alpha\_zero\_mcts.py

# coding: utf-8

from typing import Tuple, Union

import numpy as np

from .chess\_board import ChessBoard

from .node import Node

from .policy\_value\_net import PolicyValueNet

class AlphaZeroMCTS:

    """ 基於策略-價值網路的蒙特卡洛搜索樹 """

    def \_\_init\_\_(self, policy\_value\_net: PolicyValueNet, c\_puct: float = 4, n\_iters=1200, is\_self\_play=False) -> None:

        """

        Parameters

        ----------

        policy\_value\_net: PolicyValueNet

            策略價值網路

        c\_puct: float

            探索常數

        n\_iters: int

            迭代次數

        is\_self\_play: bool

            是否處於自我博弈狀態

        """

        self.c\_puct = c\_puct

        self.n\_iters = n\_iters

        self.is\_self\_play = is\_self\_play

        self.policy\_value\_net = policy\_value\_net

        self.root = Node(prior\_prob=1, parent=None)

    def get\_action(self, chess\_board: ChessBoard) -> Union[Tuple[int, np.ndarray], int]:

        """ 根據當前局面返回下一步動作

        Parameters

        ----------

        chess\_board: ChessBoard

            棋盤

        Returns

        -------

        action: int

            當前局面下的最佳動作

        pi: `np.ndarray` of shape `(board\_len^2, )`

            執行動作空间中每個動作的概率，只在 `is\_self\_play=True` 模式下返回

        """

        for i in range(self.n\_iters):

            # 複製棋盤

            board = chess\_board.copy()

            # 如果没有遇到葉節點，就一直向下搜索並更新棋盤

            node = self.root

            while not node.is\_leaf\_node():

                action, node = node.select()

                board.do\_action(action)

            # 判斷遊戲是否结束，如果没结束就拓展葉節點

            is\_over, winner = board.is\_game\_over()

            p, value = self.policy\_value\_net.predict(board)

            if not is\_over:

                # 添加狄利克雷噪聲

                if self.is\_self\_play:

                    p = 0.75\*p + 0.25 \* \

                        np.random.dirichlet(0.03\*np.ones(len(p)))

                node.expand(zip(board.available\_actions, p))

            elif winner is not None:

                value = 1 if winner == board.current\_player else -1

            else:

                value = 0

            # 反向傳播

            node.backup(-value)

        # 計算 π，在自我博弈狀態下：遊戲的前三十步，温度係數為 1，後面的温度係數趨於無窮小

        T = 1 if self.is\_self\_play and len(chess\_board.state) <= 30 else 1e-3

        visits = np.array([i.N for i in self.root.children.values()])

        pi\_ = self.\_\_getPi(visits, T)

        # 根據 π 選出動作及其對應節點

        actions = list(self.root.children.keys())

        action = int(np.random.choice(actions, p=pi\_))

        if self.is\_self\_play:

            # 創建维度為 board\_len^2 的 π

            pi = np.zeros(chess\_board.board\_len\*\*2)

            pi[actions] = pi\_

            # 更新根節點

            self.root = self.root.children[action]

            self.root.parent = None

            return action, pi

        else:

            self.reset\_root()

            return action

    def \_\_getPi(self, visits, T) -> np.ndarray:

        """ 根據節點的訪問次数計算 π """

        # pi = visits\*\*(1/T) / np.sum(visits\*\*(1/T)) 會出現標量溢出問題，所以使用對數壓縮

        x = 1/T \* np.log(visits + 1e-11)

        x = np.exp(x - x.max())

        pi = x/x.sum()

        return pi

    def reset\_root(self):

        """ 重置根節點 """

        self.root = Node(prior\_prob=1, c\_puct=self.c\_puct, parent=None)

    def set\_self\_play(self, is\_self\_play: bool):

