五子棋 AI：基于 Minimax 搜索和 AlphaZero 的实现

Xinyu Che

2025 年 6 月 26 日

目录

1 概述

本项目是上海交通大学 John 班 2025 年夏季学期 CS1207 程序设计与数据结构 III 的课程大作业。在这里，你将分别基于 Minimax 搜索和 AlphaZero，在我们提供的代码框架下分别实现两个不同的五子棋 AI。本项目的大致内容是：

- 你将会学到基于树搜索的棋类 AI 设计基本思想与相关算法。
- 你将会基本了解基于强化学习和深度神经网络的现代 AI 设计方法。
- 对于 John 班的学生，你可以在这里熟悉 Python，并学习如何使用 Pytorch 这一现代深度学习框架。

1.1 五子棋的规则

Gomoku，即五子棋。基础的五子棋规则是两名玩家分为黑白两方，黑方先手，白方后手，轮流在一个 15×15 的棋盘上落子，先将自己的棋子在横、纵或斜方向上连成五子者获胜。

正规赛事上的五子棋规则要复杂得多，包括 2013 年版《中国五子棋竞赛规则》，索索夫-8 规则以及塔拉山口开局规则等。这些规则的主要目的应当是制衡先手的巨大优势，尽量维持游戏的平衡性。方便起见，这里我们无意深入这些规则，而只会针对基本的五子棋规则设计 AI。

更确切的说，本项目将要分别实现的两个 AI 所遵循的规则有细微的区别。你将要实现的 Minimax AI 在基本规则以外还需支持“第三手可交换”。具体的，在白方第二次落子时，其可以选择交换执棋权，即自己成为黑方，对方成为白方，再由对方继续落子。你也可以理解为白方第二次落子时可以选择翻转棋盘上棋子的颜色，然后继续由黑方落子。而 AlphaZero AI 则不支持此规则，只需支持最基本的五子棋规则。

1.2 棋盘 UI

本项目先前的版本并没有提供可玩的棋盘 UI，而是要求使用 html+css+Javascript 自行实现一个用于 AI 交换的前端。鉴于前端技术在科研中用处相对不大，我们在本学期的项目中删去了这一部分，而直接提供基于 Pygame 的棋盘 UI，如图 1所示。

特别的，该 UI 最初是为 AlphaZero AI 准备的，因此其不支持“第三手换手”规则。我们为 Minimax Search AI 编写了包装器以将其接入该 UI，在使用该 UI 进行人机交互时请禁用你的 Minimax Search AI 的换手操作。

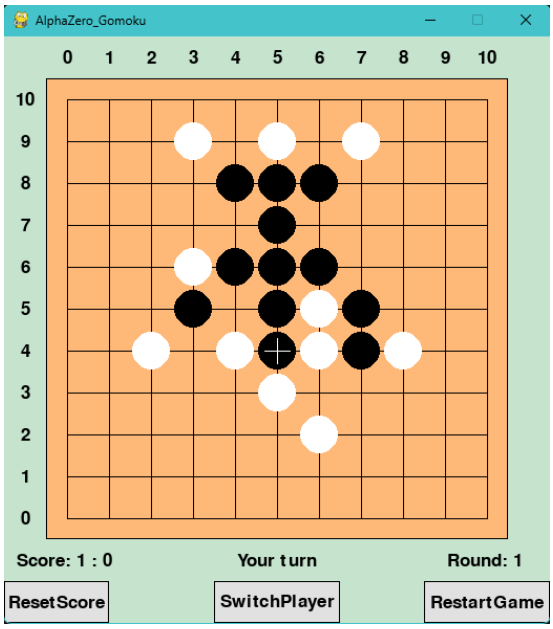

图 1: 棋盘 UI

2 如何上手

本项目对硬件资源有一定的要求。具体的，Minimax Search AI 是不需要 GPU 资源的，但是对 CPU 的要求较高。本项目最后会对 Minimax Search AI 做性能评估，方法是和 baseline 进行多轮对战，并计算胜率。去年本人使用 Intel i7-9750H 运行 64 轮对战的性能测试，一次花费了 2 个多小时。而如果你有更好的 CPU，例如 i9-14900HX 级别，运行时间会缩短非常多。

