```python
from abc import ABC, abstractmethod
from typing import Generic, Set, Tuple, TypeVar

###############################################################################
# An AdversarialSearchProblem is a representation of a game that is convenient
# for running adversarial search algorithms.
#
# A game can be put into this form by extending the AdversarialSearchProblem
# class. See tttproblem.py for an example of this.
#
# Every subclass of AdversarialSearchProblem has its game states represented
# as instances of a subclass of GameState. The only requirement that of a
# subclass of GameState is that it must implement that player_to_move(.) method,
# which returns the index (0-indexed) of the next player to move.
###############################################################################


class GameState(ABC):
    @abstractmethod
    def player_to_move(self) -> int:
        """
        Output- Returns the index of the player who will move next.
        """
        pass


State = TypeVar("State", bound=GameState)

# Action represents the type of actions that an instance of AdversarialSearchProblem uses to
# cause a transition. It's generic because different games have different actions: TTT requires
# placing a piece on a *2D* grid, while Connect 4 just involves selecting a column.
Action = TypeVar("Action")


class AdversarialSearchProblem(ABC, Generic[State, Action]):
    def get_start_state(self):
        """
        Output- Returns the state from which to start.
        """
        return self._start_state

    def set_start_state(self, state: State):
        """
        Changes the start state to the given state.
        Note to student: You should not need to use this.
        This is only for running games.

        Input:
                state- a GameState
        """
        self._start_state = state

    @abstractmethod
    def get_available_actions(self, state: State) → Set[Action]:
        """
        Input:
                state- a GameState
        Output:
                Returns the set of actions available to the player-to-move
                from the given state
        """
        pass

    @abstractmethod
    def transition(self, state: State, action: Action) → State:
        """
        Input:
                state- a Gamestate
                action- the action to take
        Ouput:
                Returns the state that results from taking the given action
                from the given state. (Assume deterministic transitions.)
        """
        assert not (self.is_terminal_state(state))
        assert action in self.get_available_actions(state)
        pass

    @abstractmethod
    def is_terminal_state(self, state: State) → bool:
        """
        Input:
                state: a GameState
        Output:
                Returns a boolean indicating whether or not the given
                state is terminal.
        """
        pass

    # Used to be called evaluate_state
    @abstractmethod
    def evaluate_terminal(self, state: State) → Tuple[int, int]:
        """
        Should be called when determining which player benefits from a given
*terminal* state.
        The range of values returned here should be synchronized with
heuristic_func.

```

```python
        Because we're evaluating terminal states, we're essentially evaluating losing, winning, and
        tieing. You should make sure that the sum of the tuple you return sums to a constant number,
        like 1. If player 0 wins, then should their score be high or low relative to player 1?

        Final note: evaluate_terminal and heuristic_func do very similar things. In fact, their
        ranges are the same! However, we split these up because heuristic_func should be used in
        only the algorithm that uses a heuristic, whereas evaluate_terminal is used across all
        algorithms, since they all need to know how good or bad a terminal state is.

        Input:
                state: a TERMINAL GameState
        Output:
                Returns a Tuple of player 0's value and player 1's value, where each value
                represents whether the player lost, tied, or won.
        """
        assert self.is_terminal_state(state)
        pass


def HeuristicAdversarialSearchProblem(AdversarialSearchProblem):
    @abstractmethod
    def heuristic(self, state: State) -> float:
        """
        Input:
                state: The current game state.
        Output:
                Returns a heuristic evaluation of state (a float)
        """
        pass


###############################################################################
# GameUI is an abstraction that allows you to interact directly with
# an AdversarialSearchProblem (through gamerunner.py). See tttproblem or
# connect4problem for examples.
#
# Utilizing GameUI is NOT necessary for this assignment, although you can use
# it with any ASPs you may decide to create.
###############################################################################

class GameUI(ABC):
    def update_state(self, state: GameState):
        """
        Updates the state currently being rendered.
        """
        self._state = state

    @abstractmethod
    def render(self):
        """
        Renders the GameUI instance's render (presumably this will be called
continuously).
        """
        pass

    @abstractmethod
    def get_user_input_action(self):
        """
        Output- Returns an action obtained through the GameUI input itself.
        (It is expected that GameUI validates that the action is valid).
        """
        pass


from go_search_problem import GoProblem, GoState, Action
from adversarial_search_problem import GameState
from heuristic_go_problems import *
import random
from abc import ABC, abstractmethod
import numpy as np
import time
from game_runner import run_many
import pickle
import torch
from torch import nn
import matplotlib.pyplot as plt
import math

def run_many_wapper(agent1, agent2, num_games):
    # Import `run_many` inside the wrapper function to avoid circular import
    from game_runner import run_many

    agent1_score, agent2_score = run_many(agent1, agent2, num_games)
    print(agent1_score, agent2_score)

    return agent1_score, agent2_score


MAXIMIZER = 0
MIMIZER = 1

class GameAgent():
    # Interface for Game agents
    @abstractmethod
```

```python
    def get_move(self, game_state: GameState, time_limit: float) → Action:
        # Given a state and time limit, return an action
        pass


class RandomAgent(GameAgent):
    # An Agent that makes random moves

    def __init__(self):
        self.search_problem = GoProblem()

    def get_move(self, game_state: GoState, time_limit: float) → Action:
        """
        get random move for a given state
        """
        actions = self.search_problem.get_available_actions(game_state)
        return random.choice(actions)

    def __str__(self):
        return "RandomAgent"


class GreedyAgent(GameAgent):
    def __init__(self, search_problem=GoProblemSimpleHeuristic()):
        super().__init__()
        self.search_problem = search_problem

    def get_move(self, game_state: GoState, time_limit: float) → Action:
        """
        get move of agent for given game state.
        Greedy agent looks one step ahead with the provided heuristic and 
chooses the best available action
        (Greedy agent does not consider remaining time)

        Args:
            game_state (GameState): current game state
            time_limit (float): time limit for agent to return a move
        """
        # Create new GoSearchProblem with provided heuristic
        search_problem = self.search_problem

        # Player 0 is maximizing
        if game_state.player_to_move() == MAXIMIZER:
            best_value = -float('inf')
        else:
            best_value = float('inf')
        best_action = None

        # Get Available actions
        actions = search_problem.get_available_actions(game_state)

        # Compare heuristic of every reachable next state
        for action in actions:
            new_state = search_problem.transition(game_state, action)
            value = search_problem.heuristic(new_state, new_state.player_to_move())
            if game_state.player_to_move() == MAXIMIZER:
                if value > best_value:
                    best_value = value
                    best_action = action
            else:
                if value < best_value:
                    best_value = value
                    best_action = action

        # Return best available action
        return best_action

    def __str__(self):
        """
        Description of agent (Greedy + heuristic/search problem used)
        """
        return "GreedyAgent + " + str(self.search_problem)


class MinimaxAgent(GameAgent):
    def __init__(self, depth=1, search_problem=GoProblemSimpleHeuristic()):
        super().__init__()
        self.depth = depth
        self.search_problem = search_problem

    def get_move(self, game_state: GoState, time_limit: float) -> Action:
        """
        Get move of agent for given game state using minimax algorithm

        Args:
            game_state (GameState): current game state
            time_limit (float): time limit for agent to return a move
        Returns:
            best_action (Action): best action for current game state
        """
        # TODO: implement get_move method of MinimaxAgent
        states_expanded = 0

        def max_value(depth, state):
            """
            Helper function for minimax. Computes the optimal action for the maximizer.
            Input:
                depth - the current depth in the search tree
                state - the current state being evaluated
            Output: tuple containing,
                - the maximum value that can be achieved from this state
                - the corresponding action that leads to this value
            """
            nonlocal states_expanded
            states_expanded += 1

            if self.search_problem.is_terminal_state(state): # check if the current state is terminal
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                else:
                    return 0, None  # Tie

            if depth == self.depth:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            max_eval = float('-inf')
            actions = self.search_problem.get_available_actions(state) # get available actions from the state
            best_action = actions[0]

            for action in actions: # check every available action
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = min_value(depth + 1, next_state)  # calls min_value to predict opponent's action for the next_state

                if curr_eval > max_eval: # update max_eval and best_action if a better value is found
                    max_eval = curr_eval
                    best_action = action

            return max_eval, best_action


        def min_value(depth, state):
            """
            Helper function for minimax. Computes the optimal action for the minimizer.