        """ 設置蒙特卡洛樹的自我博弈狀態 """

        self.is\_self\_play = is\_self\_play

chess\_board.py

# coding: utf-8

from typing import Tuple

from copy import deepcopy

from collections import OrderedDict

import torch

import numpy as np

class ChessBoard:

    """ 棋盤類 """

    EMPTY = -1

    WHITE = 0

    BLACK = 1

    def \_\_init\_\_(self, board\_len=10, n\_feature\_planes=7):

        """

        Parameters

        ----------

        board\_len: int

            棋盤邊長

        n\_feature\_planes: int

            特徵平面的個數，必须為偶數

        """

        self.board\_len = board\_len

        self.current\_player = self.BLACK

        self.n\_feature\_planes = n\_feature\_planes

        self.available\_actions = list(range(self.board\_len\*\*2))

        # 棋盤狀態字典，key 為 action，value 為 current\_player

        self.state = OrderedDict()

        # 上一個落點

        self.previous\_action = None

    def copy(self):

        """ 複製棋盤 """

        return deepcopy(self)

    def clear\_board(self):

        """ 清空棋盤 """

        self.state.clear()

        self.previous\_action = None

        self.current\_player = self.BLACK

        self.available\_actions = list(range(self.board\_len\*\*2))

    def do\_action(self, action: int):

        """ 落子並更新棋盤

        Parameters

        ----------

        action: int

            落子位置，範圍為 `[0, board\_len^2 -1]`

        """

        self.previous\_action = action

        self.available\_actions.remove(action)

        self.state[action] = self.current\_player

        self.current\_player = self.WHITE + self.BLACK - self.current\_player

    def do\_action\_(self, pos: tuple) -> bool:

        """ 落子並更新棋盤，只提供给 app 使用

        Parameters

        ----------

        pos: Tuple[int, int]

            落子在棋盤上的位置，範圍為 `(0, 0) ~ (board\_len-1, board\_len-1)`

        Returns

        -------

        update\_ok: bool

            是否成功落子

        """

        action = pos[0]\*self.board\_len + pos[1]

        if action in self.available\_actions:

            self.do\_action(action)

            return True

        return False

    def is\_game\_over(self) -> Tuple[bool, int]:

        """ 判斷遊戲是否结束

        Returns

        -------

        is\_over: bool

            遊戲是否结束，分出勝負或者平局則為 `True`, 否則為 `False`

        winner: int

            遊戲赢家，有以下幾種:

            \* 如果遊戲分出勝負，則為 `ChessBoard.BLACK` 或 `ChessBoard.WHITE`

            \* 如果還有分出勝負或者平局，則為 `None`

        """

        # 如果下的棋子不到 9 個，就直接判斷遊戲還沒结束

        if len(self.state) < 9:

            return False, None

        n = self.board\_len

        act = self.previous\_action

        player = self.state[act]

        row, col = act//n, act % n

        # 搜索方向

        directions = [[(0, -1),  (0, 1)],   # 水平搜索

                      [(-1, 0),  (1, 0)],   # 垂直搜索

                      [(-1, -1), (1, 1)],   # 主對角線搜索

                      [(1, -1),  (-1, 1)]]  # 副對角線搜索

        for i in range(4):

            count = 1

            for j in range(2):

                flag = True

                row\_t, col\_t = row, col

                while flag:

                    row\_t = row\_t + directions[i][j][0]

                    col\_t = col\_t + directions[i][j][1]

                    if 0 <= row\_t < n and 0 <= col\_t < n and self.state.get(row\_t\*n+col\_t, self.EMPTY) == player:

                        # 遇到相同顏色時 count+1

                        count += 1

                    else:

                        flag = False

            # 分出勝負

            if count >= 5:

                return True, player

        # 平局

        if not self.available\_actions:

            return True, None

        return False, None

    def get\_feature\_planes(self) -> torch.Tensor:

        """ 棋盤狀態特徵張量，维度為 `(n\_feature\_planes, board\_len, board\_len)`

        Returns

        -------

        feature\_planes: Tensor of shape `(n\_feature\_planes, board\_len, board\_len)`

            特徵平面圖像

        """

        n = self.board\_len

        feature\_planes = torch.zeros((self.n\_feature\_planes, n\*\*2))

        # 最後一張圖像代表當前玩家顏色

        # feature\_planes[-1] = self.current\_player

        # 添加歷史信息

        if self.state:

            actions = np.array(list(self.state.keys()))[::-1]

            players = np.array(list(self.state.values()))[::-1]

            Xt = actions[players == self.current\_player]

            Yt = actions[players != self.current\_player]

            for i in range((self.n\_feature\_planes-1)//2):

                if i < len(Xt):

                    feature\_planes[2\*i, Xt[i:]] = 1

                if i < len(Yt):

                    feature\_planes[2\*i+1, Yt[i:]] = 1

        return feature\_planes.view(self.n\_feature\_planes, n, n)

class ColorError(ValueError):

    def \_\_init\_\_(self, \*args: object) -> None:

        super().\_\_init\_\_(\*args)

node.py

# coding: utf-8

from math import sqrt

from typing import Tuple, Iterable, Dict

class Node:

    """ 蒙特卡洛樹節點 """

    def \_\_init\_\_(self, prior\_prob: float, c\_puct: float = 5, parent=None):

        """

        Parameters

        ----------

        prior\_prob: float

            節點的先驗概率 `P(s, a)`

        c\_puct: float

            探索常數

        parent: Node

            父級節點

        """

        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 = {}  # type:Dict[int, Node]

    def select(self) -> tuple:

        """ 返回 `score` 最大的子節點和該節點應對的 action

        Returns

        -------

        action: int

            動作

        child: Node

            子節點

        """

        return max(self.children.items(), key=lambda item: item[1].get\_score())

    def expand(self, action\_probs: Iterable[Tuple[int, float]]):

        """ 拓展節點

        Parameters

        ----------

        action\_probs: Iterable

            每個元素都為 `(action, prior\_prob)` 元组，根據這個元组創建子節點，

            `action\_probs` 的長度為當前棋盤的可用落點的總數

        """

        for action, prior\_prob in action\_probs:

            self.children[action] = Node(prior\_prob, self.c\_puct, self)

    def \_\_update(self, value: float):

        """ 更新節點的訪問次數 `N(s, a)`、節點的累計平均獎賞 `Q(s, a)`

        Parameters

        ----------

        value: float

            用來更新節點内部數據

        """

        self.Q = (self.N \* self.Q + value)/(self.N + 1)

        self.N += 1

    def backup(self, value: float):

        """ 反向傳播 """

        if self.parent:

            self.parent.backup(-value)

        self.\_\_update(value)

    def get\_score(self):

        """ 計算節點得分 """

        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):

        """ 是否為葉節點 """

        return len(self.children) == 0

# coding: utf-8

import torch

from torch import nn

from torch.nn import functional as F

from .chess\_board import ChessBoard

class ConvBlock(nn.Module):

    """ 卷積塊 """

    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)))

class ResidueBlock(nn.Module):

    """ 残差塊 """

    def \_\_init\_\_(self, in\_channels=128, out\_channels=128):

        """

        Parameters

        ----------

        in\_channels: int

            輸入圖像通道數

        out\_channels: int

            輸出圖像通道數

        """

        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)

class PolicyHead(nn.Module):

    """ 策略頭 """

    def \_\_init\_\_(self, in\_channels=128, board\_len=10):

        """

        Parameters

        ----------

        in\_channels: int

            輸入通道數

        board\_len: int

            棋盤大小

        """

        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)

class ValueHead(nn.Module):

    """ 價值頭 """

    def \_\_init\_\_(self, in\_channels=128, board\_len=10):

        """

        Parameters

        ----------

        in\_channels: int

            輸入通道數

policy\_value\_net.py

        board\_len: int

            棋盤大小

        """

