Contents

[rules.py 2](#_Toc211794826)

[NeuralNetData.py 7](#_Toc211794827)

[selfplay.py 16](#_Toc211794828)

[dataprocessor.py 25](#_Toc211794829)

[minimax.py 28](#_Toc211794830)

[animate\_game.py 31](#_Toc211794831)

[display.py 32](#_Toc211794832)

[EvalEpochs.py 34](#_Toc211794833)

[NNEvaluation.py 38](#_Toc211794834)

# rules.py

import math

import random

connections = {

        (0, 0): [(0, 3), (3, 0)],

        (0, 3): [(0, 0), (0, 6), (1, 3)],

        (0, 6): [(0, 3), (3, 6)],

        (1, 1): [(1, 3), (3, 1)],

        (1, 3): [(1, 1), (1, 5), (0, 3), (2, 3)],

        (1, 5): [(1, 3), (3, 5)],

        (2, 2): [(2, 3), (3, 2)],

        (2, 3): [(2, 2), (2, 4), (1, 3)],

        (2, 4): [(2, 3), (3, 4)],

        (3, 0): [(0, 0), (3, 1), (6, 0)],

        (3, 1): [(3, 0), (3, 2), (1, 1), (5, 1)],

        (3, 2): [(3, 1), (4, 2), (2, 2)],

        (3, 4): [(2, 4), (3, 5), (4, 4)],

        (3, 5): [(3, 4), (3, 6), (1, 5), (5, 5)],

        (3, 6): [(0, 6), (3, 5), (6, 6)],

        (4, 2): [(3, 2), (4, 3)],

        (4, 3): [(4, 2), (4, 4), (5, 3)],

        (4, 4): [(4, 3), (3, 4)],

        (5, 1): [(3, 1), (5, 3)],

        (5, 3): [(5, 1), (5, 5), (4, 3), (6, 3)],

        (5, 5): [(5, 3), (3, 5)],

        (6, 0): [(3, 0), (6, 3)],

        (6, 3): [(6, 0), (6, 6), (5, 3)],

        (6, 6): [(3, 6), (6, 3)],

    }

mill\_lines = [

    [(0, 0), (0, 3), (0, 6)],

    [(1, 1), (1, 3), (1, 5)],

    [(2, 2), (2, 3), (2, 4)],

    [(3, 0), (3, 1), (3, 2)],

    [(3, 4), (3, 5), (3, 6)],

    [(4, 2), (4, 3), (4, 4)],

    [(5, 1), (5, 3), (5, 5)],

    [(6, 0), (6, 3), (6, 6)],

    [(0, 0), (3, 0), (6, 0)],

    [(1, 1), (3, 1), (5, 1)],

    [(2, 2), (3, 2), (4, 2)],

    [(0, 3), (1, 3), (2, 3)],

    [(4, 3), (5, 3), (6, 3)],

    [(2, 4), (3, 4), (4, 4)],

    [(1, 5), (3, 5), (5, 5)],

    [(0, 6), (3, 6), (6, 6)],

]

VALID\_POSITIONS = [

    (0, 0), (0, 3), (0, 6),

    (1, 1), (1, 3), (1, 5),

    (2, 2), (2, 3), (2, 4),

    (3, 0), (3, 1), (3, 2), (3, 4), (3, 5), (3, 6),

    (4, 2), (4, 3), (4, 4),

    (5, 1), (5, 3), (5, 5),

    (6, 0), (6, 3), (6, 6)

]

def evaluate\_board(board, player):

    opponent = 1 if player == 2 else 2

    score = 0

    for row in board:

        for cell in row:

            if cell == player:

                score += 1

            if cell == opponent:

                score -= 1

    '''

    player\_mobility = count\_mobility(board, player)

    opponent\_mobility = count\_mobility(board, opponent)

    score += 0.1 \* (player\_mobility - opponent\_mobility)

    '''

    return score

def count\_potential\_mills(board, player, mill\_lines):

    count = 0

    for mill\_line in mill\_lines:

        values = [board[row][col] for row, col in mill\_line]

        if values.count(player) == 2 and values.count(0) == 1:

            count += 1

    return count

def count\_mill(board, player, mill\_lines):

    count = 0

    for line in mill\_lines:

        if all(board[i][j] == player for i, j in line):

            count += 1

    return count

def is\_mill(board, player, mill\_lines, move):

    lines = []

    for mills in mill\_lines:

        if move in mills:

            lines.append(mills)

    if count\_mill(board, player, lines) != 0:

        return True

    else:

        return False

def count\_mobility(board, player):

    mobility = 0

    for i in range(len(board)):

        for j in range(len(board[i])):

            if board[i][j] == player:

                mobility += sum(1 for ni, nj in connections.get((i, j), []) if board[ni][nj] == 0)

    return mobility

def get\_possible\_moves(board, player, move\_count):

    if move\_count < 18:

        moves = []

        for i in range(len(board)):

            for j in range(len(board[i])):

                if board[i][j] == 0:

                    moves.append((i, j))

        return moves

    else:

        possible\_moves = []

        for i in range(len(board)):

            for j in range(len(board[i])):

                if board[i][j] == player:

                    for ni, nj in connections.get((i, j), []):

                        if board[ni][nj] == 0:

                            possible\_moves.append([(i, j), (ni, nj)])

        return possible\_moves

def apply\_move(board, move, player, deterministic=False):

    # Normalize move into a canonical form

    norm = None

    if isinstance(move, tuple):

        if len(move) == 2 and all(isinstance(x, int) for x in move):

            # placement

            norm = ('place', move)

        elif len(move) == 4 and all(isinstance(x, int) for x in move):

            # movement as 4-tuple

            norm = ('move', ((move[0], move[1]), (move[2], move[3])))

        elif len(move) == 2 and all(isinstance(x, tuple) and len(x) == 2 for x in move):

            # movement as tuple of tuples

            norm = ('move', (move[0], move[1]))

    elif isinstance(move, list):

        if len(move) == 2 and all(isinstance(x, tuple) and len(x) == 2 for x in move):

            # movement as list of tuples

            norm = ('move', (move[0], move[1]))

    if norm is None:

        # everything breaks more gracefully now

        return (False, f"Invalid move format: {type(move)} {move}")

    kind, payload = norm

    if kind == 'place':

        row, col = payload

        if board[row][col] != 0:

            return (False, "Spot not empty")

        board[row][col] = player

        if is\_mill(board, player, mill\_lines, (row, col)):

            remove\_random\_opponent\_piece(board, player, deterministic=deterministic)

        return (True, board)

    else:

        (row1, col1), (row2, col2) = payload

        if (row1, col1) == (row2, col2):

            return (False, "Cant move there")

        if board[row1][col1] != player:

            return (False, "No piece there")

        if board[row2][col2] != 0:

            return (False, "Target not empty")

        # Execute move

        board[row1][col1] = 0

        board[row2][col2] = player

        if is\_mill(board, player, mill\_lines, (row2, col2)):

            remove\_random\_opponent\_piece(board, player, deterministic=deterministic)

        return (True, board)

def remove\_random\_opponent\_piece(board, player, deterministic: bool = False):

    opponent = 1 if player == 2 else 2

    opponent\_positions = [(i, j) for i in range(len(board)) for j in range(len(board[i])) if board[i][j] == opponent]

    # Check if all are mills

    all\_in\_mills = all(is\_mill(board, opponent, mill\_lines, pos) for pos in opponent\_positions)

    removable = [pos for pos in opponent\_positions if all\_in\_mills or not is\_mill(board, opponent, mill\_lines, pos)]

    if removable:

        if deterministic:

            # Deterministic choice for search stability: choose the first in scan order

            pos = removable[0]

        else:

            pos = random.choice(removable)

        board[pos[0]][pos[1]] = 0

def check\_game\_over(board, move\_count):

    count1 = sum(cell == 1 for row in board for cell in row)

    count2 = sum(cell == 2 for row in board for cell in row)

    if ((count1 < 3) or (count2 < 3)) and move\_count > 18:

        return True

    else:

        return False

def get\_actions():

    actions = []

    for element in VALID\_POSITIONS:

        actions.append(element)

    for start, ends in connections.items():

            for end in ends:

                actions.append([start, end])

    return actions

def board\_to\_key(board):

    return str(tuple(tuple(row) for row in board))

def count\_pieces(board):

    count1 = sum(cell == 1 for row in board for cell in row)

    count2 = sum(cell == 2 for row in board for cell in row)

    return count1, count2

def state\_to\_board(state):

    board = [[0 for \_ in range(7)] for \_ in range(7)]

    index = 0

    for i in range(7):

        for j in range(7):

            if (i, j) in VALID\_POSITIONS:

                board[i][j] = state[index]

                index += 1

    player1\_pieces, player2\_pieces = state[24], state[25]

    if player1\_pieces + player2\_pieces > 0:

        move\_count = 18 - player1\_pieces - player2\_pieces

    else:

        move\_count = 20

    return board, move\_count

# NeuralNetData.py

import os

import ast

import random

import pygame

import torch

import torch.nn as nn

import torch.optim as optim

import rules

from torch.utils.data import TensorDataset, DataLoader

import time

# Configuration

class TrainingConfig:

    def \_\_init\_\_(self):

        self.EPOCHS = 10000

        self.LEARNING\_RATE = 1e-4

        self.WEIGHT\_DECAY = 1e-5

        self.GRAD\_ACCUM\_STEPS = 1       # >1 simulates small batch size

        self.SAVE\_EVERY = 10            # save model every N epochs

        self.DATASET\_PATH = r"D:\Matura\NNnewer\sepember\minimax\_training\_set\_symmetries.txt"

        self.MODEL\_SAVE\_DIR = "checkpoints\_batch512\_s"

        self.LOG\_INTERVAL = 1000        # steps between console/pygame updates

        self.MAX\_LINES = 500000         # max lines to read from dataset (None=all)

        self.RESTART\_EPOCH = 5880       # epoch to resume training (None=from scratch)

        self.BATCH\_SIZE = 1024

# Model

class BiggerPolicyNetwork(nn.Module):

    def \_\_init\_\_(self, state\_size, action\_size):

        super(BiggerPolicyNetwork, self).\_\_init\_\_()

        '''

        self.network = nn.Sequential(

            nn.Linear(state\_size, 512),

            nn.LayerNorm(512),

            nn.GELU(),

            nn.Dropout(0.3),

            nn.Linear(512, 256),

            nn.LayerNorm(256),

            nn.GELU(),

            nn.Dropout(0.3),

            nn.Linear(256, 128),

            nn.LayerNorm(128),

            nn.GELU(),

            nn.Linear(128, action\_size)

        )

        '''

        self.network = nn.Sequential(

            nn.Linear(state\_size, 1024),

            nn.LayerNorm(1024),

            nn.GELU(),

            nn.Dropout(0.3),

            nn.Linear(1024, 1024),

            nn.LayerNorm(1024),

            nn.GELU(),

            nn.Dropout(0.3),

            nn.Linear(1024, 512),

            nn.LayerNorm(512),

            nn.GELU(),

            nn.Dropout(0.3),

            nn.Linear(512, 256),

            nn.LayerNorm(256),

            nn.GELU(),

            nn.Dropout(0.2),

            nn.Linear(256, 128),

            nn.LayerNorm(128),

            nn.GELU(),

            nn.Linear(128, action\_size)

        )

    def forward(self, x):

        return self.network(x)

# Data Loading

class GameDataLoader:

    def \_\_init\_\_(self, config):

        self.config = config

        self.actions = rules.get\_actions()

    def load\_dataset(self):

        """Load and preprocess dataset"""

        print(f"Loading dataset from {self.config.DATASET\_PATH}")

        states = []

        labels = []

        with open(self.config.DATASET\_PATH, "r") as f:

            for i, line in enumerate(f):

                if self.config.MAX\_LINES and i >= self.config.MAX\_LINES:

                    break

                if not line.strip():

                    continue

                state, move = ast.literal\_eval(line.strip())

                # Convert state to tensor

                state\_tensor = torch.FloatTensor(state)

                # Convert move to action index

                if isinstance(move, (tuple, list)):

                    action\_index = self.actions.index(move)

                else:

                    raise ValueError(f"Unexpected move format: {move}")

                states.append(state\_tensor)

                labels.append(action\_index)

        print(f"Loaded {len(states)} samples")

        return states, labels

    def create\_dataloader(self, states, labels):

        """Create PyTorch DataLoader"""

        dataset = TensorDataset(torch.stack(states), torch.LongTensor(labels))

        if self.config.BATCH\_SIZE > 1:

            return DataLoader(dataset, batch\_size=self.config.BATCH\_SIZE, shuffle=True)

        else:

            return list(zip(torch.stack(states), torch.LongTensor(labels)))

# Training Utils

class TrainingUtils:

    def accuracy\_topk(output, target, topk=(1,)):

        maxk = max(topk)

        \_, pred = output.topk(maxk, 1, True, True)

        pred = pred.t()

        correct = pred.eq(target.view(1, -1).expand\_as(pred))

        res = []

        for k in topk:

            correct\_k = correct[:k].reshape(-1).float().sum(0)

            res.append(correct\_k.item())

        return res

    def format\_seconds(seconds: int) -> str:

        days, seconds = divmod(seconds, 86400)

        hours, seconds = divmod(seconds, 3600)

        minutes, seconds = divmod(seconds, 60)

        parts = []

        if days:

            parts.append(f"{days}d")

        if hours:

            parts.append(f"{hours}h")

        if minutes:

            parts.append(f"{minutes}m")

        if seconds or not parts:  # show 0s if nothing else

            parts.append(f"{seconds}s")

        return " ".join(parts)

# Pygame Monitor

class TrainingMonitor:

    def \_\_init\_\_(self):

        pygame.init()

        self.width, self.height = (800, 600)

        self.screen = pygame.display.set\_mode((self.width, self.height))

        pygame.display.set\_caption("Training Monitor")

        self.font = pygame.font.SysFont("Arial", 16)

        self.loss\_history = []

        self.acc\_history = []

    def update(self, step, loss, acc, state=None, target=None, pred=None):

        self.loss\_history.append(loss)

        self.acc\_history.append(acc)

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                pygame.quit()

                exit()

        # clear

        self.screen.fill((30, 30, 30))

        # plot loss

        if len(self.loss\_history) > 1:

            points = []

            for i, loss in enumerate(self.loss\_history[-200:]):  # last 200 points

                x = i \* (self.width // 2) // min(200, len(self.loss\_history))

                y = int(self.height // 2 - loss \* 100)  # scale factor

                y = max(0, min(self.height // 2, y))

                points.append((x, y))

            if len(points) > 1:

                pygame.draw.lines(self.screen, (255, 100, 100), False, points, 2)

        # plot accuracy

        if len(self.acc\_history) > 1:

            points = []

            for i, acc in enumerate(self.acc\_history[-200:]):

                x = i \* (self.width // 2) // min(200, len(self.acc\_history))

                y = int(self.height // 2 + (1 - acc) \* 100)  # scale factor

                y = max(self.height // 2, min(self.height, y))

                points.append((x, y))

            if len(points) > 1:

                pygame.draw.lines(self.screen, (100, 255, 100), False, points, 2)