AlphaZero AI 的计算开销主要是 GPU 上的，因此如果你本地有一块本世代的主流 GPU 的话（RTX4060 级别），做这个项目会方便一些。使用本项目的方法，在 2018 年训练一个 11×11 棋盘上的五子棋 AI 需要在 2 块 CPU 和两块 1080ti GPU 上花费大概 800 小时，因此在本地短时间内训练 15×15 的标准棋盘上的五子棋 AI 其实并不现实。所以本项目将会直接提供训练参数[1]，并由你将其导入自行搭建的模型当中。你可以在本地与训练完成的 AI 通过 UI 进行交互，就本人的尝试[2]而言，如果仅使用 CPU 进行推理，单步耗时大概需要若干秒；如果使用 GPU 加速，推理时间将会显著的缩短。另外，为了测试你实现的训练步骤的正确性，你将会在本地训练一个井字棋 AI，如果你有本地的 GPU 资源，训练过程也可以被显著的加速。但总的来说，即使没有 GPU，本部分也是完全可做的。另外，你可以在 Google Colab[3] 上找到免费的 NVIDIA TESLA T4 GPU 资源。

[1] 11×11 尺寸和 15×15 尺寸的，其中后者还没有完成

[2] 本地配置 AMD R7-8845H CPU 与 NVIDIA RTX 4070 laptop GPU

[3] https://colab.research.google.com/

2.1 项目布局

本项目的布局如下所示：

```
.
├── AlphaZero # AlphaZero AI 的核心部分，需要你自行实现
│   ├── __init__.py
│   ├── mcts_alphaZero.py # AlphaZero 的 MCTS 部分
│   ├── mcts_pure.py # 用于训练过程准确率评估，支持单独运行的纯 MCTS AI
│   └── policy_value_net_torch.py # AlphaZero 的价值评估网络
├── Board # 棋盘 UI 的源代码，已经提供
│   ├── GUI_v1_4.py
│   ├── __init__.py
│   └── game_board.py
├── Deep Reinforcement Learning.pdf
├── LICENSE
├── Minimax # Minimax Search AI 的部分
│   ├── AIController.h # Minimax Search AI 的控制器
│   ├── __init__.py
│   ├── baseline.cpp # 测试用 baseline 的源代码
│   ├── evaluate.py # 用于最终的胜率测试
│   ├── judge.py # 用于 Minimax Search AI 的单独测试
│   ├── mcts_baselines # linux 环境下纯 MCTS AI 的可执行文件
│   │   └── ...
│   ├── sample.cpp # 一个用于展示接口的随机 AI，你应参照此部分实现自己的 AI
│   └── search_wrapper.py # 用于将 Minimax Search AI 接入 UI 的包装器
├── README.md
├── human_play.py
├── model # AlphaZero 的 Pytorch 模型参数
│   ├── 11_11_5.pt
│   ├── 3_3_3.model
│   └── best_policy.model # 模型训练过程自动保存的最优策略
├── tensorflow
│   ├── model_11_11_5
│   │   └── ...
│   ├── model_15_15_5
│   │   └── ...
│   ├── policy_value_net_tensorlayer.py
│   └── toolkit
│       ├── compare_tf_torch.py
│       ├── convert_tf2torch.py
│       ├── param_filter.py
│       ├── param_tf2torch.py
│       ├── tensorflow_params.txt
│       └── torch_params.txt
```

```
└── train.py
```

特别的，由于本项目中 AlphaZero AI 的策略网络部分最初是使用老版本的 tensorflow 实现的，我重写了 Pytorch 的版本，并且将 tensorflow 的网络参数转换成了 Pytorch 参数。tensorflow 文件夹中存放了 tensorflow 网络的实现代码、参数以及我的参数转换工具。

2.2 配置

你应当从 [repo] 中下载本项目的仓库。鉴于 Minimax Search 的 mcts baseline 是 Linux 环境下的可执行文件，我们推荐在 Linux 下进行本项目的开发。为了避免兼容性问题，建议使用 conda 新建本项目的开发环境。在安装 conda 以后，进入本项目的根目录，使用命令：

```
conda create --name gomoku python=3.10
conda activate gomoku
```

以创建并进入本项目的环境。接下来，使用命令：

```
pip install -r requirements.txt
```

安装本项目所需的包。特别的，requirements.txt 中默认安装带有最新版 cuda 12.1 的 Pytorch 版本，如果你没有支持 cuda 的 GPU，请安装仅 CPU 版本的 Pytorch。

2.3 接口

对于 Minimax Search AI，请参照 sample.cpp 实现你的 AI。其中 std::string ai_name 是你的 AI 的名字；extern int ai_side 是你的执棋方；void init() 会在开局时被调用；std::pair<int, int> action() 接受上一步对手的行动，并返回当前自己的行动，若输入为 (-1, -1) ，则说明你应落第一颗黑子，或者对方要求换手。

