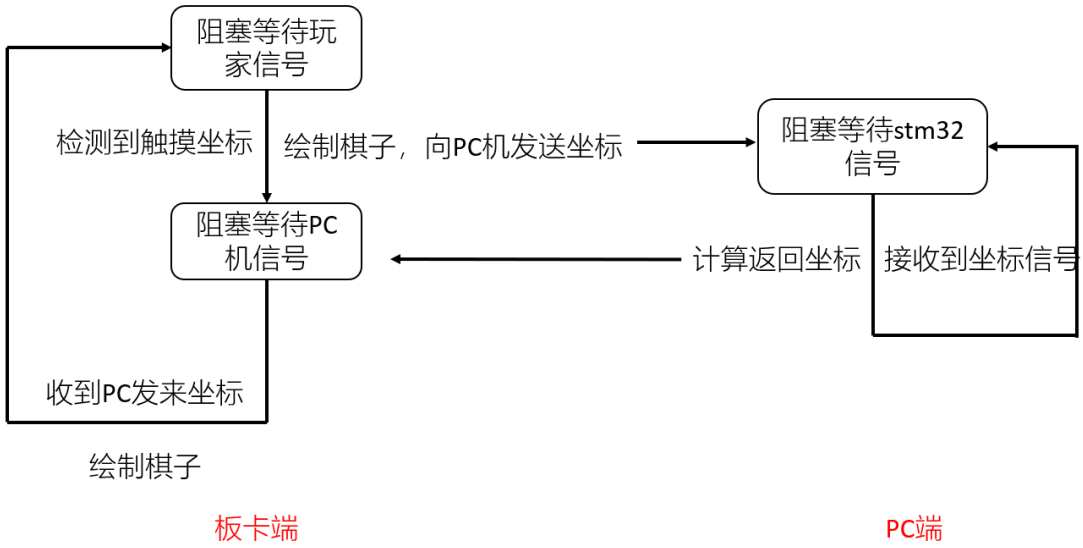
基于STM32的五子棋设计

总体设计

这个项目大致可以分为软件部分和硬件部分。软件部分主要是五子棋AI算法的实现，硬件部分主要是STM32和上位机的串口通信以及其自带触摸屏的使用。

程序状态图如下：



硬件部分(STM32)

串口通信

HDU-EID-V1 开发板提供两个 3 线串口，采用MAX3232 实现串口电平的转换，接口方式为DB9 公头。

```
ReceiveData = USART1.read(2)  # read at most 2 bytes
ReceiveString = ReceiveData.decode()
Coordinate = String2Array(ReceiveString)
```

PC端接收串口信息

```
step = move_to_location(move)
SendData = int(step[0]).to_bytes(1, 'little') + int(step[1]).to_bytes(1, 'little') + b'\r\n'
USART1.write(SendData)
```

PC端发送串口信息

触摸屏校准

HDU-EID-V1开发板含一个LCD液晶屏和一个电阻触摸屏接口，LCD分辨率为320 *240，每个像素点为16 位色。

```c
void usart_send(USART_TypeDef* USARTx,u8 *data, u8 len)
{

  u8 i;
  for(i = 0;i<len;i++)
  {
    while(USART_GetFlagStatus(USARTx,USART_FLAG_TC)==RESET);
    USART_SendData(USARTx,*data++);

  }
}
```

板卡端发送函数

```c
void USART1_IRQHandler(void)                //串口1中断服务程序
{
  u8 Res;
  unsigned char clearidle = clearidle;
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)  //接收中断(接收到的数据必须是0x0d 0x0a结尾)
    {
      Res =USART_ReceiveData(USART1); //读取接收到的数据（每次只接受一个字节的数据吗？）

      if((USART_RX_STA&0x8000)==0)//接收未完成
        {
        if(USART_RX_STA&0x4000)//之前已经接收到了0x0d
          {
          if(Res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
          else USART_RX_STA|=0x8000;  //接收完成了
          }
        else //之前还没收到0X0D
          {
          if(Res==0x0d)USART_RX_STA|=0x4000;//这次接收到的是0x0d
          else
            {
            USART_RX_BUF[USART_RX_STA&0X3FFF]=Res ;
            USART_RX_STA++;
            if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;//接收数据错误,重新开始接收
            }
          }
        }
    }
  else if(USART_GetITStatus(USART1, USART_IT_IDLE)){
    //read register to clear SR
    clearidle = USART1->SR;
    clearidle = USART1->DR;
  }
}
```