        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

class PolicyValueNet(nn.Module):

    """ 策略價值網路 """

    def \_\_init\_\_(self, board\_len=10, n\_feature\_planes=6, is\_use\_gpu=True):

        """

        Parameters

        ----------

        board\_len: int

            棋盤大小

        n\_feature\_planes: int

            輸入圖像通道數，對應特徵

        """

        super().\_\_init\_\_()

        self.board\_len = board\_len

        self.is\_use\_gpu = is\_use\_gpu

        self.n\_feature\_planes = n\_feature\_planes

        self.device = torch.device('cuda:0' if is\_use\_gpu else 'cpu')

        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):

        """ 前饋，輸出 `p\_hat` 和 `V`

        Parameters

        ----------

        x: Tensor of shape (N, C, H, W)

            棋局的狀態特徵平面張量

        Returns

        -------

        p\_hat: Tensor of shape (N, board\_len^2)

            對數先驗概率向量

        value: Tensor of shape (N, 1)

            當前局面的估值

        """

        x = self.conv(x)

        x = self.residues(x)

        p\_hat = self.policy\_head(x)

        value = self.value\_head(x)

        return p\_hat, value

    def predict(self, chess\_board: ChessBoard):

        """ 獲取當前局面上所有可用 `action` 和他對應的先驗概率 `P(s, a)`，以及局面的 `value`

        Parameters

        ----------

        chess\_board: ChessBoard

            棋盤

        Returns

        -------

        probs: `np.ndarray` of shape `(len(chess\_board.available\_actions), )`

            當前局面上所有可用 `action` 對應的先驗概率 `P(s, a)`

        value: float

            當前局面的估值

        """

        feature\_planes = chess\_board.get\_feature\_planes().to(self.device)

        feature\_planes.unsqueeze\_(0)

        p\_hat, value = self(feature\_planes)

        # 將對數概率轉換為概率

        p = torch.exp(p\_hat).flatten()

        # 只取可行的落點

        if self.is\_use\_gpu:

            p = p[chess\_board.available\_actions].cpu().detach().numpy()

        else:

            p = p[chess\_board.available\_actions].detach().numpy()

        return p, value[0].item()

    def set\_device(self, is\_use\_gpu: bool):

        """ 設置神經網路運行設備 """

        self.is\_use\_gpu = is\_use\_gpu

        self.device = torch.device('cuda:0' if is\_use\_gpu else 'cpu')

# coding: utf-8

import random

import numpy as np

from .chess\_board import ChessBoard

from .node import Node

class RolloutMCTS:

    """ 基於隨機走棋策略的蒙特卡洛樹搜索 """

    def \_\_init\_\_(self, c\_puct: float = 5, n\_iters=1000):

        """

        Parameters

        ----------

        c\_puct: float

            探索常數

        n\_iters: int

            迭代搜索次數

        """

        self.c\_puct = c\_puct

        self.n\_iters = n\_iters

        self.root = Node(1, c\_puct, parent=None)

    def get\_action(self, chess\_board: ChessBoard) -> int:

        """ 根据當前局面返回下一步動作

        Parameters

        ----------

        chess\_board: ChessBoard

            棋盤

        """

        for i in range(self.n\_iters):

            # 複製一个棋盤用来模擬

            board = chess\_board.copy()

            # 如果没有遇到葉節點，就一直向下搜索並更新棋盤

            node = self.root

            while not node.is\_leaf\_node():

                action, node = node.select()

                board.do\_action(action)

rollout\_mcts.py

            # 判斷遊戲是否结束，如果没结束就拓展葉節點

            is\_over, winner = board.is\_game\_over()

            if not is\_over:

                node.expand(self.\_\_default\_policy(board))

            # 模擬

            value = self.\_\_rollout(board)

            # 反向傳播

            node.backup(-1\*value)