要进行 Minimax Search AI 的单独测试，请先编译你的 AI。接下来执行：

```
python judge.py ai0path/human ai1path/human
```

其中 aipath 为 AI 可执行文件的路径，human 代表人类玩家，前者为先手，后者为后手。例如：

```
python judge.py ./sample human
```

若要进行胜率评测，执行命令：

```
python evaluate.py --agents-path my_agent_path opponent_agent_path
--num-plays 64 --num-workers 8
```

其中 my_agent_path 与 opponent_agent_path 为对局双方的路径，–num-plays 为测试轮数，–num-workers 为测试利用的 CPU 核数，视你本地配置而定。

对于 AlphaZero AI，你需要首先在 mcts_pure.py 中实现一个基于纯蒙特卡洛树搜索（MCTS）方法的 AI，我们已经在该文件中提供了代码框架。由于 MCTS 方法本身并不算优秀，而且 python 语言的计算效率很低，该 AI 的性能将会显著的劣于最后的 AlphaZero AI，你自己的 Minimax Search AI，以及 C++ 实现的 MCTS AI。该 AI 的主要用途是在 AlphaZero AI 的训练过程中，作为基准评估 AlphaZero AI 的性能。另外你也可以通过 UI 与该 AI 交互。

接下来，你需要在 policy_value_net_torch.py 中使用 Pytorch 实现 AlphaZero AI 的策略评估网络。我们已经在该文件中提供了代码框架，你应参照我们提供的接口，填写策略网络的关键部分。最后，你需要在 mcts_alphaZero.py 中实现基于策略评估网络和 MCTS 的 AlphaZero AI。我们同样已经在该文件中提供了代码框架，你会发现这部分流程非常类似于纯 MCTS AI，主要的不同点在于我们转而使用深度神经网络执行状态评估，而不再在叶子节点执行 roll out 操作；同时 AlphaZero 需要额外支持自我对弈。

若要进行人机交互，请查看 human_play.py。你需要修改其中的胜利判定规则（即几子连成一线判胜）和棋盘大小以适配你希望交互的 AI。如果你的 AI 需要导入预训练参数，你应该修改其中的参数路径 model_file。接下来，根据具体的 AI 类别选择 alpha_zero_player/minimax_player/mcts_player。最后，执行命令：

```
python human_play.py
```

若要进行 AlphaZero AI 的训练，请查看 train.py。你应在 class TrainPipeline() 的初始化处设置训练超参数。配置完成后执行命令：

```
python train.py
```

训练过程中的最优模型 best_policy.model 保存在 model 文件夹中，当前模型 current_policy.model 则保持在 tmp 文件夹中。特别的，由于 Pytorch 版本和原先 tensorflow 版本的区别，你需要在 policy_value_net_torch.py 中修改 learning rate。另外，本项目应该还有很多可供调整的参数，请由你自行探索。

3 Minimax Search AI

Minimax Search，顾名思义，其实是一种用于博弈游戏的带有某种剪枝的搜索算法，示意图如图 2 。总的来说，这种算法在理论上比较简单，并不需要大量的数学分

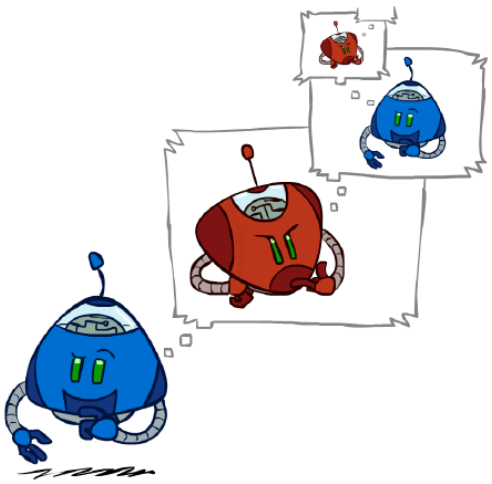

图 2: Minimax Search

析，但是有一些技术上的优化。在本项目中，您将从头开始自己使用 C++ 实现一个基于 Minimax Search 的五子棋 AI，并为其添加优化，主要包括：

- 局势评估与基于单点估价的启发式搜索
- Zobrist 缓存（某种哈希记忆化）
- 算杀

使用以上的技术，您应该可以在本地实现一个棋力尚可的 AI。当然，以上的优化并不是必做的，您的终极目标应是提升 AI 的棋力。

鉴于这部分并没有什么值得深入的地方，而且我没什么时间写 handout 了，细节处请大家直接参考 [repo] 。

4 Pytorch 与深度学习

鉴于 John 班的同学此前可能没有学习过 Pytorch，我们将在这里简要的介绍 Pytorch，主要是提供一些习题。

4.1 Pytorch

PyTorch 是一个由 Facebook 的 AI 研究团队开发的开源机器学习库，主要用于构建和训练深度学习模型。它的核心是两个主要功能：

- **张量计算（Tensor Computation）**：类似于 NumPy，但提供了强大的 GPU 加速功能。张量（Tensor）使用上只是换了个名字的多维数组，是神经网络中数据和参数的基本单位。