            Input:
                depth - the current depth in the search tree
                state - the current state being evaluated
            Output: tuple containing,
                - the minimum value that can be achieved from this state
                - the corresponding action that leads to this value
            """
            # Similar logic to the one for max_value in minimax.
            nonlocal states_expanded
            states_expanded += 1

            if self.search_problem.is_terminal_state(state): # check if the current state is terminal
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                else:
                    return 0, None  # Tie

            if depth == self.depth:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            min_eval = float('inf')
            actions = self.search_problem.get_available_actions(state)
            best_action = actions[0]

            for action in actions:
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = max_value(depth + 1, next_state)

                if curr_eval < min_eval:
                    min_eval = curr_eval
                    best_action = action

            return min_eval, best_action

        ############LOGIC FOR MINIMAX
        player = game_state.player_to_move()

        if player == 0: # maximizer plays
            _, best_action = max_value(0, game_state)
        else: # minimizer plays
            _, best_action = min_value(0, game_state)

        # after the first call to max_value/min_value, the helpers call each other recursively until reaching the final_state

        stats = {'states_expanded': states_expanded}

        if best_action is None:  # Check if no valid action was found
            raise ValueError("No valid action found by the agent.")

        return best_action

    def __str__(self):
        return f"MinimaxAgent w/ depth {self.depth} + " + str(self.search_problem)
```

```python


class AlphaBetaAgent(GameAgent):
    def __init__(self, depth=1, search_problem=GoProblemSimpleHeuristic()):
        super().__init__()
        self.depth = depth
        self.search_problem = search_problem

    def get_move(self, game_state: GoState, time_limit: float) -> Action:
        """
        Get move of agent for given game state using alpha-beta algorithm

        Args:
            game_state (GameState): current game state
            time_limit (float): time limit for agent to return a move
        Returns:
            best_action (Action): best action for current game state
        """
        # TODO: implement get_move algorithm of AlphaBeta Agent
        if self.search_problem.is_terminal_state(game_state):
            print("Terminal state reached!")
            return None

        def max_value(depth, state, alpha, beta):
            """
            Helper function for alpha-beta prunning. Computes optimal action for the maximizer.
            Input:
                depth - the current depth in the search tree
                state - the current state being evaluated
                alpha - the best value that the maximizer can guarantee so far
                beta - the best value that the minimizer can guarantee so far

            Output:
                Returns a tuple containing:
                - The maximum value that can be achieved from this state
                - The corresponding action that leads to this value
            """
            if self.search_problem.is_terminal_state(state): # check if the current state is terminal
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                else:
                    return 0, None  # Tie

            if depth == self.depth:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            max_eval = float('-inf')
            actions = self.search_problem.get_available_actions(state)
            best_action = actions[0]

            np.random.shuffle(actions)

            for action in actions:
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = min_value(depth + 1, next_state, alpha, beta)

                if curr_eval > max_eval: # update max_eval, best_action, and alpha if a better value is found
                    max_eval = curr_eval
                    best_action = action

                alpha = max(alpha, max_eval)

                if beta <= alpha: # stop exploring if beta <= alpha
                    return max_eval, best_action

            return max_eval, best_action


        def min_value(depth, state, alpha, beta):
            """
            Helper function for alpha-beta prunning. Computes optimal action for the minimizer.
            Input:
                depth - the current depth in the search tree
                state - the current state being evaluated
                alpha - the best value that the maximizer can guarantee so far
                beta - the best value that the minimizer can guarantee so far

            Output:
                Returns a tuple containing:
                - The minimum value that can be achieved from this state
                - The corresponding action that leads to this value
            """
            if self.search_problem.is_terminal_state(state): # check if the current state is terminal
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                else:
                    return 0, None  # Tie

            if depth == self.depth:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            min_eval = float('inf')
            actions = self.search_problem.get_available_actions(state)
            best_action = actions[0]

            np.random.shuffle(actions)

            for action in actions:
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = max_value(depth + 1, next_state, alpha, beta)

                if curr_eval < min_eval:
                    min_eval = curr_eval
                    best_action = action

                beta = min(beta, min_eval)

                if beta <= alpha:
                    return min_eval, best_action

            return min_eval, best_action

        ############
        alpha = float('-inf')
        beta = float('inf')

        player = game_state.player_to_move()

        if player == 0:
            _, best_action = max_value(0, game_state, alpha, beta)
        else:
            _, best_action = min_value(0, game_state, alpha, beta)

        return best_action

    def __str__(self):
        return f"AlphaBeta w/ depth {self.depth} + " + str(self.search_problem)


class IterativeDeepeningAgent(GameAgent):
    def __init__(self, cutoff_time=1, search_problem=GoProblemSimpleHeuristic()):
        super().__init__()
        self.cutoff_time = cutoff_time
        self.search_problem = search_problem

    def get_move(self, game_state: GoState, time_limit: float):
        """
        Get move of agent for given game state using iterative deepening algorithm (+ alpha-beta).
        Iterative deepening is a search algorithm that repeatedly searches for a solution to a problem,
        increasing the depth of the search with each iteration.

        The advantage of iterative deepening is that you can stop the search based on the time limit, rather than depth.
        The recommended approach is to modify your implementation of Alpha-beta to stop when the time limit is reached
        and run IDS on that modified version.

        Args:
            game_state (GameState): current game state
            time_limit (float): time limit for agent to return a move
        Returns:
            best_action (Action): best action for current game state
        """
        if self.search_problem.is_terminal_state(game_state):
            return self.search_problem.get_available_actions(game_state)[0]

        start_time = time.time()
        time_buffer = 0.05  # Prevent exceeding time limit
        end_time = start_time + time_limit - time_buffer

        best_action = None
        current_depth = 1

        def max_value(depth, state, alpha, beta):
            if time.time() >= end_time:
                raise TimeoutError

            if self.search_problem.is_terminal_state(state):
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                return 0, None

            if depth <= 0:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            max_eval = float('-inf')
            actions = self.search_problem.get_available_actions(state)
            if not actions:
                return max_eval, None

            best_action = actions[0]
            np.random.shuffle(actions)

            for action in actions:
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = min_value(depth - 1, next_state, alpha, beta)

                if curr_eval > max_eval:
                    max_eval = curr_eval
                    best_action = action

                alpha = max(alpha, max_eval)
                if beta <= alpha:
                    break

            return max_eval, best_action

        def min_value(depth, state, alpha, beta):
            if time.time() >= end_time:
                raise TimeoutError

            if self.search_problem.is_terminal_state(state):
                reward = self.search_problem.evaluate_terminal(state)
                if reward == 1:
                    return float('inf'), None
                elif reward == -1:
                    return float('-inf'), None
                return 0, None

            if depth <= 0:
                return self.search_problem.heuristic(state, state.player_to_move()), None

            min_eval = float('inf')
            actions = self.search_problem.get_available_actions(state)
            if not actions:
                return min_eval, None

            best_action = actions[0]
            np.random.shuffle(actions)

            for action in actions:
                next_state = self.search_problem.transition(state, action)
                curr_eval, _ = max_value(depth - 1, next_state, alpha, beta)

                if curr_eval < min_eval:
                    min_eval = curr_eval
                    best_action = action

                beta = min(beta, min_eval)
                if beta <= alpha:
                    break

```

```python
            return min_eval, best_action

        actions = self.search_problem.get_available_actions(game_state)
        if actions:
            best_action = actions[0]
        else:
            return None

        # Main IDS loop
        while time.time() < end_time:
            try:
                alpha = float('-inf')
                beta = float('inf')

                if game_state.player_to_move() == 0:  # MAX player
                    _, current_action = max_value(current_depth, game_state, alpha, beta)
                else:  # MIN player
                    _, current_action = min_value(current_depth, game_state, alpha, beta)

                if current_action is not None:
                    best_action = current_action

                current_depth += 1

            except TimeoutError:
                break

        return best_action

    def __str__(self):
        return f"IterativeDeepening + " + str(self.search_problem)



def load_dataset(path: str):
    with open(path, 'rb') as f:
        dataset = pickle.load(f)
    return dataset

dataset_5x5 = load_dataset('dataset_5x5.pkl')
# dataset_9x9 = load_dataset('9x9_dataset.pkl')

def save_model(path: str, model, input_size=None):
    """
    Save model to a file
    Input:
        path: path to save model to
        model: Pytorch model to save
    """

    torch.save({
        'model_state_dict': model.state_dict(),
    }, path)

def load_model(path: str, model):
    """
    Load model from file

    Note: you still need to provide a model (with the same architecture as the saved model))

    Input:
        path: path to load model from
        model: Pytorch model to load
    Output:
        model: Pytorch model loaded from file
    """
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint['model_state_dict'])
    return model

class ValueNetwork(nn.Module):
    def __init__(self, input_size):
        super(ValueNetwork, self).__init__()