        # text info

        info\_lines = [

            f"Step: {step}",

            f"Loss: {loss:.4f}",

            f"Acc: {acc:.2%}",

        ]

        if target is not None and pred is not None:

            info\_lines.extend([

                f"Target: {target}",

                f"Pred: {pred}",

                f"Correct: {target == pred}"

            ])

        for i, line in enumerate(info\_lines):

            text = self.font.render(line, True, (255, 255, 255))

            self.screen.blit(text, (self.width // 2 + 10, 10 + i \* 20))

        pygame.display.flip()

# Main Trainer Class

class NeuralNetworkTrainer:

    def \_\_init\_\_(self, config=None):

        self.config = config or TrainingConfig()

        self.device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

        print(f"Using device: {self.device}")

        # Initialize components

        self.data\_loader = GameDataLoader(self.config)

        self.utils = TrainingUtils()

        self.actions = rules.get\_actions()

        # Create model save directory

        os.makedirs(self.config.MODEL\_SAVE\_DIR, exist\_ok=True)

        # Initialize model

        self.model = BiggerPolicyNetwork(28, len(self.actions)).to(self.device)

        self.criterion = nn.CrossEntropyLoss()

        self.optimizer = optim.AdamW(

            self.model.parameters(),

            lr=self.config.LEARNING\_RATE,

            weight\_decay=self.config.WEIGHT\_DECAY

        )

        # Load checkpoint if specified

        self.start\_epoch = self.\_load\_checkpoint()

        # Initialize monitor

        self.monitor = TrainingMonitor()

    def \_load\_checkpoint(self):

        """Load model checkpoint if specified"""

        if self.config.RESTART\_EPOCH is not None:

            checkpoint\_path = os.path.join(

                self.config.MODEL\_SAVE\_DIR,

                f"model\_epoch{self.config.RESTART\_EPOCH}.pth"

            )

            if os.path.exists(checkpoint\_path):

                self.model.load\_state\_dict(torch.load(checkpoint\_path, map\_location=self.device))

                print(f"Resumed training from checkpoint: {checkpoint\_path}")

                return self.config.RESTART\_EPOCH

            else:

                print(f"Checkpoint not found: {checkpoint\_path}")

        return 0

    def \_save\_checkpoint(self, epoch):

        """Save model checkpoint"""

        path = os.path.join(self.config.MODEL\_SAVE\_DIR, f"model\_epoch{epoch + self.start\_epoch}.pth")

        torch.save(self.model.state\_dict(), path)

        return path

    def \_train\_single\_batch(self, state, label, step):

        """Train on a single sample"""

        state = state.to(self.device)

        label = label.to(self.device)

        outputs = self.model(state.unsqueeze(0))

        loss = self.criterion(outputs, label.unsqueeze(0))

        loss.backward()

        # Accumulate gradients

        if (step + 1) % self.config.GRAD\_ACCUM\_STEPS == 0:

            self.optimizer.step()

            self.optimizer.zero\_grad()

        # Calculate accuracy

        acc1, = self.utils.accuracy\_topk(outputs, label.unsqueeze(0), topk=(1,))

        return loss.item(), acc1, outputs, label

    def \_train\_batch(self, state\_batch, label\_batch):

        """Train on a batch of samples"""

        state\_batch = state\_batch.to(self.device)

        label\_batch = label\_batch.to(self.device)

        outputs = self.model(state\_batch)

        loss = self.criterion(outputs, label\_batch)

        loss.backward()

        self.optimizer.step()

        self.optimizer.zero\_grad()

        # Calculate accuracy

        acc1, = self.utils.accuracy\_topk(outputs, label\_batch, topk=(1,))

        return loss.item(), acc1, outputs, label\_batch

    def train(self):

        """Main training loop"""

        print("Starting training...")

        starttime = time.time()

        # Load dataset

        states, labels = self.data\_loader.load\_dataset()

        if self.config.BATCH\_SIZE > 1:

            dataloader = self.data\_loader.create\_dataloader(states, labels)

        else:

            dataset = self.data\_loader.create\_dataloader(states, labels)

        step = 0

        for epoch in range(1, self.config.EPOCHS + 1):

            running\_loss, running\_acc = 0.0, 0.0

            sample\_count = 0

            self.optimizer.zero\_grad()

            if self.config.BATCH\_SIZE == 1:

                # Single sample training

                random.shuffle(dataset)

                for i, (state, label) in enumerate(dataset):

                    loss, acc, outputs, target = self.\_train\_single\_batch(state, label, i)

                    running\_loss += loss

                    running\_acc += acc

                    step += 1

                    if step % self.config.LOG\_INTERVAL == 0:

                        avg\_loss = running\_loss / self.config.LOG\_INTERVAL

                        avg\_acc = running\_acc / self.config.LOG\_INTERVAL

                        self.monitor.update(

                            step, avg\_loss, avg\_acc,

                            state=state, target=target.item(),

                            pred=outputs.argmax(1).item()

                        )

                        running\_loss, running\_acc = 0.0, 0.0

            else:

                # Batch training

                for i, (state\_batch, label\_batch) in enumerate(dataloader):

                    loss, acc, outputs, targets = self.\_train\_batch(state\_batch, label\_batch)

                    running\_loss += loss \* state\_batch.size(0)

                    running\_acc += acc

                    sample\_count += state\_batch.size(0)

                    step += 1

                    if step % self.config.LOG\_INTERVAL == 0:

                        avg\_loss = running\_loss / sample\_count

                        avg\_acc = running\_acc / sample\_count

                        self.monitor.update(

                            step, avg\_loss, avg\_acc,

                            state=state\_batch[0], target=targets[0].item(),

                            pred=outputs.argmax(1).tolist()[0]

                        )

                        running\_loss, running\_acc = 0.0, 0.0

                        sample\_count = 0

            # Save checkpoint

            if epoch % self.config.SAVE\_EVERY == 0 or epoch == self.config.EPOCHS:

                checkpoint\_path = self.\_save\_checkpoint(epoch)

                elapsed\_time = time.time() - starttime

                remaining\_epochs = self.config.EPOCHS - epoch - self.start\_epoch

                print(f"Saved checkpoint: {checkpoint\_path}")

                print(f"Time elapsed: {self.utils.format\_seconds(int(elapsed\_time))}")

                if remaining\_epochs > 0:

                    predicted\_time = (elapsed\_time / epoch) \* remaining\_epochs

                    print(f"Runtime prediction: {self.utils.format\_seconds(int(predicted\_time))}")

        print("Training completed!")

def train():

    trainer = NeuralNetworkTrainer()

    trainer.train()

if \_\_name\_\_ == "\_\_main\_\_":

    train()

# selfplay.py

"""Self-play training loop for Nine Men's Morris.

This module runs self-play games between two instances of the same neural

network, stores experiences in a replay buffer, and trains a policy/Q-network

using a target network for stability. It relies on `rules.py` for game logic

and `NeuralNetData.py` for the model architecture.