- **自动微分（Automatic Differentiation）：**PyTorch 内置了一个名为 'autograd' 的自动微分引擎，其核心是微分的链式法则，可以自动计算任意计算图的梯度。这对于神经网络的训练至关重要，因为它极大地简化了反向传播算法的实现。

由于其灵活性、易用性和 Pythonic 的接口设计，PyTorch 已经成为学术界和工业界最受欢迎的深度学习框架之一。你可以在本项目中看到另一深度学习框架 tensorflow 实现的老版策略网络，其结构相比 Pytorch 会复杂很多。

我们并不会深究 Pytorch 的实现机理，而只是希望学会使用它。为此，从一个工具的角度出发，我们建议你直接在实践中熟悉 Pytorch。你可以在 UMich EECS 498-007 / 598-005: Deep Learning for Computer Vision 的 Assignment 1 PyTorch 101 和 Assignment 4 PyTorch Autograd 中分别学习 Pytorch 的基本使用及其高级抽象和自动微分引擎。

4.2 深度学习

深度学习是机器学习的一个分支，它使用多层（即“深度”）的人工神经网络来学习数据中的复杂模式。具体的，深度神经网络通过梯度方法来拟合数据。在训练过程中神经网络接受样本数据，做出预测，然后通过一个（经验上的）损失函数来衡量预测的准确性。最后，利用梯度方法调整参数以最小化损失，这一过程往往是采用反向传播实现的。

深度学习现阶段还没有非常严格的理论框架，所以正经的深度学习课的重点一般也是网络的设计和以往的技术，重点在于直观。尽管我应该会介绍本项目涉及到的深度学习基础，本文并不会深入深度学习。推荐的参考资料是 UMich EECS 498-007 / 598-005: Deep Learning for Computer Vision ，特别是 Lec 5,6,7,8 。另外你可以在 这里 找到本人撰写的课程笔记。

5 AlphaZero AI

相比 Minimax Search，AlphaZero 是一个现代得多的方法。大家都知道 2017 年 DeepMind 的 AlphaGo 击败了围棋世界冠军柯洁，而 AlphaZero 则是 AlphaGo 的后续迭代版本。具体的，现在的 AlphaZero 并不需要学习高段人类棋手的对局，其基本只需蒙特卡洛树搜索与深度强化学习（再加上大量的预训练），与围棋的基本规则，就可以自行找到相当不平凡的策略。这里我们将简要介绍 AlphaZero 的基本机理，以便你实现自己的 AlphaZero for Gomoku。

特别的，以下的讲义并不会对所涉及的理论提供非常严格的数学分析，而只为流程理解和代码实现而服务[4]。该部分参考了 UMich EECS 498-007 / 598-005: Deep Learn-

[4]我甚至没法保证我讲的是完全正确的。没正经学过是这样的。

ing for Computer Vision 与 Deep Reinforcement Learning: Fundamentals, Research and Applications，部分摘自 Blog 。

5.1 什么是强化学习

鉴于读者可能没有系统的学习过机器学习，我们将简要的介绍强化学习这一概念。总的来说，强化学习是一个相当难的问题，其目标是训练一个依据某种规则与外界交互的模型。一个合理的建模是，设想存在某个外部环境，以及某个我们希望训练的模型。该模型在时刻 t 从环境中获取当前环境的状态 s_t ，然后做出自己的行动 a_t ，该行动对环境造成影响，同时环境要向模型发送奖励 r_t 。接下来环境和模型都会变化，并进行下一轮交互，以此类推。其难处主要在于：

- 随机性。环境可能会随机改变。
- 反馈模糊。我们可能不知道是哪一步行动导致了当前的奖励，即奖励会受先前的行动影响。
- 不可微。我们往往希望通过梯度手段来调整模型，而强化学习的过程不一定是可微的。
- 环境不稳定。随着模型的行动，外界环境可能会发生较大的改变，导致模型先前收集到的信息、学习到的方式难以适用于新环境。更严格的说，外界环境对应的分布不是一个稳定的分布。