板卡端串口接收中断函数，需要寄存器的状态

软件部分（上位机）

论文研读

我仔细阅读了AlphaZero的相关论文：**Mastering the Game of Go without Human Knowledge**，并且参考了相关算法在五子棋上的实现[2]。设计了五子棋AI的软件部分。

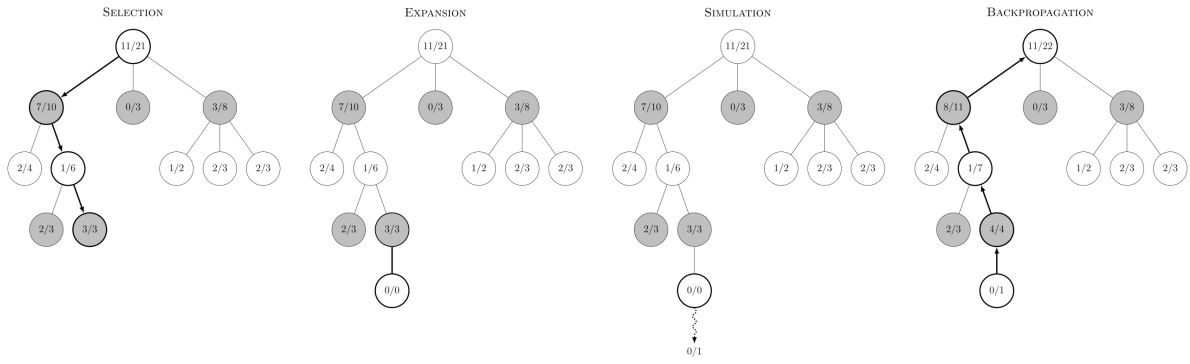
Alpha Go使用的方法：

Alpha Go/Zero system is a mix of several methods assembled into one great engineering piece of work. The core components of the Alpha Go/Zero are:

- **Monte Carlo Tree Search** (certain variant with PUCT function for tree traversal)
- Residual Convolutional **Neural Networks** – policy and value network(s) used for game evaluation and move prior probability estimation
- **Reinforcement learning** used for training the network(s) via self-plays

蒙特卡洛树搜索（MCTS）

蒙特卡洛树搜索（**Monte Carlo Tree Search**，简称 MCTS）是 Rémi Coulom 在 2006 年在它的围棋人机对战引擎「Crazy Stone」中首次发明并使用的的，并且取得了很好的效果。



伪代码：

```python
def monte_carlo_tree_search(root):
    while resources_left(time, computational power):
        leaf = traverse(root) # leaf = unvisited node
        simulation_result = rollout(leaf)
        backpropagate(leaf, simulation_result)
    return best_child(root)
```

```
7
8  def traverse(node):
9      while fully_expanded(node):
10         node = best_uct(node)
11     return pick_univisted(node.children) or node # in case no children are
   present / node is terminal
12
13 def rollout(node):
14     while non_terminal(node):
15         node = rollout_policy(node)
16     return result(node)
17
18 def rollout_policy(node):
19     return pick_random(node.children)
20
21 def backpropagate(node, result):
22    if is_root(node) return
23    node.stats = update_stats(node, result)
24    backpropagate(node.parent)
25
26 def best_child(node):
27     pick child with highest number of visits
```

代码设计

这部分主要是从代码库[5]和[2]学习的，摘录如下。

`mcts_pure.py`

纯蒙特卡洛树法，用来评估当前AI的能力。

`mcts_alphaZero.py`

AlphaGo中使用的蒙特卡洛树

代码解读:

`TreeNode`

蒙特卡洛树的节点类

`MCTS`

蒙特卡洛树搜索类

`MCTSPlayer`

蒙特卡洛树玩家

自我对局（self-play）

self-play数据的扩充

围棋具有旋转和镜像翻转等价的性质，其实五子棋也具有同样的性质。在AlphaGo Zero中，这一性质被充分的利用来扩充self-play数据，以及在MCTS评估叶子节点的时候提高局面评估的可靠性。但是在AlphaZero中，因为要同时考虑国际象棋和将棋这两种不满足旋转等价性质的棋类，所以对于围棋也没有利用这个性质。而在我们的实现中，因为生成self-play数据本身就是计算的瓶颈，为了能够在算力非常弱的情况下尽快的收集数据训练模型，每一局self-play结束后，我们会把这一局的数据进行旋转和镜像翻转，将8种等价情况的数据全部存入self-play的data buffer中。这种旋转和翻转的数据扩充在一定程度上也能提高self-play数据的多样性和均衡性。

