型，让它们对弈，若当前模型的胜率超过55%，则更新历史最优模型。

下图为训练2000次的loss曲线变化，可以看到前期loss值以较快速度下降，而后进入相对稳定的状态，虽然loss值没有相对下降，在后期反而略微提高，但在与此前模型的对战中，仍然能以较高的胜率取得胜利。



可视化对弈过程

至此，我们获得了一个五子棋模型，如上所属，它接受11×11×6的输入尺寸，输出对当前状态的评估值，以及执行各个动作的概率向量。接下来，我们构建了一个简单的可视化界面，使玩家能够与模型对弈，完成交互过程。

GUI界面的设计使用PyQt5进行，由于任务较为简单，因此使用QWidget即可，部分功能的实现过程如下：

- 背景：Widget的背景设置为棋盘图片，根据棋盘尺寸值选取相应的棋盘图片；
- 点击：鼠标点击图像上特定位置，换算为棋盘坐标，在此处添加一个QLabel，插入对应棋子图片；
- 鼠标跟随：重载鼠标移动函数，让一个棋子图片跟随鼠标移动；

在实现基本功能后，就可以构建整体程序逻辑是：

1. 玩家点击图像相应位置，获得在UI中的位置；
2. 计算距离该点的最近点位，在棋盘中检查目标位置是否合理，若不合理，返回步骤1；
3. 则在棋盘中更新相应位置数据，插入玩家棋子图像；

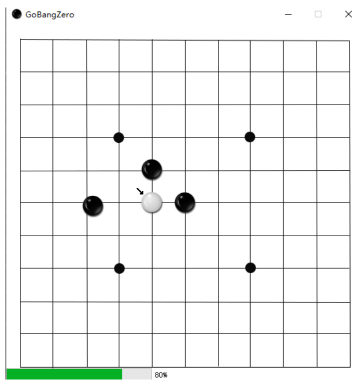
4. 判断本局是否终结，若终结则跳转到步骤8；
5. 从棋盘处导出包含过去状态的11×11×6数据，通过预训练模型和蒙特卡洛搜索树获得下一步决策；
6. 在棋盘中更新对应位置数据，插入AI的棋子图像；
7. 判断本局是否终结，若终结则跳转到步骤8，否则回到步骤1；
8. 根据不同的胜负状况弹出不同的对话框，选择是否继续进行游戏，若选择进行，则清空所有状态，回到步骤1；

虽然整体逻辑比较简单，但在实际测试中我们主要发现了两个问题，一是如果使用单线程，在点击后会出现一定时间锁死程序的情况，二是模型搜索时间较长，玩家在搜索过程中难以判断是程序出现问题还是在进行搜索过程。

对于单线程问题的改进，我们使用了一个额外的线程进行搜索，在完成搜索后通过信号槽发送结束消息，不影响主线程的进行。在搜索过程中，棋子图像跟随鼠标移动，但是在点击时，由于上一次落子颜色与当前一致，因此不进行后续步骤。

对于搜索时间较长的问题，我们首先尝试减少蒙特卡洛搜索书的搜索次数，这样确实能加快搜索速度，但是得到的结果并不理想，会出现不合理的下棋位置。因此我们添加了一个进度条，从蒙特卡洛搜索树处获得当前搜索的进度，并以一定的频率更新，这样能给用户带来一个合理的心理预期。

最终的GUI界面如下，选择的棋盘尺寸为11×11，其中黑棋为玩家，白旗为模型。



总结

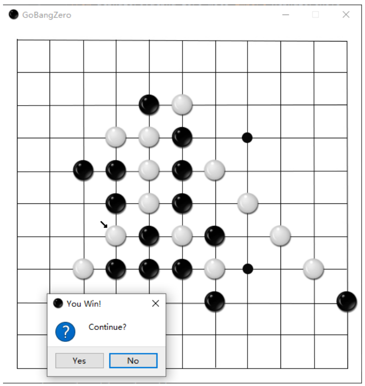
在本项目中，我们尝试把AlphaGo Zero的基本思想应用于五子棋游戏中，经过一段时间的训练后，能够得到一个策略较为合理的模型。并且我们还搭建了一个简单的可视化界面，让玩家可以与模型交互，直观检验训练成果。

由于没有规则限制的五子棋存在先手必胜的策略，因此我们参考竞赛五子棋的规则，添加了禁手限制，在黑棋（先手）落子后形成双活三、双四或长连三种棋型时，判定黑子负，从而提高了游戏的可玩性。

当然，我们在开发的过程中也遇到了一些问题：

- 训练速度较慢，在蒙特卡洛搜索的过程中消耗较长时间；
- 支持的棋盘尺寸较小，棋盘尺寸增加到13及以上时训练速度太慢；
- 对弈过程中难以做到实时响应，每一步都需要经过一段时间的运算；

总体而言，能够完成基本任务，我们也与模型进行了许多对弈，如果对五子棋不是很了解，很难取胜，下面是一张玩家获胜的棋谱：



参考资料

AlphaGo Zero算法复现 https://github.com/junxiaosong/AlphaZero_Gomoku

PyQt5可视化设计 https://github.com/ColinFred/GoBang#gobang