        # TODO: What should the output size of a Value function be?

        ''' Handout: the goal is to classify each state as a future
        win for one player or the other, or more generally, to
        generate a prediction in the range [-1, +1] that is indicative
        of which player will win the game.'''

        output_size = 1

        # TODO: Add more layers, non-linear functions, etc.

        # Layers
        self.fc1 = nn.Linear(input_size, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, output_size)

        # Activation functions
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        """
        Run forward pass of network

        Input:
```

```
        x: input to network
        Output:
        output of network
        """
        # TODO: Update as more layers are added
        z1 = self.fc1(x)
        a1 = self.relu(z1)

        z2 = self.fc2(a1)
        a2 = self.relu(z2)

        z3 = self.fc3(a2)
        a3 = self.relu(z3)

        return a3


class GoProblemLearnedHeuristic(GoProblem):
    def __init__(self, model=None, state=None):
        super().__init__(state=state)
        self.model = model

    def __call__(self, model=None):
        """
        Use the model to compute a heuristic value for a given state.
        """
        return self

    def encoding(self, state):
        """
        Get encoding of state (convert state to features)
        Note, this may call get_features() from Task 1.

        Input:
            state: GoState to encode into a fixed size list of features
        Output:
            features: list of features
        """
        # TODO: get encoding of state (convert state to features)

        return get_features(state)

    def heuristic(self, state, player_index):
        """
        Return heuristic (value) of current state

        Input:
            state: GoState to encode into a fixed size list of features
            player_index: index of player to evaluate heuristic for
        Output:
            value: heuristic (value) of current state
```

```python
        """
        # TODO: Compute heuristic (value) of current state
        value = 0

        features = self.encoding(state)
        features_tensor = torch.tensor(features, dtype=torch.float32)

        with torch.no_grad():
            value = self.model(features_tensor)

        '''value = max(-1, min(1, value))
        if player_index ≠ state.player_to_move():
            value = -value'''

        # Note, your agent may perform better if you force it not to pass
        # (i.e., don't select action #25 on a 5×5 board unless necessary)
        return value

    def __str__(self) → str:
        return "Learned Heuristic"

    import go_utils
def create_value_agent_from_model():
    """
    Create agent object from saved model. This (or other methods like this) will be how your agents will be created in gradescope and in the final tournament.
    """

    model_path = "value_model.pt"
    # TODO: Update number of features for your own encoding size

    feature_size = len(get_features(dataset_5x5[0][0]))

    model = load_model(model_path, ValueNetwork(feature_size))

    heuristic_search_problem = GoProblemLearnedHeuristic(model)

    # TODO: Try with other heuristic agents (IDS/AB/Minimax)
    learned_agent = GreedyAgent(heuristic_search_problem)

    return learned_agent


def get_features(game_state: GoState):
    """
    Map a game state to a list of features.

    Some useful functions from game_state include:
        game_state.size: size of the board
        get_pieces_coordinates(player_index): get coordinates of all pieces of a player (0 or 1)
        get_pieces_array(player_index): get a 2D array of pieces of a player (0 or 1)

        get_board(): get a 2D array of the board with 4 channels (player 0, player 1, empty, and player to move). 4 channels means the array will be of size 4 x n x n

        Descriptions of these methods can be found in the GoState

    Input:
        game_state: GoState to encode into a fixed size list of features
    Output:
        features: list of features
    """
    board_size = game_state.size

    # TODO: Encode game_state into a list of features
    features = []

    board = game_state.get_board()

    for channel in range(4):
        for row in range(board_size):
            for col in range(board_size):
                features.append(board[channel][row][col])

    return features


class PolicyNetwork(nn.Module):
    def __init__(self, input_size, board_size=5):
      super(PolicyNetwork, self).__init__()

      # TODO: What should the output size of the Policy be?
      self.output_size = board_size * board_size + 1

      # TODO: Add more layers, non-linear functions, etc.
      self.fc1 = nn.Linear(input_size, 512)
      self.fc2 = nn.Linear(512, 128)
      self.fc3 = nn.Linear(128, 64)
      self.fc4 = nn.Linear(64, self.output_size)

      self.relu = nn.ReLU()

    def forward(self, x):
      # TODO: Update as more layers are added
      z1 = self.fc1(x)
      a1 = self.relu(z1)
      z2 = self.fc2(a1)
```

```
   a2 = self.relu(z2)
   z3 = self.fc3(a2)
   a3 = self.relu(z3)
   z4 = self.fc4(a3)

   return z4

class PolicyAgent(GameAgent):
  def __init__(self, search_problem, model_path, board_size=5):
    super().__init__()
    self.search_problem = search_problem
    # self.model = load_model(model_path, PolicyNetwork)

    input_size = len(get_features(dataset_5x5[0][0]))
    model_template = PolicyNetwork(input_size, board_size)
    self.model = load_model(model_path, model_template)

    self.board_size = board_size

  def encoding(self, state):
    # TODO: get encoding of state (convert state to features)
    return get_features(state)

  def get_move(self, game_state: GoState, time_limit=1):
   """
   Get best action for current state using self.model

   Input:
    game_state: current state of the game
    time_limit: time limit for search (This won't be used in this agent)
   Output:
    action: best action to take
   """
   legal_actions = self.search_problem.get_available_actions(game_state)

   features = self.encoding(game_state)
   features_tensor = torch.tensor(features,
dtype=torch.float32).unsqueeze(0)

   with torch.no_grad():
     action_logits = self.model(features_tensor)
     action_probs = torch.softmax(action_logits, dim=1).squeeze(0)

   all_probs = action_probs.tolist()

   # Get probabilities for legal actions
   legal_actions_probs = [
     (action, all_probs[action-1] if 1 ⩽ action ⩽ len(all_probs) else
0)
     for action in legal_actions
   ]
```

```python

        # Sort legal actions by probability
        sorted_legal_actions = sorted(legal_actions_probs, key=lambda x: x[1], reverse=True)

        # Return best legal action
        return sorted_legal_actions[0][0] if sorted_legal_actions else None

    def __str__(self) -> str:
        return "Policy Agent"

def create_policy_agent_from_model():
    """
    Create agent object from saved model. This (or other methods like this) will be how your agents will be created in gradescope and in the final tournament.