"""

import os

import copy

import random

from collections import deque

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

import matplotlib.pyplot as plt

import rules

import NeuralNetData

import NNEvaluation

# Configuration

class SelfPlayConfig:

    # Self-play parameters

    SELFPLAY\_GAMES = 100            # Games to play per iteration

    TRAINING\_ITERATIONS = 1000      # Number of self-play iterations

    # Neural network parameters

    STATE\_SIZE = 28

    ACTION\_SIZE = 88

    LEARNING\_RATE = 1e-4

    BATCH\_SIZE = 64

    EPOCHS\_PER\_ITERATION = 10       # train steps per iteration

    # Experience replay

    MEMORY\_SIZE = 500\_000

    MIN\_MEMORY\_SIZE = 1\_000

    # Temperature

    TEMPERATURE = 0.3               # Higher => more exploration

    TEMPERATURE\_DECAY = 0.995

    MIN\_TEMPERATURE = 0.1

    # Model saving

    SAVE\_INTERVAL = 10              # Save every N iterations

    MODEL\_SAVE\_DIR = "selfplay\_checkpoints\_large"

    CHECKPOINT\_EPOCH = 0            # For naming only

    CHECKPOINT\_PATH = None          # Path to resume from (None => scratch)

    # Evaluation

    EVAL\_INTERVAL = 5

    EVAL\_GAMES = 50

class ExperienceBuffer:

    """Buffer to store game experiences for training."""

    def \_\_init\_\_(self, max\_size: int):

        self.buffer = deque(maxlen=max\_size)

    def add(self, state, action, reward, next\_state, done):

        self.buffer.append((state, action, reward, next\_state, done))

    def sample(self, batch\_size: int):

        return random.sample(self.buffer, min(batch\_size, len(self.buffer)))

    def size(self) -> int:

        return len(self.buffer)

class SelfPlayTrainer:

    def \_\_init\_\_(self, config: SelfPlayConfig):

        self.config = config

        self.device = torch.device("cpu")

        # Initialize neural networks

        self.policy\_net = NeuralNetData.BiggerPolicyNetwork(

            state\_size=config.STATE\_SIZE,

            action\_size=config.ACTION\_SIZE,

        ).to(self.device)

        self.target\_net = NeuralNetData.BiggerPolicyNetwork(

            state\_size=config.STATE\_SIZE,

            action\_size=config.ACTION\_SIZE,

        ).to(self.device)

        self.target\_net.load\_state\_dict(self.policy\_net.state\_dict())

        self.target\_net.eval()

        self.optimizer = optim.Adam(self.policy\_net.parameters(), lr=config.LEARNING\_RATE)

        # Load pre-trained model

        if config.CHECKPOINT\_PATH and os.path.exists(config.CHECKPOINT\_PATH):

            self.load\_checkpoint(config.CHECKPOINT\_PATH)

            print(f"Loaded model weights from: {config.CHECKPOINT\_PATH}")

        else:

            print("Starting training from scratch")

        # Experience buffer and training stats

        self.experience\_buffer = ExperienceBuffer(config.MEMORY\_SIZE)

        self.iteration\_win\_rate = []

        self.iteration\_wins = []

        self.temperature = config.TEMPERATURE

        # Ensure save directory exists

        os.makedirs(config.MODEL\_SAVE\_DIR, exist\_ok=True)

        self.actions = rules.get\_actions()

    def load\_checkpoint(self, checkpoint\_path: str):

        checkpoint = torch.load(checkpoint\_path, map\_location=self.device)

        self.policy\_net.load\_state\_dict(checkpoint)

        self.target\_net.load\_state\_dict(checkpoint)

    def state\_to\_tensor(self, board, move\_count: int, player: int):

        """game state as the 28-dimensional vector.

        Layout (28):

        - 24 values: board at `rules.VALID\_POSITIONS` (0 empty, 1 P1, 2 P2)

        - 4 values: [player1\_in\_hand, player2\_in\_hand, total\_on\_board\_p1, total\_on\_board\_p2]

        """

        board\_positions = [board[r][c] for (r, c) in rules.VALID\_POSITIONS]

        p1\_pool, p2\_pool = rules.count\_pieces(board)

        total\_p1 = sum(cell == 1 for row in board for cell in row)

        total\_p2 = sum(cell == 2 for row in board for cell in row)

        state = board\_positions + [p1\_pool, p2\_pool, total\_p1, total\_p2]

        return torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)

    def select\_action(self, board, move\_count: int, player: int, use\_exploration: bool = True):

        """Select an action using the policy network with optional exploration.

        Returns (selected\_move, action\_index) or (None, None) if no moves.

        """

        state\_tensor = self.state\_to\_tensor(board, move\_count, player)

        with torch.no\_grad():

            logits = self.policy\_net(state\_tensor)

            action\_probs = torch.softmax(logits, dim=1)

        # Get valid moves and map to actions

        valid\_moves = rules.get\_possible\_moves(board, player, move\_count)

        if not valid\_moves:

            return None, None

        valid\_indices = []

        for mv in valid\_moves:

            valid\_indices.append(self.actions.index(mv))

        # remove invalid moves

        masked = torch.zeros\_like(action\_probs)

        for idx in valid\_indices:

            masked[0, idx] = action\_probs[0, idx]

        if masked.sum() <= 0:

            # weird error that I managed to trigger

            for idx in valid\_indices:

                masked[0, idx] = 1.0 / len(valid\_indices)

        else:

            masked = masked / masked.sum()

        # Exploration using temperature

        if use\_exploration and self.temperature > self.config.MIN\_TEMPERATURE:

            scaled = torch.pow(masked, 1.0 / max(self.temperature, 1e-6))

            scaled = scaled / scaled.sum()

            action\_index = torch.multinomial(scaled, 1).item()

        else:

            action\_index = torch.argmax(masked, dim=1).item()

        return self.actions[action\_index], action\_index

    def play\_self\_game(self):

        board, move\_count, \_, board\_history, game\_moves = NNEvaluation.initialize\_variables()

        game\_states = []

        game\_actions = []

        winner = None

        current\_player = 1

        while True:

            state\_tensor = self.state\_to\_tensor(board, move\_count, current\_player)

            # Select an action

            move, action\_index = self.select\_action(board, move\_count, current\_player)

            if move is None:

                # failsafe for no legal moves

                winner = 2 if current\_player == 1 else 1

                break

            game\_states.append(state\_tensor.squeeze(0).detach().cpu().numpy())

            game\_actions.append(action\_index)

            # Apply move

            success, new\_board = rules.apply\_move(copy.deepcopy(board), move, current\_player, deterministic=True)

            if not success:

                # failsafe because weird errors

                winner = 2 if current\_player == 1 else 1

                break

            board = new\_board

            game\_moves.append(move)

            board\_history.append(rules.board\_to\_key(board))

            move\_count += 1

            # Terminal checks

            if rules.check\_game\_over(board, move\_count):

                winner = current\_player

                break

            if board\_history.count(rules.board\_to\_key(board)) >= 3:

                winner = None  # Draw by repetition

                break

            # Switch player

            current\_player = 2 if current\_player == 1 else 1

        # Assign rewards

        for i, (state\_vec, action\_idx) in enumerate(zip(game\_states, game\_actions)):

            player\_turn = 1 if i % 2 == 0 else 2

            if winner is None:

                reward = 0.0

            elif winner == player\_turn:

                reward = 1.0

            else:

                reward = -1.0

            next\_state = game\_states[i + 1] if i + 1 < len(game\_states) else np.zeros\_like(state\_vec)

            done = (i == len(game\_states) - 1)

            self.experience\_buffer.add(state\_vec, action\_idx, reward, next\_state, done)

        return winner, len(game\_moves)

    def train\_network(self) -> float:

        if self.experience\_buffer.size() < self.config.MIN\_MEMORY\_SIZE:

            return 0.0

        total\_loss = 0.0

        gamma = 0.99

        for \_ in range(self.config.EPOCHS\_PER\_ITERATION):

            batch = self.experience\_buffer.sample(self.config.BATCH\_SIZE)