训练网络

摘录自[5]。

策略价值网络训练

所谓的策略价值网络，就是在给定当前局面 s 的情况下，返回当前局面下每一个可行action的概率以及当前局面评分的模型。

1. 局面描述方式

使用了4个8 × 8的二值特征平面，其中前两个平面分别表示当前player的棋子位置和对手player的棋子位置，有棋子的位置是1，没棋子的位置是0. 然后第三个平面表示对手player最近一步的落子位置，也就是整个平面只有一个位置是1，其余全部是0. 第四个平面，也就是最后一个平面表示的是当前player是不是先手player，如果是先手player则整个平面全部为1，否则全部为0。

2. 网络结构

最开始是公共的3层全卷积网络，分别使用32、64和128个3 * 3的filter，使用ReLu激活函数。然后再分成policy和value两个输出，在policy这一端，先使用4个 1 * 1 (或者4个128 × 1 × 1)的filter进行降维，再接一个全连接层，使用softmax非线性函数直接输出棋盘上每个位置的落子概率；在value这一端，先使用2个 1 * 1的filter进行降维，再接一个64个神经元的全连接层，最后再接一个全连接层，使用tanh非线性函数直接输出 [-1, 1] 之间的局面评分。

整个策略价值网络的深度只有5~6层，训练和预测都相对比较快。

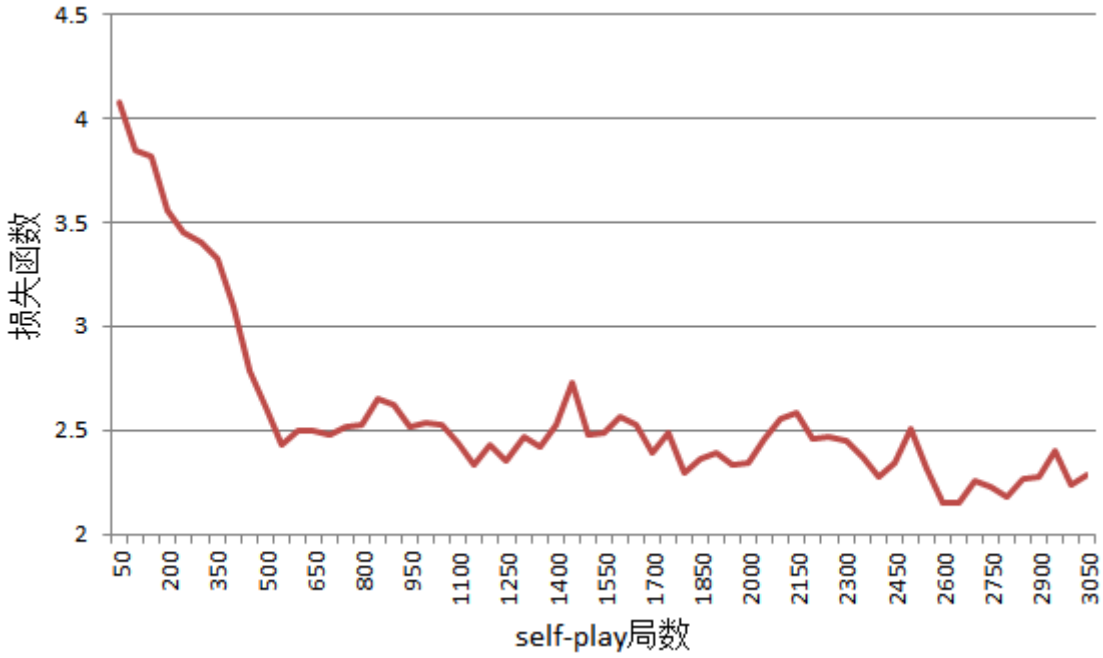
网络结构的代码如下：

```
class Net(nn.Module):
    """policy-value network module"""
    def __init__(self, board_width, board_height):
        super(Net, self).__init__()

        self.board_width = board_width
        self.board_height = board_height
        # common layers
        self.conv1 = nn.Conv2d(4, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        # action policy layers
        self.act_conv1 = nn.Conv2d(128, 4, kernel_size=1)
        self.act_fc1 = nn.Linear(4*board_width*board_height,
                                 board_width*board_height)
        # state value layers
        self.val_conv1 = nn.Conv2d(128, 2, kernel_size=1)
        self.val_fc1 = nn.Linear(2*board_width*board_height, 64)
        self.val_fc2 = nn.Linear(64, 1)

    def forward(self, state_input):
        # common layers
```

```
        x = F.relu(self.conv1(state_input))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        # action policy layers
        x_act = F.relu(self.act_conv1(x))
        x_act = x_act.view(-1, 4*self.board_width*self.board_height)
        x_act = F.log_softmax(self.act_fc1(x_act))
        # state value layers
        x_val = F.relu(self.val_conv1(x))
        x_val = x_val.view(-1, 2*self.board_width*self.board_height)
        x_val = F.relu(self.val_fc1(x_val))
        x_val = F.tanh(self.val_fc2(x_val))
        return x_act, x_val
```