    """

    model_path = "policy_model.pt"
    agent = PolicyAgent(GoProblem(size=5), model_path)
    return agent



def plot_agent_comparisons(learned_agent_name, learned_agent):
    """
    Create a bar plot comparing the performance of learned agents against other agents

    :param value_agent: Learned value network agent
    :param policy_agent: Learned policy network agent
    """
    random_agent = RandomAgent()
    greedy_agent = GreedyAgent()
    minimax_agent = MinimaxAgent()
    alpha_beta_agent = AlphaBetaAgent()
    iterative_deepening_agent = IterativeDeepeningAgent()
    mcts_agent = MCTSAgent()

    agents = [
        ("Random", random_agent),
        ("Greedy", greedy_agent),
        ("Minimax", minimax_agent),
        ("AlphaBeta", alpha_beta_agent),
        ("IterativeDeepening", iterative_deepening_agent),
        ("MCTS", mcts_agent)
    ]

    num_games = 5

    value_scores = []
    policy_scores = []
    agent_names = []

    for agent_name, agent in agents:
        agent1_score, agent2_score = run_many_wapper(agent, learned_agent, num_games)
        agent_names.append(f"{learned_agent_name} vs {agent_name}")
        value_scores.append(agent1_score)
        policy_scores.append(agent2_score)

    plt.figure(figsize=(12, 6))

    bar_width = 0.35

    r1 = np.arange(len(agents))
    r2 = [x + bar_width for x in r1]

    plt.bar(r1, value_scores, color='skyblue', width=bar_width, label=learned_agent_name)
    plt.bar(r2, policy_scores, color='lightgreen', width=bar_width, label=f'Opponent {learned_agent_name}')

    plt.xlabel('Opponent Agents')
    plt.ylabel('Score')
    plt.title(f'Performance of {learned_agent_name} Against Different Opponents')
    plt.xticks([r + bar_width/2 for r in range(len(agents))], agent_names, rotation=45)

    plt.legend()

    for i, (v1, v2) in enumerate(zip(value_scores, policy_scores)):
        plt.text(r1[i], v1, f'{v1:.2f}', ha='center', va='bottom')
        plt.text(r2[i], v2, f'{v2:.2f}', ha='center', va='bottom')

    plt.tight_layout()
    plt.savefig(f'agent_comparison_{learned_agent_name}.png')
    plt.close()


class MCTSNode:
    def __init__(self, state, parent=None, action=None, prior_probability=0.0):
        self.state = state
        self.parent = parent
        self.children = []
        self.visits = 0
        self.value = 0
        self.action = action
        self.prior_probability = prior_probability

```

```python

    def is_leaf(self):
        """
        Checks if the node is a leaf (i.e., has no children).
        """
        return (len(self.children) == 0 or
                self.state.is_terminal_state() or
                len(self.children) < len(self.state.legal_actions()))

    def __hash__(self):
        return hash(self.state)


class MCTSAgent(GameAgent):
    def __init__(self, c=np.sqrt(2)):
        super().__init__()
        self.c = c
        self.search_problem = GoProblem()
        self.action_choices = []

    def get_move(self, game_state: GoState, time_limit: float) -> Action:
        root = MCTSNode(game_state)
        start_time = time.time()

        # While time remains
        while time.time() - start_time < 0.9:
            leaf = self.select(root)

            # Only expand if not terminal
            if not self.search_problem.is_terminal_state(leaf.state):
                self.expand(leaf)

                # Simulate and backpropagate for each child
                for child in leaf.children:
                    result = self.simulate(child)
                    self.backprop(result, child)

        # Return action with most visits
        return max(root.children, key=lambda child: child.visits).action

    def select(self, node):
        """SELECT: Find a leaf node using UCT policy."""
        while not node.is_leaf():
            node = max(node.children, key=lambda child: self.uct_value(child))
        return node

    def expand(self, leaf):
        """EXPAND: Create all possible child nodes."""
        actions = leaf.state.legal_actions()

        # Create a child node for each legal action
        for action in actions:
            child_state = self.search_problem.transition(leaf.state, action)
            child_node = MCTSNode(state=child_state, parent=leaf, action=action)
            leaf.children.append(child_node)

    def simulate(self, node):
        """SIMULATE: Run rollout from given node."""
        curr_state = node.state

        while not self.search_problem.is_terminal_state(curr_state):
            actions = curr_state.legal_actions()
            action = np.random.choice(actions)
            curr_state = self.search_problem.transition(curr_state, action)
            self.action_choices.append(action)

        return self.search_problem.evaluate_terminal(curr_state)

    def backprop(self, result, node):
        """BACKPROPAGATE: Update statistics from leaf to root."""
        while node is not None:
            node.visits += 1

            if result < 0 and node.state.player_to_move() == 0:
                node.value += 1
            elif result > 0 and node.state.player_to_move() == 1:
                node.value += 1

            node = node.parent

    def uct_value(self, node):
        """Calculate UCT value for node selection."""
        if node.visits == 0:
            return float('inf')

        exploitation = node.value / node.visits
        exploration = (self.c * np.sqrt(np.log(node.parent.visits) / node.visits) if node.parent else 0)

        return exploitation + exploration

    def plot_action_frequencies(self, actions):
        action_counts = {action: actions.count(action) for action in set(actions)}
        actions = list(action_counts.keys())
        frequencies = list(action_counts.values())

        plt.figure(figsize=(8, 6))
        plt.bar(actions, frequencies)
        plt.xlabel("Action")
        plt.ylabel("Frequency")
        plt.title("Action Frequencies in Rollouts")
        plt.savefig('action_frequencies_plot.png')
        plt.close()

    def __str__(self):
        return "MCTS"


class NeuralMCTSAgent(GameAgent):
    def __init__(self, policy_network, value_network, c=np.sqrt(2)):
        super().__init__()
        self.c = c
        self.policy_network = policy_network
        self.value_network = value_network
        self.search_problem = GoProblem()
        self.action_choices = []

    def get_move(self, game_state: GoState, time_limit: float) → Action:
        root = MCTSNode(game_state)
        start_time = time.time()

        # While time remains
        while time.time() - start_time < 0.9:
            leaf = self.select(root)

            # Only expand if not terminal
            if not self.search_problem.is_terminal_state(leaf.state):
                self.expand(leaf)

                # Simulate and backpropagate for each child
                for child in leaf.children:
                    result = self.simulate(child)
                    self.backprop(result, child)

        # Return action with most visits
        return max(root.children, key=lambda child: child.visits).action

    def select(self, node):
        """SELECT: Find a leaf node using PUCT policy."""
        while not node.is_leaf():
            node = max(node.children, key=lambda child:
self.puct_value(child))
        return node

    def expand(self, leaf):
        """EXPAND: Create all possible child nodes, guided by the policy
network."""
        actions = leaf.state.legal_actions()

        # Get the policy distribution from the policy network for the current
state
```

```python
        features = get_features(leaf.state)  # Get features from the state
        features_tensor = torch.tensor(features, dtype=torch.float32)

        with torch.no_grad():
            # Get policy logits and convert them to probabilities
            policy_logits = self.policy_network(features_tensor)
            policy_probs = torch.softmax(policy_logits, dim=0).tolist()

        # Filter out illegal actions
        legal_actions_probs = {action: policy_probs[action - 1] for action in actions}

        # Sort actions based on their probability from the policy network
        sorted_actions = sorted(legal_actions_probs.items(), key=lambda x: x[1], reverse=True)

        # Create a child node for each legal action, prioritizing higher probability actions
        for action, _ in sorted_actions:
            child_state = self.search_problem.transition(leaf.state, action)
            prior_probability = policy_probs[action - 1]
            child_node = MCTSNode(state=child_state, parent=leaf, action=action, prior_probability=prior_probability)
            leaf.children.append(child_node)

    def simulate(self, node):
        """SIMULATE: Use the value network to simulate the outcome from the given node."""
        features = get_features(node.state)
        features_tensor = torch.tensor(features, dtype=torch.float32)

        # Use the value network to evaluate the state
        with torch.no_grad():
            value = self.value_network(features_tensor).item()

        return value

    def backprop(self, result, node):
        """BACKPROPAGATE: Update statistics from leaf to root."""