5.2 马尔可夫决策过程与 Deep-Q Learning

现在我们将简要介绍强化学习的数学建模之一，马尔可夫决策过程。[5]这里应当并不特别需要什么前置的数学知识。我们使用一个 5 元组来描述马尔可夫决策过程的状态空间，即 (S, A, R, P, γ) ，其中：

- S ：环境的所有可能状态的集合。
- A ：模型的所有可能行动的集合。
- R ：给定 $(s, a) \in S \times A$ 时奖励的分布。
- P ：给定 $(s, a) \in S \times A$ 时，下一步转移到的状态的分布。
- γ ：衰减系数。

[5]本节的讲法来自 UMich EECS 498-007 / 598-005 Lec 21

特别的，我们要求当前状态 $s \in S$ 完全描述了外部环境，即给定 $(s, a) \in S \times A$ 时分布 R 和 P 就完全确定了。我们将该性质称作马尔可夫性。我们的目标是让模型学习某一策略 π ，使其最大化长期的收益：

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

其中 r_t 代表第 t 步的奖励，γ 即为衰减系数。因此，衰减系数的作用主要有两个，其一是使得模型在一定程度上更为关注较近的奖励，其二也潜在的帮助长期收益收敛。而模型在马尔可夫决策过程中的行动模式则是：

- 当 $t = 0$ ，环境从给定的初始的状态分布，记作 P_0 ，中取样，得到初始状态。
- 接下来，模型不断重复：
  - 模型根据策略 π，从分布 $\pi(a \mid s_t)$ 中取样得到当前行动 a_t 。
  - 环境从分布 $R(r \mid s_t, a_t)$ 中取样得到当前奖励 r_t 。
  - 环境从分布 $P(s \mid s_t, a_t)$ 中取样得到下一状态 s_{t+1} 。
  - 模型接受奖励 r_t 并转移到下一状态 s_{t+1} 。

当然，这是一个随机过程，所以我们无法期待稳定的收益。因此，我们的优化目标应是给定策略 π 时的最大期望收益，也就是：

$$\pi^* = \arg\max_{\pi} \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi\right]$$

在给定策略 π 时，我们希望衡量某一个状态的优劣，为此我们定义价值函数：

$$V^{\pi}(s) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi\right]$$

有时我们希望衡量在当前状态 s 下采取行动 a 时的收益，为此进一步定义 Q 函数：

$$Q^{\pi}(s, a) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

现在我们考虑最优的 Q 函数，即：

$$Q^*(s, a) = \arg\max_{\pi} \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

现在我们就要做一些假设了。一般来说，我们认为 $\pi(a \mid s)$ 应是一个分布，然而此处我们假设每一步仅做确定性的选择，即令 $\pi(a \mid s)$ 为一 $S \to A$ 的函数。这样，我们可以期待：

$$\pi^*(s) = \arg\max_{a'} Q^*(s, a')$$

现在考虑如何求解 Q^*，我们期待 Q^* 满足递推关系，而这一点其实应该是有保障的，即如下的 Bellman Equation：

$$Q^*(s,a) = \mathbf{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

其中 $r \sim R(s,a), s' \sim P(s,a)$。实际上我们期待 Bellman Equation 满足一些更强的性质。我们不加证明的给出：

- 对于任一满足 Bellman Equation 的 $Q(s,a)$，其必然恰为最优的 Q^*。
- 考察函数列：$Q_0, Q_1, \cdots$，其中 Q_0 为任意 $S \times A \Rightarrow \mathbb{R}$，满足：

  $$\forall i \geq 1, s \in S, a \in A : Q_{i+1}(s,a) = \mathbf{E}\left[r + \gamma \max_{a'} Q_i(s',a')\right]$$

  有 $Q_0, Q_1, \cdots$ 一致收敛到 Q^*。

要求解 Q^*，一个简单的思路是我们直接按照如上性质，从任意初始函数开始做迭代。然而 Q^* 的定义域可能很大，使得这一方法无法实现。此处我们考虑引入神经网络来预测 Q^*，并使用 Bellman Equation 作为损失函数。即，假设我们使用参数为 θ 的网络预测 Q^*，得到 $Q(s,a;\theta)$。考虑：

$$y_{s,a;\theta} = \mathbf{E}\left[r + \gamma \max_{a'} Q(s',a';\theta)\right]$$

我们期待 $y_{s,a;\theta} = Q(s,a;\theta)$，例如设定损失函数为：

$$L(s,a) = (y_{s,a;\theta} - Q(s,a;\theta))^2$$

其中的期望则通过对 r, s' 取样来近似，这就是深度强化学习中的 Deep-Q Learning 方法。通过该方法，我们可以基于 Q^* 来求解 $\pi^*(s)$ 从而进行策略规划。