            # Convert to tensors

            states = torch.tensor(np.array([e[0] for e in batch]), dtype=torch.float32, device=self.device)

            actions = torch.tensor(np.array([e[1] for e in batch]), dtype=torch.long, device=self.device)

            rewards = torch.tensor(np.array([e[2] for e in batch]), dtype=torch.float32, device=self.device)

            next\_states = torch.tensor(np.array([e[3] for e in batch]), dtype=torch.float32, device=self.device)

            dones = torch.tensor(np.array([e[4] for e in batch]), dtype=torch.bool, device=self.device)

            q\_values = self.policy\_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

            with torch.no\_grad():

                next\_q = self.target\_net(next\_states).max(1)[0]

                targets = rewards + gamma \* next\_q \* (~dones)

            loss = nn.MSELoss()(q\_values, targets)

            self.optimizer.zero\_grad()

            loss.backward()

            torch.nn.utils.clip\_grad\_norm\_(self.policy\_net.parameters(), max\_norm=1.0)

            self.optimizer.step()

            total\_loss += loss.item()

        return total\_loss / self.config.EPOCHS\_PER\_ITERATION

    def update\_target\_network(self):

        self.target\_net.load\_state\_dict(self.policy\_net.state\_dict())

    def evaluate\_against\_random(self):

        """Evaluate current model against a random player."""

        player1wins, player2wins, draws = NNEvaluation.match('2', '4', self.config.EVAL\_GAMES, self.policy\_net)[2:]

        win\_rate = player1wins / self.config.EVAL\_GAMES

        return win\_rate, player1wins, draws

    def train(self):

        for iteration in range(self.config.TRAINING\_ITERATIONS):

            print(f"\n--- Iteration {iteration + 1}/{self.config.TRAINING\_ITERATIONS} ---")

            wins = 0

            draws = 0

            total\_moves = 0

            # Play self-play games

            for \_ in range(self.config.SELFPLAY\_GAMES):

                winner, num\_moves = self.play\_self\_game()

                total\_moves += num\_moves

                if winner is None:

                    draws += 1

                elif winner in (1, 2):

                    wins += 1

            # Train from buffer

            avg\_loss = self.train\_network()

            # Periodically update target network

            if iteration % 5 == 0:

                self.update\_target\_network()

            # Decay exploration with temperature

            self.temperature = max(self.config.MIN\_TEMPERATURE, self.temperature \* self.config.TEMPERATURE\_DECAY)

            # Stats

            win\_rate = wins / self.config.SELFPLAY\_GAMES

            draw\_rate = draws / self.config.SELFPLAY\_GAMES

            avg\_moves = total\_moves / self.config.SELFPLAY\_GAMES

            self.iteration\_win\_rate.append(win\_rate)

            self.iteration\_wins.append(wins)

            print(f"Win rate: {win\_rate:.3f}, Draw rate: {draw\_rate:.3f}")

            print(f"Average moves per game: {avg\_moves:.1f}")

            print(f"Average loss: {avg\_loss:.6f}")

            print(f"Temperature: {self.temperature:.3f}")

            print(f"Experience buffer size: {self.experience\_buffer.size()}")

            # Save models

            if iteration % self.config.SAVE\_INTERVAL == 0:

                model\_path = os.path.join(self.config.MODEL\_SAVE\_DIR, f"selfplay\_model\_iter\_{iteration + self.config.CHECKPOINT\_EPOCH}.pth")

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

                print(f"Model saved to {model\_path}")

            # evaluation vs random

            if (iteration + 1) % self.config.EVAL\_INTERVAL == 0:

                wr\_rand, wins\_rand, draws\_rand = self.evaluate\_against\_random()

                losses\_rand = self.config.EVAL\_GAMES - wins\_rand - draws\_rand

                print(f"Eval vs Random: Win rate {wr\_rand:.3f} ({wins\_rand}W/{draws\_rand}D/{losses\_rand}L)")

        # Final save and plot

        final\_model\_path = os.path.join(self.config.MODEL\_SAVE\_DIR, "selfplay\_final\_model.pth")

        torch.save(self.policy\_net.state\_dict(), final\_model\_path)

        print(f"Final model saved to {final\_model\_path}")

        self.plot\_training\_progress()

    def plot\_training\_progress(self):

        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)

        plt.plot(self.iteration\_win\_rate)

        plt.title('Win Rate Over Training')

        plt.xlabel('Iteration')

        plt.ylabel('Win Rate')

        plt.grid(True)

        plt.subplot(1, 3, 2)

        plt.plot(self.iteration\_wins)

        plt.title('Wins Per Iteration')

        plt.xlabel('Iteration')

        plt.ylabel('Number of Wins')

        plt.grid(True)

        plt.subplot(1, 3, 3)

        temps = [self.config.TEMPERATURE \* (self.config.TEMPERATURE\_DECAY \*\* i) for i in range(len(self.iteration\_win\_rate))]

        plt.plot(temps)

        plt.title('Temperature Decay')

        plt.xlabel('Iteration')

        plt.ylabel('Temperature')

        plt.grid(True)

        plt.tight\_layout()

        os.makedirs(self.config.MODEL\_SAVE\_DIR, exist\_ok=True)

        plt.savefig(os.path.join(self.config.MODEL\_SAVE\_DIR, 'training\_progress.png'))

        plt.show()

def main():

    config = SelfPlayConfig()

    trainer = SelfPlayTrainer(config)

    trainer.train()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# dataprocessor.py

import minimax

import rules

import numpy as np

import ast

PATH = 'DATASET.txt'

DATASET\_LENGTH = 100154

def fetch(n):

    with open(PATH, 'r') as file:

            for i, line in enumerate(file, 1):

                if i == n:

                    return line.strip()

            return None

def process(input\_str):

    parts = input\_str.split('-')

    if len(parts) != 2:

        return "wrong format"

    first\_part, second\_part = parts

    replacement = {'O': '0', 'M': '1', 'E': '2'}

    state = [int(replacement.get(char, char)) for char in first\_part]

    coords = []

    letters = second\_part[0::2]

    digits = second\_part[1::2]

    if int(first\_part[24]) != 0:

        x = ord(letters[0]) - ord('a')

        y = 7 - int(digits[0])

        coords = (x,y)

    else:

        x = ord(letters[0]) - ord('a')

        y = 7 - int(digits[0])

        coords.append((x,y))

        x = ord(letters[1]) - ord('a')

        y = 7 - int(digits[1])

        coords.append((x,y))

    return state, coords

def make\_set(n):

    training\_set = []

    i = 1

    while len(training\_set) < n and i < DATASET\_LENGTH:

        state, coords = process(fetch(i))

        if state[26] > 3 and state[27] > 3:

            training\_set.append((state, coords))

        i += 1

    return training\_set

def minimax\_make\_set(depth, n, output\_path):

    training\_set = []

    i = 1

    while len(training\_set) < n and i < DATASET\_LENGTH:

        state, data\_coords = process(fetch(i))

        if state[26] > 3 and state[27] > 3:

            board, move\_count = rules.state\_to\_board(state)

            coords = minimax.minimax(board, depth, 1 if move\_count % 2 == 0 else 2, True, move\_count, -float('inf'), float('inf'))[2]

            if coords not in rules.VALID\_POSITIONS:

                coords = data\_coords

            training\_set.append((state, coords))

        i += 1

    with open(output\_path, 'w', encoding='utf-8') as f:

        for item in training\_set:

            f.write(str(item) + '\n')

def generate\_symmetries(input\_path, output\_path):

    input = []

    with open(input\_path, 'r') as infile:

        for line in infile:

            input.append(ast.literal\_eval(line.strip()))

    with open(output\_path, 'w', encoding='utf-8') as outfile:

        def board\_to\_matrix(board):

            matrix = [[0 for \_ in range(7)] for \_ in range(7)]

            index = 0

            for i in range(7):

                for j in range(7):

                    if (i, j) in rules.VALID\_POSITIONS:

                        matrix[i][j] = board[index]