        while node is not None:
            node.visits += 1

            if result < 0 and node.state.player_to_move() == 0:
                node.value += 1
            elif result > 0 and node.state.player_to_move() == 1:
                node.value += 1

            node = node.parent

    def uct_value(self, node):
        """Calculate UCT value for node selection."""
        if node.visits == 0:
            return float('inf')

        exploitation = node.value / node.visits
        exploration = (self.c * np.sqrt(np.log(node.parent.visits) / node.visits) if node.parent else 0)

        return exploitation + exploration


    def puct_value(self, node):
        """Calculate PUCT value for node selection."""
        if node.visits == 0:
            return float('inf')

        exploitation = node.value / node.visits

        '''
        UCT implementation:
            exploration = (self.c * np.sqrt(np.log(node.parent.visits) / node.visits) if node.parent else 0)
        '''
        exploration = self.c * node.prior_probability * np.sqrt(np.log(node.parent.visits) / node.visits) if node.parent else 0

        return exploitation + exploration

    def __str__(self):
        return "Neural MCTS"



class OpeningBook:
    def __init__(self):
        self.openings_5x5 = {
            'empty_board': [
                (2, 2),    # Center
                (1, 1),
                (1, 3),
                (3, 1),
                (3, 3)
            ],
            'center_taken': [
                (0, 0),    # Corner
                (0, 4),
                (4, 0),
                (4, 4),
                (1, 0),    # Edge
                (1, 4),
                (3, 0),
                (3, 4)
            ]
        }

        self.openings_9x9 = {
            'empty_board': [
                (4, 4),    # Center
                (2, 2),    # Star points (4-4 points)
                (2, 6),
                (6, 2),
                (6, 6),
                (2, 4),    # Side star points
                (4, 2),
                (4, 6),
                (6, 4)
            ],
            'center_taken': [
                (1, 1),    # 3-3 points
                (1, 7),
                (7, 1),
                (7, 7),
                (4, 1),    # Side approaches
                (1, 4),
                (4, 7),
                (7, 4)
            ],
            'star_point_taken': [
                (3, 3),    # 5-5 points
                (3, 5),
                (5, 3),
                (5, 5),
                (0, 0),    # Corner moves
                (0, 8),
                (8, 0),
                (8, 8)
            ]
        }

    def get_opening_move(self, game_state: GoState, time_limit: float):
        """
        Get an opening move based on predefined strategies for specific board
states.
        """
        board = game_state.get_board()
        size = board.shape[1]

        # Select appropriate opening book based on board size
        openings = self.openings_5x5 if size == 5 else self.openings_9x9

        # Check if the board is empty
        if self.is_empty_board(board):
            for move in openings['empty_board']:
                if self.is_legal_move(board, move):
                    return self.move_to_index(move, size)

        # For 9x9 board, check if any star points are taken
        if size == 9 and not self.is_empty_board(board):
            star_points = [(2, 2), (2, 6), (6, 2), (6, 6)]
            if any(not self.is_empty_point(board, point) for point in star_points):
                for move in openings['star_point_taken']:
                    if self.is_legal_move(board, move):
                        return self.move_to_index(move, size)

        # If the center is taken, play from 'center_taken' strategies
        if not self.is_empty_board(board):
            for move in openings['center_taken']:
                if self.is_legal_move(board, move):
                    return self.move_to_index(move, size)

        # No suitable opening move found
        return None

    def is_empty_board(self, board: np.ndarray) -> bool:
        """
        Check if the board is empty by verifying the first three channels.
        An empty board has no black or white pieces, and all cells in the EMPTY channel are 1.
        """
        return np.all(board[0] == 0) and np.all(board[1] == 0) and np.all(board[2] == 1)

    def is_legal_move(self, board: np.ndarray, move: tuple) -> bool:
        """
        Check if the move is legal (i.e., within bounds and on an empty cell).
        """
        x, y = move
        size = board.shape[1]

        # Move within bounds and on an empty cell
        empty_board = (board[2])
        return 0 <= x < size and 0 <= y < size and empty_board[x][y] == 1

    def move_to_index(self, move: tuple, size: int) -> int:
        """Convert a 2D (row, col) move to a 1D index."""
        x, y = move
        return x * size + y


class HybridGoAgent5x5(GameAgent):
    def __init__(self, board_size=5):
        super().__init__()

        self.opening_book = OpeningBook()

        # input_size = len(get_features(dataset_5x5[0][0]))

        # policy_model = PolicyNetwork(input_size)
        # value_model = ValueNetwork(input_size)
        # self.mcts_agent = NeuralMCTSAgent(policy_network=policy_model, value_network=value_model)
        self.mcts_agent = MCTSAgent()

        self.alphabeta_agent = AlphaBetaAgent()
        self.ids_agent = IterativeDeepeningAgent(1, GoProblemAdvancedHeuristic())
        self.move_count = 0
        self.total_moves = board_size * board_size

    def get_move(self, state: GoState, time_limit: float) -> Action:
        self.move_count += 1
        if self.move_count <= 3:
            book_move = self.opening_book.get_opening_move(state, time_limit)
            if book_move is not None:
                return book_move

        endgame_threshold = int(self.total_moves * 0.75)
        if self.move_count >= endgame_threshold:
            # return self.alphabeta_agent.get_move(state, time_limit)
            return self.ids_agent.get_move(state, time_limit)

        return self.mcts_agent.get_move(state, time_limit)

    def __str__(self):
        return "HybridGoAgent5x5"



class HybridGoAgent9x9(GameAgent):
    def __init__(self, board_size=9):
        super().__init__()

        self.opening_book = OpeningBook()

        self.mcts_agent = MCTSAgent()

        self.alphabeta_agent = AlphaBetaAgent()
        self.ids_agent = IterativeDeepeningAgent(1, GoProblemAdvancedHeuristic())
        self.move_count = 0
        self.total_moves = board_size * board_size

    def get_move(self, state: GoState, time_limit: float) -> Action:
        self.move_count += 1
```

```python
        if self.move_count ⩽ 4:
            book_move = self.opening_book.get_opening_move(state, time_limit)
            if book_move is not None:
                return book_move

        endgame_threshold = int(self.total_moves * 0.7)
        if self.move_count ⩾ endgame_threshold:
            return self.ids_agent.get_move(state, time_limit)

        return self.mcts_agent.get_move(state, time_limit)

    def __str__(self):
        return "HybridGoAgent9×9"


def get_final_agent_5×5():
    """Called to construct agent for final submission for 5×5 board"""
    return HybridGoAgent5×5()

def get_final_agent_9×9():
    """Called to construct agent for final submission for 9×9 board"""
    return HybridGoAgent9×9()


def plot_compare_hybrid_agent(hybrid_agent : HybridGoAgent5×5):
    """
    Create a bar plot comparing the performance of HybridGoAgent5×5 against other agents
    """
    hybrid_agent_name = str(hybrid_agent)

    agents = [
        ("IterativeDeepening", IterativeDeepeningAgent(GoProblemSimpleHeuristic)),
        ("MCTS", MCTSAgent()),
        ("Random", RandomAgent()),
        ("Greedy", GreedyAgent()),
        ("Minimax", MinimaxAgent()),
        ("AlphaBeta", AlphaBetaAgent())
    ]

    num_games = 5
    hybrid_scores = []
    opponent_scores = []
    agent_names = []

    for agent_name, opponent in agents:
        # Run games with hybrid agent as both first and second player
        hybrid_first, opp_first = run_many(hybrid_agent, opponent, num_games)
        opp_second, hybrid_second = run_many(opponent, hybrid_agent, num_games)

        # Average scores from both positions
        hybrid_avg = (hybrid_first + hybrid_second) / 2
        opp_avg = (opp_first + opp_second) / 2

        print(f"{hybrid_agent_name}: {hybrid_avg}, {agent_name} Score: {opp_avg}")

        agent_names.append(f"vs {agent_name}")
        hybrid_scores.append(hybrid_avg)
        opponent_scores.append(opp_avg)

    plt.figure(figsize=(12, 6))
    bar_width = 0.35

    r1 = np.arange(len(agents))
    r2 = [x + bar_width for x in r1]

    plt.bar(r1, hybrid_scores, color='skyblue', width=bar_width, label=hybrid_agent_name)
    plt.bar(r2, opponent_scores, color='lightgreen', width=bar_width, label='Opponent')

    plt.xlabel('Opponent Agents')
    plt.ylabel('Average Score')
    plt.title(f'Performance of {hybrid_agent_name} Against Different Opponents')
    plt.xticks([r + bar_width/2 for r in range(len(agents))], agent_names, rotation=45)

    plt.legend()