5.3 蒙特卡洛树搜索

在 Deep-Q Learning 的基础上，还有两个实践上的问题：

- 我们如何对 r, s' 取样，从而近似 $y_{s,a;\theta}$？
- 我们如何得到训练策略网络所需的大量数据？

在棋类游戏中，这些问题可以通过结合 Deep-Q Learning 和蒙特卡洛树搜索（MCTS）来解决。具体的，我们先要介绍基本的 MCTS 方法。本质上，MCTS 和 Minimax Search 一样，都是五子棋的策略树上的搜索算法，不同之处在于 Minimax Search 的出发点是遍历，而 MCTS 则是随机取样。考虑如何评估当前状态的优劣，一个简单的思路是，我们直接从这个状态开始完全均匀随机的落子，直到棋局结束为止。我们大量重复随机对

局，最后取胜利的局数与失败的局数的差的平均值作为该状态优劣的评估指标。一个主要的问题是我们必然没有足够的算力对所有的后继状态做充分的随机模拟，所以我们期望找到一个更聪明的方法，这就是 MCTS。

具体的，给定当前局面，我们期望判断下一步该落哪一子。MCTS 的搜索树最初只有根节点，代表当前局面，然后 MCTS 会在大量的模拟中逐步构建其搜索树。本项目中 MCTS AI 的节点类为 class TreeNode，和 AlphaZero 相同，都维护了以下几个值：

- parent：父节点。
- children：一个从 action 到 TreeNode 的映射，对应子节点。
- n：MCTS 过程中该节点被访问的次数。
- Q：该节点的 Q 值。和 Deep-Q Learning 中不同，其为该节点的当前玩家在子树内受到贡献的均值。
- u：该节点的最终评估优劣度。
- P：该节点的先验概率。

MCTS 的具体步骤是：

- 从根节点出发，每一次根据当前节点所有儿子的 u 值，选择最优者，并向其移动。其中每个节点的 u 值计算按照：

  $$\frac{c \cdot P \cdot \sqrt{N}}{1+n} + Q$$

  其中 N 为该节点的父亲的 n 值，c 为一事先设定的常数，在代码中对应 c_puct。该公式为 UCT（Upper Confidence Bound for Trees）的变体，是多臂老虎机问题在树搜索中的推广，这里也不会深入。

- 当移动到叶子节点 leaf 以后，若 leaf 处的棋局尚未结束，则扩展 leaf 的所有儿子，令其先验概率为均匀分布。
- 从 leaf 开始，完全随机的落子，直到棋局结束。
- 根据第三步的模拟结果计算贡献。若 leaf 处的玩家获胜，则贡献为 1；若失败，贡献为-1；若平局，贡献为 0。然后根据此贡献更新 leaf 及其全体祖先的 n 和 Q。

在当前轮中我们大量重复以上步骤，然后选择被访问次数最大的叶子节点作为实际行动，并相应的移动根节点。

总的来说，MCTS 是一个简单且应用相当广泛的算法，然而其单独使用的性能却不够理想。Google 在 2018 年发表的论文[6]中提出了将 MCTS 与 Deep-Q Learning 结合的方法，也就是 AlphaZero。其主要的修改是，不再在叶子节点处进行随机模拟，而直接使用深度神经网络针对当前局面进行估价。本项目中具体过程的描述为：

- 从根节点出发，每一次根据当前节点所有儿子的 u 值，选择最优者，并向其移动。其中每个节点的 u 值计算仍然按照：

  $$\frac{c \cdot P \cdot \sqrt{N}}{1+n} + Q$$

  其中 N 为该节点的父亲的 n 值。

- 当移动到叶子节点 leaf 以后，若 leaf 处的棋局已经结束，则若 leaf 处的玩家获胜，令其贡献为 1；若失败，贡献为-1；若平局，贡献为 0。而若 leaf 处的棋局尚未结束，则使用策略网络预测其全体儿子的先验概率以及贡献，并使用先验概率的预测值初始化全体儿子。