                        index += 1

            return matrix

        def rotate\_coords(coords):

            x, y = coords

            new\_best\_moves = []

            new\_best\_moves.append((x, y))

            new\_best\_moves.append((y, 6 - x))

            new\_best\_moves.append((6 - x, 6 - y))

            new\_best\_moves.append((6 - y, x))

            mirrored\_moves = []

            for mx, my in new\_best\_moves:

                mirrored\_moves.append((mx, 6 - my))

            return new\_best\_moves, mirrored\_moves

        def rotate\_point(point):

            if type(point) is tuple:

                return rotate\_coords(point)

            else:

                p1, m1 = rotate\_coords(point[0])

                p2, m2 = rotate\_coords(point[1])

            new\_best\_moves = []

            mirrored\_moves = []

            for i in range(4):

                new\_best\_moves.append([p1[i], p2[i]])

                mirrored\_moves.append([m1[i], m2[i]])

            return new\_best\_moves, mirrored\_moves

        def matrix\_to\_board(matrix, board):

            new\_board = []

            for i in range(7):

                for j in range(7):

                    if (i, j) in rules.VALID\_POSITIONS:

                        new\_board.append(int(matrix[i][j]))

            new\_board.extend(board[24:28])

            return new\_board

        for board, best\_move in input:

            board\_matrix = board\_to\_matrix(board)

            new\_best\_moves, mirrored\_moves = rotate\_point(best\_move)

            outfile.write(str((board, new\_best\_moves[0])) + '\n')

            for i in range(3):

                board\_matrix = np.rot90(board\_matrix)

                outfile.write(str((matrix\_to\_board(board\_matrix, board), new\_best\_moves[i+1])) + '\n')

            mirrored\_matrix = np.fliplr(board\_matrix)

            outfile.write(str((matrix\_to\_board(mirrored\_matrix, board), mirrored\_moves[0])) + '\n')

            for i in range(3):

                mirrored\_matrix = np.rot90(mirrored\_matrix)

                outfile.write(str((matrix\_to\_board(mirrored\_matrix, board), mirrored\_moves[i+1])) + '\n')

# minimax.py

import math

import copy

import rules

# try moves that form a mill first to improve pruning

def \_forms\_mill\_fast(board, move, player):

    # Determine the destination

    if isinstance(move, tuple):

        pos = move

    elif isinstance(move, list) and len(move) == 2:

        pos = move[1]

    else:

        return False

    # another fail-safe check

    if board[pos[0]][pos[1]] != 0:

        return False

    # cant use rules.is\_mill here since it needs the move to be fully applied

    for line in rules.mill\_lines:

        if pos in line:

            other = [p for p in line if p != pos]

            if board[other[0][0]][other[0][1]] == player and board[other[1][0]][other[1][1]] == player:

                return True

    return False

def minimax(board, depth, player, is\_maximizing, move\_count, alpha, beta, \_tt=None):

    if \_tt is None:

        \_tt = {}

    # Terminal evaluation

    if depth == 0:

        return rules.evaluate\_board(board, player), [], None

    # Early terminal detection

    if rules.check\_game\_over(board, move\_count):

        c1, c2 = rules.count\_pieces(board)

        if player == 1:

            if c2 < 3:

                return 10\*\*9, [], None  # player wins

            if c1 < 3:

                return -10\*\*9, [], None  # player loses

        else:

            if c1 < 3:

                return 10\*\*9, [], None

            if c2 < 3:

                return -10\*\*9, [], None

        # more unnecessary failsafe

        return rules.evaluate\_board(board, player), [], None

    # Transposition table lookup makes code much faster

    tt\_key = (rules.board\_to\_key(board), depth, is\_maximizing, move\_count, player)

    if tt\_key in \_tt:

        return \_tt[tt\_key]

    opponent = 1 if player == 2 else 2

    best\_score = -math.inf if is\_maximizing else math.inf

    best\_move = None

    best\_move\_chain = []

    # Generate and order moves

    current\_actor = player if is\_maximizing else opponent

    moves = rules.get\_possible\_moves(board, current\_actor, move\_count)

    moves.sort(key=lambda m: \_forms\_mill\_fast(board, m, current\_actor), reverse=True)

    if not moves:

        # more unnecessary failsafe

        eval\_score = rules.evaluate\_board(board, player)

        result = (eval\_score, [], None)

        \_tt[tt\_key] = result

        return result

    for move in moves:

        new\_board = copy.deepcopy(board)

        success, new\_board = rules.apply\_move(new\_board, move, current\_actor, deterministic=True)

        if not success:

            continue

        sub\_score, sub\_chain, \_ = minimax(new\_board, depth - 1, player, not is\_maximizing, move\_count + 1, alpha=alpha, beta=beta, \_tt=\_tt)

        if is\_maximizing:

            if sub\_score > best\_score:

                best\_score = sub\_score

                best\_move = move

                best\_move\_chain = [move] + sub\_chain

            alpha = max(alpha, best\_score)

            if beta <= alpha:

                break

        else:

            if sub\_score < best\_score:

                best\_score = sub\_score

                best\_move = move

                best\_move\_chain = [move] + sub\_chain

            beta = min(beta, best\_score)

            if beta <= alpha:

                break

    result = (best\_score, best\_move\_chain, best\_move)

    \_tt[tt\_key] = result

    return result

# animate\_game.py

import pygame

import ast

def draw\_board\_pygame(board, screen):

    screen.fill((255, 255, 255))

    cell\_size = 40

    offset = 25

    colors = {0: (200, 200, 200), 1: (0, 0, 0), 2: (200, 0, 0), -1: (255, 255, 255)}

    for row in range(len(board)):

        for col in range(len(board[row])):

            x = offset + col \* cell\_size

            y = offset + row \* cell\_size

            color = colors.get(board[row][col], (100, 100, 100))

            pygame.draw.circle(screen, color, (x, y), 15)

    pygame.display.flip()

def animate\_game(board\_history, speed=1.0):

    pygame.init()

    screen\_size = (350, 350)

    screen = pygame.display.set\_mode(screen\_size)

    pygame.display.set\_caption("Nine Men's Morris Animation")

    clock = pygame.time.Clock()

    running = True

    idx = 0

    last\_update = pygame.time.get\_ticks()

    while running:

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                running = False

            elif event.type == pygame.KEYDOWN:

                if event.key == pygame.K\_r:

                    idx = 0  # Restart animation

        now = pygame.time.get\_ticks()

        if now - last\_update > speed \* 1000:

            last\_update = now

            if idx < len(board\_history):

                board\_key = board\_history[idx]

                board = [list(map(int, row)) for row in ast.literal\_eval(board\_key)]

                draw\_board\_pygame(board, screen)

                idx += 1

        clock.tick(60)  # Limit to 60 FPS

# display.py

import tkinter as tk

import matplotlib

import os

def display\_board(array):

    os.environ['SDL\_VIDEO\_CENTERED'] = '1'

    def on\_enter(event=None):

        global user\_input

        user\_input = entry.get()  # assign text

        root.destroy()            # close window

    colors = {

        -1: "white",

         0: "light gray",

         1: "red",

         2: "blue"