3. 训练目标

策略价值网络的输入是当前的局面描述 $\boldsymbol{s}$，输出是当前局面下每一个可行action的概率 $\mathbf{p}$ 以及当前局面的评分 $\boldsymbol{v}$，而用来训练策略价值网络的是我们在self-play过程中收集的一系列的 $(\boldsymbol{s}, \boldsymbol{\pi}, \boldsymbol{z})$ 数据。根据上面的策略价值网络训练示意图，我们训练的目标是让策略价值网络输出的action概率 $\mathbf{p}$ 更加接近MCTS输出的概率 $\boldsymbol{\pi}$,让策略价值网络输出的局面评分 $\boldsymbol{v}$ 能更准确的预测真实的对局结果 $\boldsymbol{z}$。从优化的角度来说，我们是在self-play数据集上不断的最小化损失函数：$\ell = (z - v)^2 - \boldsymbol{\pi}^T \log \mathbf{p} + c||\theta||^2$,其中第三项是用于防止过拟合的正则项。既然是在最小化损失函数，那么在训练的过程中，如果正常的话我们就会观察到损失函数在慢慢减小。下图展示的是一次在 8×8 棋盘上进行五子棋训练的过程中损失函数随着self-play局数变化的情况，这次实验一共进行了3050局对局，损失函数从最开始的4点几慢慢减小到了2.2左右。



在训练过程中，除了观察到损失函数在慢慢减小，我们一般还会关注策略价值网络输出的策略（输出的落子概率分布）的entropy的变化情况。正常来讲，最开始的时候，我们的策略网络基本上是均匀的随机输出落子的概率，所以entropy会比较大。随着训练过程的慢慢推进，策略网络会慢慢学会在不同的局面下哪些位置应该有更大的落子概率，也就是说落子概率的分布不再均匀，会有比较强的偏向，这样entropy就会变小。也正是由于策略网络输出概率的偏向，才能帮助MCTS在搜索过程中能够在更有潜力的位置进行更多的模拟，从而在比较少的模拟次数下达到比较好的性能。下图展示的是同一次训练过程中观察到的策略网络输出策略的entropy的变化情况。

最后每隔50次self-play对局就对当前的AI模型进行一次评估，评估的方式是使用当前最新的AI模型和纯的MCTS AI（基于随机rollout）对战10局。pure MCTS AI最开始每一步使用1000次模拟，当被我们训练的AI模型10:0打败时，pure MCTS AI就升级到每一步使用2000次模拟，以此类推，不断增强，而我们训练的AlphaZero AI模型每一步始终只使用400次模拟。

学到了什么

这个作业是软硬件相互结合的一个作业，在做这个作业的过程中，我体会到了软件和硬件相互联调的重要性和趣味性。

从软件方面，我学到了人工智能算法。当年alphago非常火热，我当时还在上初中，对他的原理非常好奇，现在我能亲身体会理解算法的实现流程，感觉非常骄傲自豪。

从硬件方面，我学到了触摸屏校准和串口通信的知识。

从思想政治方面，我体会到了科学技术是第一生产力的重要性，决心励志努力学习科学技术，励志报效国家。

参考资料

[1] Monte Carlo tree search - Wikipedia

[2] **AlphaZero_Gomoku**

[3] *Mastering the Game of Go without Human Knowledge*

[4] Monte Carlo Tree Search – beginners guide

[5] AlphaZero实战：从零学下五子棋（附代码）