- 使用上一步得到的贡献，更新选择的节点及其全体祖先的 n 和 Q。

以上两种 MCTS 的实现中有一些共同的细节：

- 请注意搜索树上每一层节点的当前玩家是交替的，因此每一次的贡献都是需要根据当前玩家变号的。

- 在每一轮做出实际行动以后，有一些实现中会选择抛弃搜索树，并在下一次重新构建；然而本项目中推荐复用之前的结果，而不选择抛弃搜索树，实现上你只需向对应的儿子移动即可。

- u 值的计算公式的主要目的是平衡 explore-exploit trade-off，即一方面我们期望多探索哪些表现更好的节点，另一方面我们也要兼顾那些探索次数较少的节点，以避免遗漏了好的策略。c 即为调整 explore 和 exploit 权重的参数，此处我们建议取 5。

- Q 计算的是子树中每一次的贡献的平均值，因而值一定是在 [-1,1] 之间的。

另外，可以看出 AlphaZero 的策略网络不仅要预测每个节点的 Q 值，还要预测全体儿子的先验概率[7]，并且我们使用同一神经网络 self.policy_value_function 来执行所有的价值评估，因此该网络具备自行辨认当前玩家的能力。区别于 MCTS AI，AlphaZero 需要额外支持训练模式的自我对弈，在代码中使用参数 is_selfplay 指示。AlphaZero 的额外细节如下所示：

[6]Silver D, Hubert T, Schrittwieser J, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play[J]. Science, 2018, 362(6419): 1140-1144.

[7]实现上和理论上有区别其实比较正常，这毕竟是深度学习（

- 自我对弈模式下，AlphaZero 每一次的移动都需要通过 MCTS 决定；而在交互模式下接受对方的移动后只需对应改变根节点即可。

- 自我对弈模式下，AlphaZero 在扩展儿子时应添加 Dirichlet Noise，以鼓励 AI 探索未尝试过的策略。我们推荐以 0.75+0.25 的比例混合预测先验概率与噪声，噪声参数推荐为 0.3，如下所示：

```
dirichlet_noise = np.random.dirichlet(0.3 * np.ones(length))
```

- 在决定实际行动时，我们会对访问次数的分布根据温度常数 temp 做指数放缩：

```
probs = softmax(1.0 / temp * np.log(np.array(visits) + 1e-10))
```

  然后我们会从此分布中取样得到下一步的真实行动，而不是像 MCTS AI 那样固定走访问次数最多的点。

- 自我对弈模式下，每一盘棋的前 first_n_moves 手应设定较高的温度，推荐 temp =1，以鼓励探索不同的策略。而在以后的决策，以及交互模式的全程下，temp 应设定得较低，推荐 temp=1e-3，以选择好的策略。

5.4 策略估价网络

本项目中 AlphaZero AI 的策略估价网络是一个较为简单的，基于 Resnet 的双头图像识别网络。特别的，为了避免大量耗时的训练任务，我们提前准备了训练参数，你需要将这些参数导入到你自行实现的网络当中，所以请务必保持网络结构一致。

本项目的网络结构保存在 class PolicyValueNetModel 中，class PolicyValueNet 是其包装器。网络结构示意图如图 3 所示，其中主干部分由一个用于转换 channel 数的卷积块和若干 resblock 构成。后面的两个分类头是不同结构的简单前馈神经网络。此示意图中省去了 batch normalization 等次要结构，具体网络结构请参照代码 policy_value_net_torch.py。特别的，为了保证参数导入无误，你应完全参照框架中的注释实现你的网络，你不应添加或删去任何结构，同时要确保每一层的命名和注释完全一致，除非你可以自行训练出棋力不差的 AlphaZero AI。

5.5 训练过程

AlphaZero 的训练包含若干轮，每一轮的训练中会通过自我对弈获取训练数据，并训练策略网络拟合每个节点的全体儿子的访问次数的分布以及 Q 值。分布采取的损失是交叉熵损失，Q 值采取均方误差损失，另外所有的非 bias 参数都需要带有大小为 1e-4 的 L2 正则化，总损失为以上三项的和。