        # 根據子節點的訪問次數来選擇動作

        action = max(self.root.children.items(), key=lambda i: i[1].N)[0]

        # 更新根節點

        self.root = Node(prior\_prob=1)

        return action

    def \_\_default\_policy(self, chess\_board: ChessBoard):

        """ 根据當前局面返回可進行的動作及其概率

        Returns

        -------

        action\_probs: List[Tuple[int, float]]

            每個元素都為 `(action, prior\_prob)` 元组，根據這個元组創建子節點，

            `action\_probs` 的長度為當前棋盤的可用落點的總數

        """

        n = len(chess\_board.available\_actions)

        probs = np.ones(n) / n

        return zip(chess\_board.available\_actions, probs)

    def \_\_rollout(self, board: ChessBoard):

        """ 快速走棋，模擬一局 """

        current\_player = board.current\_player

        while True:

            is\_over, winner = board.is\_game\_over()

            if is\_over:

                break

            action = random.choice(board.available\_actions)

            board.do\_action(action)

        # 計算 Value，平局為 0，當前玩家勝利為 1, 輸為 -1

        if winner is not None:

            return 1 if winner == current\_player else -1

        return 0

self\_play\_dataset.py

# coding:utf-8

from collections import deque, namedtuple

import torch

from torch import Tensor

from torch.utils.data import Dataset

SelfPlayData = namedtuple(

    'SelfPlayData', ['pi\_list', 'z\_list', 'feature\_planes\_list'])

class SelfPlayDataSet(Dataset):

    """ 自我博弈數據集類，每個樣本為元组 `(feature\_planes, pi, z)` """

    def \_\_init\_\_(self, board\_len=10):

        super().\_\_init\_\_()

        self.\_\_data\_deque = deque(maxlen=10000)

        self.board\_len = board\_len

    def \_\_len\_\_(self):

        return len(self.\_\_data\_deque)

    def \_\_getitem\_\_(self, index):

        return self.\_\_data\_deque[index]

    def clear(self):

        """ 清空數據集 """

        self.\_\_data\_deque.clear()

    def append(self, self\_play\_data: SelfPlayData):

        """ 向數據集中插入數據 """

        n = self.board\_len

        z\_list = Tensor(self\_play\_data.z\_list)

        pi\_list = self\_play\_data.pi\_list

        feature\_planes\_list = self\_play\_data.feature\_planes\_list

        # 使用翻轉和鏡像擴充已有數據集

        for z, pi, feature\_planes in zip(z\_list, pi\_list, feature\_planes\_list):

            for i in range(4):

                # 逆時針旋轉 i\*90°

                rot\_features = torch.rot90(Tensor(feature\_planes), i, (1, 2))

                rot\_pi = torch.rot90(Tensor(pi.reshape(n, n)), i)

                self.\_\_data\_deque.append(

                    (rot\_features, rot\_pi.flatten(), z))

                # 對逆時針旋轉後的數組進行水平翻轉

                flip\_features = torch.flip(rot\_features, [2])

                flip\_pi = torch.fliplr(rot\_pi)

                self.\_\_data\_deque.append(

                    (flip\_features, flip\_pi.flatten(), z))

train.py

# coding:utf-8

import json

import os

import time

import traceback

import torch

import torch.nn.functional as F

from torch import nn, optim, cuda

from torch.optim.lr\_scheduler import MultiStepLR

from torch.utils.data import DataLoader

from .alpha\_zero\_mcts import AlphaZeroMCTS

from .chess\_board import ChessBoard

from .policy\_value\_net import PolicyValueNet

from .self\_play\_dataset import SelfPlayData, SelfPlayDataSet

def save\_model(train\_func):