    for i, (h_score, o_score) in enumerate(zip(hybrid_scores, opponent_scores)):
        plt.text(r1[i], h_score, f'{h_score:.2f}', ha='center', va='bottom')
        plt.text(r2[i], o_score, f'{o_score:.2f}', ha='center', va='bottom')

    plt.tight_layout()
    plt.savefig(f'{hybrid_agent_name}_comparison.png')
    plt.close()


def main():
    agent5×5 = HybridGoAgent5×5()
    agent9×9 = HybridGoAgent9×9()

    plot_compare_hybrid_agent(agent5×5)
    # plot_compare_hybrid_agent(agent9×9)

    '''
    go_agent5×5 = HybridGoAgent5×5()
```

```python
    go_agent9x9 = HybridGoAgent9x9()

    random_agent = RandomAgent()
    greedy_agent = GreedyAgent()
    minimax_agent = MinimaxAgent()
    alpha_beta_agent = AlphaBetaAgent()
    iterative_deepening_agent = IterativeDeepeningAgent()
    mcts_agent = MCTSAgent()
    policy_agent = create_policy_agent_from_model()
    value_agent = create_value_agent_from_model()

    agents = [
        #("Random", random_agent),
        ("Greedy", greedy_agent),
        #("Minimax", minimax_agent),
        #("AlphaBeta", alpha_beta_agent),
        #("IterativeDeepening", iterative_deepening_agent),
        #("MCTS", mcts_agent),
        #("Policy", policy_agent),
        #("Value", value_agent)
    ]

    num_games = 5

    for agent_name, agent in agents:
        go_agent_score9x9, simple_agent_score = run_many(go_agent9x9, agent, num_games)
        print(f"{str(go_agent9x9)}: {go_agent_score9x9}, {agent_name} Score: {simple_agent_score}")

        go_agent_score5x5, simple_agent_score = run_many(go_agent5x5, agent, num_games)
        print(f"{str(go_agent5x5)}: {go_agent_score5x5}, {agent_name} Score: {simple_agent_score}")
    '''


if __name__ == "__main__":
    main()


import time
from go_search_problem import GoProblem
import abc
import tqdm
import numpy as np
from go_gui import GoGUI
# from agents import *
import pygame
import argparse

```

```python
pygame.init()
clock = pygame.time.Clock()

BLACK = MAXIMIZER = 0
WHITE = MINIMIZER = 1


def run_game(agent1, agent2, time_limit=15, time_increment=1, hard_time_cutoff=True, size=5):
    """
    Run a single game between two agents.
    :param agent1: The first agent
    :param agent2: The second agent
    :param time_limit: The time limit for each player (starting time)
    :param time_increment: The time increment for each player (additional time per move)
    :param hard_time_cutoff: If true, will terminate the game when a player runs out of time
                                If false, will continue to play until the game is over.
    :return: The result of the game (1 for agent1 win, -1 for agent2 win)
    """
    my_go = GoProblem(size=size)
    state = my_go.start_state
    player1_time = time_limit
    player2_time = time_limit
    player1_durations = []
    player2_durations = []
    while (not my_go.is_terminal_state(state)):
        start_time = time.time()
        # Clone so as to avoid side effects from agents
        player1_action = agent1.get_move(state.clone(), player1_time)
        move_duration = time.time() - start_time
        player1_time -= move_duration
        player1_durations.append(move_duration)
        if (player1_time <= 0):
            print("Player 1 over time")
            if hard_time_cutoff:
                info = {"Agent 1 End Time": player1_time, "Agent 2 End Time": player2_time,
                        "Agent 1 Average Duration": np.mean(player1_durations),
                        "Agent 2 Average Duration": np.mean(player2_durations),
                        "Agent 1 Longest Duration": np.max(player1_durations),
                        "Agent 2 Longest Duration": np.max(player2_durations),
                        "Agent 1 Score": -1, "Agent 2 Score": 1}
                return -1, info
        player1_time += time_increment
        state = my_go.transition(state, player1_action)
        if (my_go.is_terminal_state(state)):
            break
        start_time = time.time()
        player2_action = agent2.get_move(state.clone(), player2_time)
        duration = time.time() - start_time
        player2_durations.append(duration)
        player2_time -= duration
        if (player2_time <= 0):
            print("Player 2 over time")
            if hard_time_cutoff:
                info = {"Agent 1 End Time": player1_time, "Agent 2 End Time": player2_time,
                        "Agent 1 Average Duration": np.mean(player1_durations),
                        "Agent 2 Average Duration": np.mean(player2_durations),
                        "Agent 1 Longest Duration": np.max(player1_durations),
                        "Agent 2 Longest Duration": np.max(player2_durations),
                        "Agent 1 Score": -1, "Agent 2 Score": 1}
                return 1, info
        else:
            player2_time += time_increment
        state = my_go.transition(state, player2_action)
    info = {"Agent 1 End Time": player1_time, "Agent 2 End Time": player2_time,
            "Agent 1 Average Duration": np.mean(player1_durations),
            "Agent 2 Average Duration": np.mean(player2_durations),
            "Agent 1 Longest Duration": np.max(player1_durations),
            "Agent 2 Longest Duration": np.max(player2_durations),
            "Agent 1 Score": -1, "Agent 2 Score": 1}
    return my_go.evaluate_terminal(state), info


def run_many(agent1, agent2, num_games=10, verbose=True, size=5):
    print(f"Number of games: {num_games}")
    agent1_score = 0
    agent2_score = 0
    agent1_score_black = 0
    agent2_score_black = 0
    agent1_average_duration = 0
    agent2_average_duration = 0

    agent1_longest_duration = 0
    agent2_longest_duration = 0

    agent1_average_time_remaining = 0
    agent2_average_time_remaining = 0

    agent1_min_time_remaining = float('inf')
    agent2_min_time_remaining = float('inf')

    for _ in tqdm.tqdm(range(int(num_games / 2))):
        result, info = run_game(agent1, agent2)
        agent1_score += result
        agent2_score += -result
        agent1_score_black += result

        agent1_average_duration += info["Agent 1 Average Duration"] /
num_games
        agent2_average_duration += info["Agent 2 Average Duration"] /
num_games

        agent1_longest_duration = max(
            agent1_longest_duration, info["Agent 1 Longest Duration"])
        agent2_longest_duration = max(
            agent2_longest_duration, info["Agent 2 Longest Duration"])

        agent1_average_time_remaining += info["Agent 1 End Time"] / num_games
        agent2_average_time_remaining += info["Agent 2 End Time"] / num_games

        agent1_min_time_remaining = min(
            agent1_min_time_remaining, info["Agent 1 End Time"])
        agent2_min_time_remaining = min(
            agent2_min_time_remaining, info["Agent 2 End Time"])

        result, info = run_game(agent2, agent1)

        # Note that since player 2 goes first in the second game,
        # The stats will look backwards
        agent2_score_black += result
        agent1_score += -result
        agent2_score += result

        agent1_average_duration += info["Agent 2 Average Duration"] /
num_games
        agent2_average_duration += info["Agent 1 Average Duration"] /
num_games

        agent1_longest_duration = max(
            agent1_longest_duration, info["Agent 2 Longest Duration"])
        agent2_longest_duration = max(
            agent2_longest_duration, info["Agent 1 Longest Duration"])

        agent1_average_time_remaining += info["Agent 2 End Time"] / num_games
        agent2_average_time_remaining += info["Agent 1 End Time"] / num_games

        agent1_min_time_remaining = min(
            agent1_min_time_remaining, info["Agent 2 End Time"])
        agent2_min_time_remaining = min(
            agent2_min_time_remaining, info["Agent 1 End Time"])

    if verbose:
        print("Agent 1: " + str(agent1) + " Score: " + str(agent1_score))
```

```python
        print("Agent 2: " + str(agent2) + " Score: " + str(agent2_score))
        print("Agent 1: " + str(agent1) + " Score with Black (first move): " +
              str(agent1_score_black))
        print("Agent 2: " + str(agent2) + " Score with Black (first move): " +
              str(agent2_score_black))
        print("Agent 1: " + str(agent1) + " Average Duration: " +
              str(agent1_average_duration))
        print("Agent 2: " + str(agent2) + " Average Duration: " +
              str(agent2_average_duration))
        print("Agent 1: " + str(agent1) + " Longest Duration: " +
              str(agent1_longest_duration))
        print("Agent 2: " + str(agent2) + " Longest Duration: " +
              str(agent2_longest_duration))
        print("Agent 1: " + str(agent1) + " Average Time Remaining: " +
              str(agent1_average_time_remaining))
        print("Agent 2: " + str(agent2) + " Average Time Remaining: " +
              str(agent2_average_time_remaining))
        print("Agent 1: " + str(agent1) + " Min Time Remaining: " +
              str(agent1_min_time_remaining))
        print("Agent 2: " + str(agent2) + " Min Time Remaining: " +
              str(agent2_min_time_remaining))

    return agent1_score, agent2_score


def run_game_with_gui(agent, size=5):
    """
    Run a single game between a human and an agent with a GUI.