    }

    root = tk.Tk()

    root.title("Nine Menn's morris")

    rows = len(array)

    cols = len(array[0])

    entry = tk.Entry(root, font=("Arial", 16))

    canvas = tk.Canvas(root, width=cols\*50, height=rows\*50, bg="white")

    canvas.pack()

    entry.bind("<Return>", on\_enter)

    for i in range(rows):

        for j in range(cols):

            x1, y1 = j\*50, i\*50

            x2, y2 = x1+50, y1+50

            value = array[i][j]

            color = colors.get(value, "gray")

            canvas.create\_rectangle(x1, y1, x2, y2, fill=color, outline="white")

    canvas.pack()

    entry.pack()

    entry.focus\_set()

    root.mainloop()

    return user\_input

def graph\_wins(player1, player2, player1\_wins, player2\_wins, draws):

    matplotlib.use('TkAgg')

    import matplotlib.pyplot as plt

    labels = [player1, player2, 'Draws']

    sizes = [player1\_wins, player2\_wins, draws]

    colors = ['lightcoral', 'lightskyblue', 'lightgreen']

    explode = (0.1, 0.1, 0.1)

    plt.figure(figsize=(8, 6))

    plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)

    plt.title('Game Outcomes')

    plt.axis('equal')

    plt.show()

# EvalEpochs.py

import ast

import NNEvaluation

import NeuralNetData

import torch

import rules

import matplotlib.pyplot as plt

TEST\_PATH = r"D:\Matura\NNnewer\sepember\minimax\_training\_set\_symmetries.txt"

NN\_PATH = r'checkpoints\_batch512'

GAME\_AMOUNT = 1000

MAX\_EPOCH = 2090

EPOCH\_STEP = 100

def eval\_epoch\_test(epoch):

    policy\_net = NeuralNetData.BiggerPolicyNetwork(state\_size=28, action\_size=88)

    policy\_net.load\_state\_dict(torch.load(f'{NN\_PATH}/model\_epoch{epoch}.pth'))

    policy\_net.eval()

    actions = rules.get\_actions()

    correct = 0

    total = 0

    with open(TEST\_PATH, "r") as f:

        start\_line = 500000

        end\_line = 527000

        for i, line in enumerate(f):

            if i < start\_line:

                continue

            if i >= end\_line:

                break

            state, move = ast.literal\_eval(line.strip())

            state\_tensor = torch.FloatTensor(state).unsqueeze(0)

            with torch.no\_grad():

                output = policy\_net(state\_tensor)

                \_, predicted = torch.max(output.data, 1)

                total += 1

                if actions[predicted.item()] == move:

                    correct += 1

    accuracy = correct / total if total > 0 else 0

    print(f'Epoch {epoch}: Accuracy on test set: {accuracy \* 100:.2f}% ({correct}/{total})')

    return accuracy

def eval\_epoch\_play(epoch, opponent):

    policy\_net = NeuralNetData.BiggerPolicyNetwork(state\_size=28, action\_size=88)

    policy\_net.load\_state\_dict(torch.load(f'{NN\_PATH}/model\_epoch{epoch}.pth'))

    policy\_net.eval()

    if opponent == 'random':

        player1wins, player2wins, draws = NNEvaluation.match('2', '4', GAME\_AMOUNT, policy\_net)[2:]

    if opponent == 'minimax':

        player1wins, player2wins, draws = NNEvaluation.match('2', '1', GAME\_AMOUNT, policy\_net, opening\_random\_moves=10)[2:]

    print(f'Epoch {epoch}: NN Wins: {player1wins}, {opponent} Wins: {player2wins}, Draws: {draws}')

    return player1wins, player2wins, draws

def graph\_results\_test(accuracies, epochs, title\_suffix=''):

    plt.figure(figsize=(12, 6))  # Make figure wider to accommodate labels

    plt.plot(epochs, accuracies, marker='o', linewidth=2, markersize=4)

    plt.title(f'NN Test Set Accuracy Over Epochs{title\_suffix}')

    plt.xlabel('Epoch')

    plt.ylabel('Accuracy')

    # Set x-axis ticks to show every few epochs to avoid crowding

    if len(epochs) > 10:

        tick\_step = max(1, len(epochs) // 10)  # Show about 10 ticks

        plt.xticks(epochs[::tick\_step], rotation=45)

    else:

        plt.xticks(epochs)

    plt.ylim(0, 1)

    plt.grid(True, alpha=0.3)

    plt.tight\_layout()  # Ensure labels don't get cut off

    plt.savefig('nn\_test\_accuracy.png', dpi=300, bbox\_inches='tight')

    plt.show()

def graph\_results\_play(player1wins, player2wins, draws, epochs, title\_suffix: str = ''):

    # Choose distinct colors

    colors = {

        "NN Wins": "tab:blue",

        "Opponent Wins": "tab:red",

        "Draws": "tab:green"

    }

    # Make sure inputs are valid

    if not (len(player1wins) == len(player2wins) == len(draws) == len(epochs)):

        raise ValueError("All input lists must be of the same length.")

    # Calculate appropriate bar width based on epoch spacing

    if len(epochs) > 1:

        epoch\_spacing = epochs[1] - epochs[0]

        bar\_width = epoch\_spacing \* 0.8  # Use 80% of the spacing for bar width

    else:

        bar\_width = 80  # Default width if only one epoch

    # Create the bar chart with larger figure size

    plt.figure(figsize=(14, 8))

    # Plot stacked bars with explicit width

    plt.bar(epochs, player1wins, width=bar\_width, color=colors["NN Wins"],

            label="NN Wins", edgecolor='black', linewidth=0.5)

    plt.bar(epochs, player2wins, width=bar\_width, bottom=player1wins,

        color=colors["Opponent Wins"], label="Opponent Wins",

            edgecolor='black', linewidth=0.5)

    plt.bar(epochs, draws, width=bar\_width,

            bottom=[p1 + p2 for p1, p2 in zip(player1wins, player2wins)],

            color=colors["Draws"], label="Draws",

            edgecolor='black', linewidth=0.5)

    # Labels and title

    plt.xlabel("Epochs")

    plt.ylabel("Game Results")

    plt.title(f"Training Results Over Epochs{title\_suffix}")

    plt.legend()

    # Set x-axis ticks to show every few epochs to avoid crowding

    if len(epochs) > 10:

        tick\_step = max(1, len(epochs) // 10)  # Show about 10 ticks

        plt.xticks(epochs[::tick\_step], rotation=45)

    else:

        plt.xticks(epochs)

    plt.grid(True, alpha=0.3)

    plt.tight\_layout()

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    accuracies = []

    p1w\_r = []

    p2w\_r = []

    d\_r = []

    p1w\_m = []

    p2w\_m = []

    d\_m = []

    epochs = list(range(0, MAX\_EPOCH + 1, EPOCH\_STEP))

    for epoch in epochs:

        if epoch == 0:

            accuracies.append(0)

            p1w\_r.append(0)

            p2w\_r.append(0)

            d\_r.append(0)

            p1w\_m.append(0)

            p2w\_m.append(0)

            d\_m.append(0)

            continue

        accuracy = eval\_epoch\_test(epoch)

        accuracies.append(accuracy)

        p1w, p2w, d = eval\_epoch\_play(epoch, 'random')

        p1w\_r.append(p1w)

        p2w\_r.append(p2w)

        d\_r.append(d)

        p1w, p2w, d = eval\_epoch\_play(epoch, 'minimax')

        p1w\_m.append(p1w)

        p2w\_m.append(p2w)

        d\_m.append(d)

    graph\_results\_test(accuracies, epochs)

    graph\_results\_play(p1w\_r, p2w\_r, d\_r, epochs, title\_suffix=' vs Random')

    graph\_results\_play(p1w\_m, p2w\_m, d\_m, epochs, title\_suffix=' vs Minimax')