    """ 保存模型 """

    def wrapper(train\_pipe\_line, \*args, \*\*kwargs):

        try:

            train\_func(train\_pipe\_line)

        except BaseException as e:

            if not isinstance(e, KeyboardInterrupt):

                traceback.print\_exc()

            os.makedirs('model', exist\_ok=True)

            t = time.strftime('%Y-%m-%d\_%H-%M-%S',

                              time.localtime(time.time()))

            path = f'model/last\_policy\_value\_net\_{t}.pth'

            train\_pipe\_line.policy\_value\_net.eval()

            torch.save(train\_pipe\_line.policy\_value\_net, path)

            print(f'🎉 訓練结束，已將當前模型保存到 {os.path.join(os.getcwd(), path)}')

            # 保存數據

            with open('log/train\_losses.json', 'w', encoding='utf-8') as f:

                json.dump(train\_pipe\_line.train\_losses, f)

            if train\_pipe\_line.is\_save\_game:

                with open('log/games.json', 'w', encoding='utf-8') as f:

                    json.dump(train\_pipe\_line.games, f)

    return wrapper

class PolicyValueLoss(nn.Module):

    """ 根據 self-play 產生的 `z` 和 `π` 計算誤差 """

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def forward(self, p\_hat, pi, value, z):

        """ 前饋

        Parameters

        ----------

        p\_hat: Tensor of shape (N, board\_len^2)

            對數動作概率向量

        pi: Tensor of shape (N, board\_len^2)

            `mcts` 產生的動作概率向量

        value: Tensor of shape (N, )

            對每個局面的估值

        z: Tensor of shape (N, )

            最终的遊戲结果相對每一個玩家的獎賞

        """

        value\_loss = F.mse\_loss(value, z)

        policy\_loss = -torch.sum(pi\*p\_hat, dim=1).mean()

        loss = value\_loss + policy\_loss

        return loss

class TrainModel:

    """ 訓練模型 """

    def \_\_init\_\_(self, board\_len=10, lr=0.01, n\_self\_plays=2000, n\_mcts\_iters=500,

                 n\_feature\_planes=4, batch\_size=512, start\_train\_size=500, check\_frequency=50,

                 n\_test\_games=20, c\_puct=4, is\_use\_gpu=True, is\_save\_game=True, \*\*kwargs):

        """

        Parameters

        ----------

        board\_len: int

            棋盤大小

        lr: float

            學習率

        n\_self\_plays: int

            自我博弈遊戲局数

        n\_mcts\_iters: int

            蒙特卡洛樹搜索次數

        n\_feature\_planes: int

            特徵平面個數

        batch\_size: int

            mini-batch 的大小

        start\_train\_size: int

            開始訓練模型時的最小數據集尺寸

        check\_frequency: int

           測試模型的頻率

        n\_test\_games: int

            測試模型時與歷史最優模型的比赛局數

        c\_puct: float

            探索常數

        is\_use\_gpu: bool

            是否使用 GPU

        is\_save\_game: bool

            是否保存自對弈的棋譜

        """

        self.c\_puct = c\_puct

        self.is\_use\_gpu = is\_use\_gpu

        self.batch\_size = batch\_size

        self.n\_self\_plays = n\_self\_plays

        self.n\_test\_games = n\_test\_games

        self.n\_mcts\_iters = n\_mcts\_iters

        self.is\_save\_game = is\_save\_game

        self.check\_frequency = check\_frequency

        self.start\_train\_size = start\_train\_size

        self.device = torch.device(

            'cuda:0' if is\_use\_gpu and cuda.is\_available() else 'cpu')

        self.chess\_board = ChessBoard(board\_len, n\_feature\_planes)

        # 實例化策略-價值網路和蒙特卡洛搜索樹

        self.policy\_value\_net = self.\_\_get\_policy\_value\_net()

        self.mcts = AlphaZeroMCTS(

            self.policy\_value\_net, c\_puct=c\_puct, n\_iters=n\_mcts\_iters, is\_self\_play=True)