    :param agent: The agent to play against (must be a subclass of GameAgent)
    """
    my_go = GoProblem(size=size)
    state = my_go.start_state
    gui = GoGUI(my_go)
    while (not my_go.is_terminal_state(state)):
        player1_action = agent.get_move(state.clone(), 1)
        state = my_go.transition(state, player1_action)
        gui.update_state(state)
        gui.render()
        if (my_go.is_terminal_state(state)):
            break
        action = None
        while action is None:
            while action not in state.legal_actions():
                action = gui.get_user_input_action()
                gui.render()
                clock.tick(60)
            print("Human Action:", action, ", which corresponds to coordinate ", my_go.action_index_to_string(action))
            gui.render()
            clock.tick(60)
        state = my_go.transition(state, action)
        gui.update_state(state)
        gui.render()
        clock.tick(60)
    print("Done!")
    if my_go.evaluate_terminal(state) == 1:
        print("Agent wins!")
    else:
        print("You won!")

def create_agent(agent_type: str, **kwargs):
    """
    Factory function to create agents based on command line arguments

    :param agent_type: The type of agent to create (string)
    :param kwargs: Additional arguments for the agent (e.g., depth, parameters, etc.)
    """
    if agent_type.lower() == "alphabeta":
        depth = kwargs.get('depth', 2)
        return AlphaBetaAgent(depth=depth)
    elif agent_type.lower() == "random":
        return RandomAgent()
    elif agent_type.lower() == "greedy":
        return GreedyAgent()
    elif agent_type.lower() == "mcts":
        return MCTSAgent()
    # Add more agent types here as needed
    else:
        raise ValueError(f"Unknown agent type: {agent_type}")

def parse_args():
    parser = argparse.ArgumentParser(description='Go Game Runner')

    # Mode selection
    parser.add_argument('--mode', choices=['gui', 'vs', 'tournament'], default='gui',
                        help='Run mode: gui (play against AI), vs (single game between agents), tournament (multiple games)')

    # Agent configuration
    parser.add_argument('--agent1-type', default='alphabeta',
                        help='Type of agent 1 (e.g., alphabeta)')
    parser.add_argument('--agent1-depth', type=int, default=2,
                        help='Depth limit for agent 1 if applicable')

    parser.add_argument('--agent2-type', default='alphabeta',
                        help='Type of agent 2 (e.g., alphabeta)')
    parser.add_argument('--agent2-depth', type=int, default=2,
                        help='Depth limit for agent 2 if applicable')

    # Game settings
    parser.add_argument('--time-limit', type=float, default=15,
                        help='Time limit per player in seconds')
    parser.add_argument('--time-increment', type=float, default=1,
                        help='Time increment per move in seconds')
    parser.add_argument('--soft-time', action='store_true',
                        help='Continue game even if time limit is exceeded')
    parser.add_argument('--size', type=int, default=5,
                        help='Size of the Go board')

    # Tournament settings
    parser.add_argument('--num-games', type=int, default=10,
                        help='Number of games to play in tournament mode')
    parser.add_argument('--quiet', action='store_true',
                        help='Suppress detailed output in tournament mode')

    args = parser.parse_args()
    return args

def main():
    args = parse_args()

    # Create agents based on arguments
    agent1 = create_agent(args.agent1_type, depth=args.agent1_depth)

    if args.mode == 'gui':
        run_game_with_gui(agent1)
    else:
        agent2 = create_agent(args.agent2_type, depth=args.agent2_depth)
        if args.mode == 'vs':
            result, info = run_game(agent1, agent2,
                                    time_limit=args.time_limit,
                                    time_increment=args.time_increment,
                                    hard_time_cutoff=not args.soft_time,
                                    size=args.size)
            print("Game Info:", info)
        elif args.mode == 'tournament':
            run_many(agent1, agent2,
                     num_games=args.num_games,
                     verbose=not args.quiet,
                     size=args.size)


if __name__ == "__main__":
    main()

import pygame
import sys
from go_search_problem import GoProblem, GoState


class GoGUI:
    # Define GUI colors
    BOARD = (210, 180, 140)  # brown
    EMPTY = (0, 0, 0)  # black
    P1 = (0, 0, 0)  # black
    P2 = (255, 255, 255)  # white
    BUTTON = (200, 200, 200)  # grey
    BUTTON_HOVER = (180, 180, 180)  # darker grey
    BUTTON_TEXT = (0, 0, 0)  # black
    COLOR_MAP = [EMPTY, P1, P2]

    def __init__(self, problem: GoProblem):
        # Initialize Pygame
        print("Setting up Board...")
        print("Use the arrow keys to navigate and the enter key to select an action.")