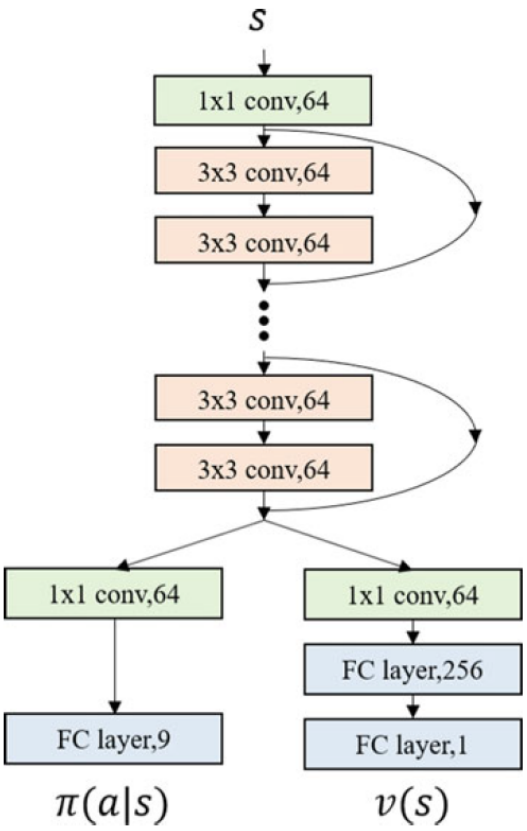

图 3: 策略估价网络

我们的评估方式是在训练一定轮次后让 AlphaZero AI 与 MCTS AI 对弈，若其胜过了之前的版本，就保存最佳模型。若 AlphaZero AI 以百分百的胜率战胜了 MCTS AI，我们会提升 MCTS AI 的模拟次数。

特别的，我们会要求你在 policy_value_net_torch.py 中实现 AI 单次训练的过程，但导入参数并让 AI 在 UI 中跑起来并不会测试这部分代码，所以我们会要求你在本地训练一个井字棋 AI 以保证你正确实现了这部分。我已经调整了 train.py 中的参数，以符合训练井字棋 AI 的要求，但你仍需要调整 policy_value_net_torch.py 中 resblock 的大小以适配 train.py，这是因为井字棋 AI 并不需要非常庞大的策略网络。我们建议你阅读 train.py 的代码，其中注释的参数是原先用于训练 11×11 棋盘上五子棋 AI 的版本。你并不需要等待训练进程结束，因为进程会自动保存训练过程中的最佳参数。我本地的训练仅花费了数分钟的时间。当你训练完成以后，你应当得到了一个不会输的井字棋 AI。

6 要求与评分标准

首先我们要声明，为了确保得分，一般情况下你务必要遵循以下的要求。由于本项目的 Handout 在今年第一次完成，我们的要求可能是具备不合理之处的。所以如果你

认为这些要求有任何的不妥，请尽早告知我，否则一律当作没有完成。考虑到本文档可能随项目一同传承，如果你能挑出 Handout 的错来，或者协助改进本项目，我们也可以设置一些附加得分。

对于 Minimax Search AI，我们要求其单步推理时间限制在 5s 以内。对于你具体实现了哪些部分，我们没有要求。为了保证有效的测试，你的 AI 应具备一定的随机性。方便起见，我们要求先手开局方的第一步必须在棋盘中央 7×7 的范围内随机落子，baseline 已被修改以满足这一要求。你的终极目标是在调用 evaluate.py 进行评估时胜过 baseline，并取得尽量高的胜率。

对于 AlphaZero AI，你应完整实现 AlphaZero 文件夹下的三份代码，我们会基于完成度给分。

特别的，由于算力上的限制，本项目第二部分的自由度实际上并不大。由于你的自由发挥空间基本上集中在 Minimax Search AI 上，我们期望你能独立的开发出以稳定较高的胜率击败 baseline 的 AI。另外，我们发现 Minimax Search AI 在 5s 时限内的表现实际上与你本地的 CPU 资源有极大的关系，所以你需要明确的告知你本地的 CPU 型号。较老的 CPU 可能在推理耗时以及最终胜率上都会产生较大的影响。

最后你需要提交一份报告（建议使用 LaTeX 或者 markdown 编写），包括以下内容：

- 本地的详细运行环境，包括系统环境、CPU 资源、GPU 资源。
- 你实现的 Minimax Search AI 的详细结构与参数，以及你使用的优化。
- 本地运行与 baseline 对战的胜率。
- 你在实现 MCTS AI 与 AlphaZero AI 的过程中遇到的问题，以及你对 AlphaZero 的理解。
- 对本项目的建议。

具体的分数构成是 35% Minimax Search AI ，35% AlphaZero AI 以及 30% Code Review 和 Report 。

最后，请各位把握好时间，尽量自己解决问题。代码上任何形式的抄袭或共享都是不允许的，但我们鼓励交流思想。

7 致谢

[1] Zhicheng Zhang for Minimax Search codebase.

[2] Wendi Chen, Haoyi You, Luda Chen, Jinbo Hu for maintenance.

[3] Hongming Zhang, Tianyang Yu for AlphaZero codebase.