        # 創建優化器和損失函數

        self.optimizer = optim.Adam(

            self.policy\_value\_net.parameters(), lr=lr, weight\_decay=1e-4)

        self.criterion = PolicyValueLoss()

        self.lr\_scheduler = MultiStepLR(

            self.optimizer, [1500, 2500], gamma=0.1)

        # 實例化數據集

        self.dataset = SelfPlayDataSet(board\_len)

        # 記錄數據

        self.train\_losses = self.\_\_load\_data('log\\train\_losses.json')

        self.games = self.\_\_load\_data('log\\games.json')

    def \_\_self\_play(self):

        """ 自我博弈一局

        Returns

        -------

        self\_play\_data: namedtuple

            自我博弈數據，有以下三個成員:

            \* `pi\_list`: 蒙特卡洛樹搜索產生的動作概率向量 π 组成的列表

            \* `z\_list`: 一局之中每個動作的玩家相對最後的遊戲结果的獎賞列表

            \* `feature\_planes\_list`: 一局之中每個動作對應的特徵平面组成的列表

        """

        # 初始化棋盤和數據容器

        self.policy\_value\_net.eval()

        self.chess\_board.clear\_board()

        pi\_list, feature\_planes\_list, players = [], [], []

        action\_list = []

        # 開始一局遊戲

        while True:

            action, pi = self.mcts.get\_action(self.chess\_board)

            # 保存每一步的數據

            feature\_planes\_list.append(self.chess\_board.get\_feature\_planes())

            players.append(self.chess\_board.current\_player)

            action\_list.append(action)

            pi\_list.append(pi)

            self.chess\_board.do\_action(action)

            # 判斷遊戲是否结束

            is\_over, winner = self.chess\_board.is\_game\_over()

            if is\_over:

                if winner is not None:

                    z\_list = [1 if i == winner else -1 for i in players]

                else:

                    z\_list = [0]\*len(players)

                break

        # 重置根節點

        self.mcts.reset\_root()

        # 返回數據

        if self.is\_save\_game:

            self.games.append(action\_list)

        self\_play\_data = SelfPlayData(

            pi\_list=pi\_list, z\_list=z\_list, feature\_planes\_list=feature\_planes\_list)

        return self\_play\_data

    @save\_model

    def train(self):

        """ 訓練模型 """

        for i in range(self.n\_self\_plays):

            print(f'🏹 正在進行第 {i+1} 局自我博弈遊戲...')

            self.dataset.append(self.\_\_self\_play())

            # 如果數據集中的數據量大於 start\_train\_size 就進行一次訓練

            if len(self.dataset) >= self.start\_train\_size:

                data\_loader = iter(DataLoader(

                    self.dataset, self.batch\_size, shuffle=True, drop\_last=False))

                print('💊 开始训练...')

                self.policy\_value\_net.train()

                # 隨機選出一批數據来訓練，防止過度擬合

                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):

                    # 前饋

                    p\_hat, value = self.policy\_value\_net(feature\_planes)

                    # 梯度清零

                    self.optimizer.zero\_grad()

                    # 計算損失

                    loss = self.criterion(p\_hat, pi, value.flatten(), z)

                    # 誤差反向傳播

                    loss.backward()

                    # 更新参数

                    self.optimizer.step()

                    # 學習率退火

                    self.lr\_scheduler.step()

                # 紀錄誤差

                self.train\_losses.append([i, loss.item()])

                print(f"🚩 train\_loss = {loss.item():<10.5f}\n")

            # 測試模型

            if (i+1) % self.check\_frequency == 0:

                self.\_\_test\_model()

    def \_\_test\_model(self):

        """ 測試模型 """

        model\_path = 'model\\best\_policy\_value\_net.pth'

        # 如果最佳模型不存在保存當前模型為最佳模型

        if not os.path.exists(model\_path):

            torch.save(self.policy\_value\_net, model\_path)

            return

        # 載入歷史最優模型

        best\_model = torch.load(model\_path)  # type:PolicyValueNet

        best\_model.eval()

        best\_model.set\_device(self.is\_use\_gpu)

        mcts = AlphaZeroMCTS(best\_model, self.c\_puct, self.n\_mcts\_iters)

        self.mcts.set\_self\_play(False)

        self.policy\_value\_net.eval()