# NNEvaluation.py

import torch

import rules

import NeuralNetData

import minimax

import display

from animate\_game import animate\_game

import random

import math

def initialize\_model():

    policy\_net = NeuralNetData.BiggerPolicyNetwork(state\_size=28, action\_size=88)

    policy\_net.load\_state\_dict(torch.load('selfplay\_checkpoints\_large/selfplay\_model\_iter\_2600.pth'))

    policy\_net.eval()

    return policy\_net

def initialize\_variables():

    board = [

        [0, -1, -1, 0, -1, -1, 0],

        [-1, 0, -1, 0, -1, 0, -1],

        [-1, -1, 0, 0, 0, -1, -1],

        [0, 0, 0, -1, 0, 0, 0],

        [-1, -1, 0, 0, 0, -1, -1],

        [-1, 0, -1, 0, -1, 0, -1],

        [0, -1, -1, 0, -1, -1, 0],

    ]

    move\_count = 0

    game\_moves = []

    board\_history = []

    depth = 3

    return board, move\_count, depth, board\_history, game\_moves

def match(player1, player2, matches, policy\_net, opening\_random\_moves: int = 0):

    board, move\_count, depth, board\_history, game\_moves = initialize\_variables()

    player1wins = 0

    player2wins = 0

    draws = 0

    board\_history.append(rules.board\_to\_key(board))

    if player1 is None and player2 is None:

        player1 = input("Choose Player 1 (1 for Minimax, 2 for NN, 3 for Human, 4 for random move selection): ")

        player2 = input("Choose Player 2 (1 for Minimax, 2 for NN, 3 for Human, 4 for random move selection): ")

        if player1 not in ['1', '2', '3', '4'] or player2 not in ['1', '2', '3', '4']:

            print("Invalid choice. Please choose 1, 2, 3 or 4 for both players.")

            return

        if player1 and player2 in ['1', '2', '4']:

            matches = int(input("How many matches should be played? "))

        else:

            matches = 1

    for match\_num in range(matches):

        board, move\_count, depth, board\_history, game\_moves = initialize\_variables()

        winner = None

        while True:

            if move\_count % 2 == 0:

                # If within opening randomization window, force a random legal move

                if opening\_random\_moves and move\_count < opening\_random\_moves:

                    board, move = random\_player\_move(board, move\_count, 1)

                else:

                    if player1 == '1':

                        board, move = minimax\_player\_move(board, depth, 1, move\_count)

                    elif player1 == '2':

                        board, move = NN\_player\_move(board, move\_count, 1, policy\_net)

                    elif player1 == '3':

                        move\_raw = display.display\_board(board)

                        board, move = player\_move(board, move\_count, 1, move\_raw)

                    elif player1 == '4':

                        board, move = random\_player\_move(board, move\_count, 1)

                if move is None:

                    winner = "Player 2"

                    break

                game\_moves.append(move)

                board\_history.append(rules.board\_to\_key(board))

                move\_count += 1

                game\_over = rules.check\_game\_over(board, move\_count)

                if game\_over:

                    winner = "Player 1"

                    break

                if board\_history.count(rules.board\_to\_key(board)) >= 3:

                    winner = None

                    break

            else:

                # If within opening randomization window, force a random legal move

                if opening\_random\_moves and move\_count < opening\_random\_moves:

                    board, move = random\_player\_move(board, move\_count, 2)

                else:

                    if player2 == '1':

                        board, move = minimax\_player\_move(board, depth, 2, move\_count)

                    elif player2 == '2':

                        board, move = NN\_player\_move(board, move\_count, 2, policy\_net)

                    elif player2 == '3':

                        move\_raw = display.display\_board(board)

                        board, move = player\_move(board, move\_count, 2, move\_raw)

                    elif player2 == '4':

                        board, move = random\_player\_move(board, move\_count, 2)

                if move is None:

                    winner = "Player 1"

                    break

                game\_moves.append(move)

                board\_history.append(rules.board\_to\_key(board))

                move\_count += 1

                game\_over = rules.check\_game\_over(board, move\_count)

                if game\_over:

                    winner = "Player 2"

                    break

                if board\_history.count(rules.board\_to\_key(board)) >= 3:

                    winner = None

                    break

        if winner:

            if winner == "Player 1":

                player1wins += 1

            else:

                player2wins += 1

        else:

            draws += 1

    if matches > 1:

        return player1, player2, player1wins, player2wins, draws

    else:

        if winner:

            print(f"{winner} wins!")

        else:

            print("It's a draw due to threefold repetition!")

        return game\_moves, board, board\_history

def player\_move(board, move\_count, player, move\_raw):

    move = move\_raw

    move\_parts = move.split()

    if move\_parts[0] == 'place' and len(move\_parts) == 3:

        try:

            row, col = int(move\_parts[1]), int(move\_parts[2])

            action = (row, col)

        except ValueError:

            print("Invalid input. Please enter integers for row and column.")

            display.display\_board(board)

    elif move\_parts[0] == 'move' and len(move\_parts) == 5:

        try:

            from\_row, from\_col, to\_row, to\_col = map(int, move\_parts[1:])

            action = (from\_row, from\_col, to\_row, to\_col)

        except ValueError:

            print("Invalid input. Please enter integers for rows and columns.")

            display.display\_board(board)

    else:

        print("Invalid command. Use 'place' or 'move'.")

        display.display\_board(board)

    possible\_actions = rules.get\_possible\_moves(board, player, move\_count=move\_count)

    if action not in possible\_actions:

        print("Invalid move. Try again.")

        display.display\_board(board)

    \_, board = rules.apply\_move(board, action, player)

    return board, action

def minimax\_player\_move(board, depth, player, move\_count):

    alpha, beta = -math.inf, math.inf

    \_, \_, best\_move = minimax.minimax(board, depth, player, is\_maximizing=True, move\_count=move\_count, alpha=alpha, beta=beta)

    \_, board = rules.apply\_move(board, best\_move, player)

    return board, best\_move

def NN\_player\_move(board, move\_count, player, policy\_net):

    actions = rules.get\_actions()

    player1\_pieces, player2\_pieces = rules.count\_pieces(board)

    selected\_action = None

    board\_positions = [board[row][col] for (row, col) in rules.VALID\_POSITIONS]

    count1 = sum(cell == 1 for row in board for cell in row)

    count2 = sum(cell == 2 for row in board for cell in row)

    state = board\_positions + [player1\_pieces, player2\_pieces, count1, count2]

    state\_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)

    with torch.no\_grad():

        logits = policy\_net(state\_tensor)

        probs = torch.softmax(logits, dim=1)

        sorted\_indices = torch.argsort(probs, dim=1, descending=True)[0]  # Get 1D tensor of indices

        possible\_actions = rules.get\_possible\_moves(board, player, move\_count=move\_count)

        for idx in sorted\_indices:

            top\_action = actions[idx.item()]

            if top\_action in possible\_actions:

                selected\_action = top\_action

                break

    if selected\_action:

        \_, board = rules.apply\_move(board, selected\_action, player)

        return board, selected\_action

    else:

        return board, None

def random\_player\_move(board, move\_count, player):

    possible\_actions = rules.get\_possible\_moves(board, player, move\_count=move\_count)

    if possible\_actions:

        selected\_action = random.choice(possible\_actions)

        \_, board = rules.apply\_move(board, selected\_action, player)

        return board, selected\_action

    else:

        return board, None  # No move possible

if \_\_name\_\_ == "\_\_main\_\_":

    policy\_net = initialize\_model()

    players = ['Minimax', 'NN', 'Human', 'Random']

    result = match(None, None, None, policy\_net)

    if isinstance(result[2], int):  # Multiple matches

        player1, player2, player1wins, player2wins, draws = result

        player1 = players[int(player1)-1]

        player2 = players[int(player2)-1]

        print(f"{player1} wins: {player1wins}, {player2} wins: {player2wins}, Draws: {draws}")

        display.graph\_wins(player1, player2, player1wins, player2wins, draws)

    else:

        game\_moves, board, board\_history = result

        #print("Move chain:", game\_moves)

        #display.display\_board(board)

        animate\_game(board\_history)