        # 開始比赛

        print('🩺 正在測試當前模型...')

        n\_wins = 0

        for i in range(self.n\_test\_games):

            self.chess\_board.clear\_board()

            self.mcts.reset\_root()

            mcts.reset\_root()

            while True:

                # 當前模型走一步

                is\_over, winner = self.\_\_do\_mcts\_action(self.mcts)

                if is\_over:

                    n\_wins += int(winner == ChessBoard.BLACK)

                    break

                # 歷史最優模型走一步

                is\_over, winner = self.\_\_do\_mcts\_action(mcts)

                if is\_over:

                    break

        # 如果勝率大於 55%，就保存當前模型為最优模型

        win\_prob = n\_wins/self.n\_test\_games

        if win\_prob > 0.55:

            torch.save(self.mcts.policy\_value\_net, model\_path)

            print(f'🥇 保存當前模型為最优模型，當前模型胜率為：{win\_prob:.1%}\n')

        else:

            print(f'🎃 保持历史最优模型不变，當前模型胜率為：{win\_prob:.1%}\n')

        self.mcts.set\_self\_play(True)

    def \_\_do\_mcts\_action(self, mcts):

        """ 獲取動作 """

        action = mcts.get\_action(self.chess\_board)

        self.chess\_board.do\_action(action)

        is\_over, winner = self.chess\_board.is\_game\_over()

        return is\_over, winner

    def \_\_get\_policy\_value\_net(self):

        """ 創建策略-價值網路，如果存在歷史最優模型則直接載入最優模型 """

        best\_model = 'best\_policy\_value\_net.pth'

        history\_models = sorted(

            [i for i in os.listdir('model') if i.startswith('last')])

        # 從歷史模型中選取最新模型

        model = history\_models[-1] if history\_models else best\_model

        model = f'model\\{model}'

        if os.path.exists(model):

            print(f'💎 載入模型 {model} ...\n')

            net = torch.load(model).to(self.device)  # type:PolicyValueNet

            net.set\_device(self.is\_use\_gpu)

        else:

            net = PolicyValueNet(n\_feature\_planes=self.chess\_board.n\_feature\_planes,

                                 is\_use\_gpu=self.is\_use\_gpu).to(self.device)

        return net

    def \_\_load\_data(self, path: str):

        """ 載入歷史損失數據 """

        data = []

        try:

            with open(path, encoding='utf-8') as f:

                data = json.load(f)

        except:

            os.makedirs('log', exist\_ok=True)

        return data

game.py

# coding:utf-8

import sys

import torch

from PyQt5.QtCore import Qt, QTranslator, QLocale

from PyQt5.QtWidgets import QApplication

from app.View.main\_window import MainWindow

app = QApplication(sys.argv)

app.setAttribute(Qt.AA\_DontCreateNativeWidgetSiblings)

# 設置語言

translator = QTranslator()

translator.load(QLocale.system(), ':/i18n/AlphaGobangZero\_')

app.installTranslator(translator)

# 創建主界面

w = MainWindow(board\_len=10)

w.show()

sys.exit(app.exec\_())

train.py

# coding: utf-8

from alphazero.train import TrainModel

train\_config = {

    'lr': 1e-2,

    'c\_puct': 3,

    'board\_len': 10,

    'batch\_size': 512,

    'is\_use\_gpu': True,

    'n\_test\_games': 10,

    'n\_mcts\_iters': 500,

    'n\_self\_plays': 2000,

    'is\_save\_game': False,

    'n\_feature\_planes': 6,

    'check\_frequency': 50,

    'start\_train\_size': 500

}

train\_model = TrainModel(\*\*train\_config)

train\_model.train()