        pygame.init()

        # Constants
        self.WIDTH, self.HEIGHT = 600, 700  # Increased height for pass button
        self.BOARD_SIZE = problem.start_state.size
        self.CELL_SIZE = 600 // self.BOARD_SIZE  # Using original width for board

        # Pass button dimensions
        self.BUTTON_WIDTH = 100
        self.BUTTON_HEIGHT = 40
        self.BUTTON_X = (self.WIDTH - self.BUTTON_WIDTH) // 2
        self.BUTTON_Y = 620  # Position below the board
        self.BUTTON_COLOR = self.BUTTON

        # Set up the display
        self.screen = pygame.display.set_mode((self.WIDTH, self.HEIGHT))
        pygame.display.set_caption("Go Game")

        # Initialize font
        self.font = pygame.font.Font(None, 36)

        self.problem = problem
        self.state = problem.start_state
        self.cursor_pos = [self.BOARD_SIZE // 2, self.BOARD_SIZE // 2]

    def render(self):
        self.screen.fill(self.BOARD)
        self.draw_board()
        self.draw_pieces()
        self.draw_cursor()
        self.draw_pass_button()
        pygame.display.flip()

    def draw_pass_button(self):
        # Check if mouse is hovering over button
```

```python
        mouse_pos = pygame.mouse.get_pos()
        button_rect = pygame.Rect(self.BUTTON_X, self.BUTTON_Y, self.BUTTON_WIDTH, self.BUTTON_HEIGHT)
        button_color = self.BUTTON_HOVER if button_rect.collidepoint(mouse_pos) else self.BUTTON_COLOR

        # Draw button
        pygame.draw.rect(self.screen, button_color, button_rect)
        pygame.draw.rect(self.screen, self.BUTTON_TEXT, button_rect, 2)  # Border

        # Draw text
        text = self.font.render("PASS", True, self.BUTTON_TEXT)
        text_rect = text.get_rect(center=button_rect.center)
        self.screen.blit(text, text_rect)

    def process_window_event(self, event):
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    def is_pass_button_clicked(self, pos):
        button_rect = pygame.Rect(self.BUTTON_X, self.BUTTON_Y, self.BUTTON_WIDTH, self.BUTTON_HEIGHT)
        return button_rect.collidepoint(pos)

    def get_user_input_action(self):
        for event in pygame.event.get():
            self.process_window_event(event)

            if event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1:  # Left click
                    if self.is_pass_button_clicked(event.pos):
                        return self.BOARD_SIZE * self.BOARD_SIZE  # Pass move

            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_UP:
                    self.cursor_pos[1] = max(0, self.cursor_pos[1] - 1)
                elif event.key == pygame.K_DOWN:
                    self.cursor_pos[1] = min(
                        self.BOARD_SIZE - 1, self.cursor_pos[1] + 1)
                elif event.key == pygame.K_LEFT:
                    self.cursor_pos[0] = max(0, self.cursor_pos[0] - 1)
                elif event.key == pygame.K_RIGHT:
                    self.cursor_pos[0] = min(
                        self.BOARD_SIZE - 1, self.cursor_pos[0] + 1)
                elif event.key == pygame.K_RETURN:
                    return self.cursor_pos[1] * self.BOARD_SIZE + self.cursor_pos[0]
                elif event.key == pygame.K_SPACE:  # Added space as alternative for pass
```

```python
                    return self.BOARD_SIZE * self.BOARD_SIZE
        return None

    def update_state(self, action):
        if action is not None and action in self.problem.get_available_actions(self.state):
            self.state = self.problem.transition(self.state, action)
        elif action is not None:
            self.state = action

    def draw_cursor(self):
        x, y = self.cursor_pos
        pygame.draw.rect(self.screen, (255, 0, 0),
                         (x * self.CELL_SIZE, y * self.CELL_SIZE, self.CELL_SIZE, self.CELL_SIZE), 3)

    def draw_board(self):
        for i in range(self.BOARD_SIZE):
            # Draw horizontal lines
            pygame.draw.line(self.screen, self.EMPTY, (0, i * self.CELL_SIZE),
                             (600, i * self.CELL_SIZE))
            # Draw vertical lines
            pygame.draw.line(self.screen, self.EMPTY, (i * self.CELL_SIZE, 0),
                             (i * self.CELL_SIZE, 600))
        # Draw bottom line
        pygame.draw.line(self.screen, self.EMPTY, (0, self.BOARD_SIZE * self.CELL_SIZE),
                             (600, self.BOARD_SIZE * self.CELL_SIZE))

    def draw_pieces(self):
        board = self.state.get_board()
        for y in range(self.BOARD_SIZE):
            for x in range(self.BOARD_SIZE):
                if board[0][y][x] == 1:
                    self.draw_piece(x, y, self.P1)
                elif board[1][y][x] == 1:
                    self.draw_piece(x, y, self.P2)

    def draw_piece(self, x, y, color):
        center = (x * self.CELL_SIZE + self.CELL_SIZE // 2,
                  y * self.CELL_SIZE + self.CELL_SIZE // 2)
        pygame.draw.circle(self.screen, color, center, self.CELL_SIZE // 2 - 2)


def main():
    problem = GoProblem()
    gui = GoGUI(problem)
    clock = pygame.time.Clock()

    while True:
        action = gui.get_user_input_action()
        gui.update_state(action)
        gui.render()
        clock.tick(60)


if __name__ == "__main__":
    main()

from typing import Sequence, Type
# import go_utils
import numpy as np
from adversarial_search_problem import AdversarialSearchProblem, GameState
import copy
import go_utils
from pyspiel import Game

Action = int

DEFAULT_SIZE = 9

class GoState(GameState):
    """
    A state of the game of Go.
    Includes methods and properties for the state of the board, player to 
move, and other useful methods
    """
    def __init__(self, pyspiel_state: Game, player_to_move: int = 0):
        """
        Initialize GoState with pyspiel as backend Go engine.
        The initial state is created with a call to create_go_game() in 
go_utils.py
        Every other state will be generated from applying actions to the 
initial state.
        This essentially functions as a wrapper class to conver pyspiel game 
states to
        The ASP interface used previously.

        :param pyspiel_state: pyspiel state of the game
        :param player_to_move: player to move
        """
        self.internal_state = pyspiel_state
        self.size = int(np.sqrt(len(pyspiel_state.observation_tensor()) / 4))


    def player_to_move(self) -> int:
        """
        Get the current player to move
        :return: player to move BLACK (0) or WHITE (1)
        """
        return self.internal_state.current_player()

    def get_board(self) -> np.ndarray:
        """
        Return the current board as a numpy array
        The board will have shape (4, size, size)
        The first channel (i.e., get_board()[0]) is the board for BLACK. There are 1's where the black pieces are and 0's elsewhere.
        The second channel (i.e., get_board()[1]) is the board for WHITE. There are 1's where the white pieces are and 0's elsewhere.
        The third channel (i.e., get_board()[2]) is the board for EMPTY. There are 1's where the empty spaces are and 0's elsewhere.
        The fourth channel (i.e., get_board()[3]) is the board for whose turn it is. There are 0's when it is BLACK's turn and 1's when it is white's.

        This is the default observation tensor used by pyspiel.
        """
        return np.array(self.internal_state.observation_tensor(0)).reshape(-1, self.size, self.size)

    def terminal_value(self) -> float:
        """
        Return the terminal value of the game.
        :return: 1 if BLACK wins, -1 if WHITE wins
        """
        return self.internal_state.returns()

    def clone(self) -> GameState:
        """
        Create a copy of the current game state.
        This is used for safety with the game runner.
        We don't want search algorithms to be able to directly modify the game state,
        so we only pass a copy of the state to the search algorithms.
        :return: a copy of the current game state
        """
        return GoState(self.internal_state.clone(), self.internal_state.current_player())

    def is_terminal_state(self) -> bool:
        """
        Checks if the game is in a terminal state.
        The state is if there are no legal actions left or the players have passed twice in a row.

        :return: True if the game is in a terminal state, False otherwise
        """
        return self.internal_state.is_terminal()

    def legal_actions(self) -> Sequence[Action]:
        """
        Return all possible legal actions for the given state.
        Note: Actions are represented as integers, by default.
        For a more human-readable representation, use action_index_to_coord()

        NOTE: It is preferrable to get the available actions from the search problem,
        not this state.

        :return: list of legal actions
        """
        return self.internal_state.legal_actions()

    def apply_action(self, action: Action):
        """
        Apply action and update internal state.
        Action must be an int, not a coordinate.

        NOTE: It is preferrable to use the transition function from the search problem,
        not this method to apply actions.
        """
        self.internal_state.apply_action(action)

    def get_pieces_coordinates(self, player_index: int):
        """
        Get the indices of the pieces of the given player.
        :param player_index: 0 for BLACK, 1 for WHITE
        :return: list of coordinates of the pieces of the given player
        """
        player_board = np.array(self.internal_state.observation_tensor(
            0)).reshape((-1, self.size, self.size))[player_index]
        return np.argwhere(player_board == 1)

    def get_pieces_array(self, player_index):
        """
        Get the 2D array of the pieces of the given player.
        The array will have shape (size, size) and will have 1's where the pieces are and 0's elsewhere.

        :param player_index: 0 for BLACK, 1 for WHITE
        :return: 2D np array of the pieces of the given player
        """
        player_board = np.array(self.internal_state.observation_tensor(
            0)).reshape((-1, self.size, self.size))[player_index]
        return player_board

    def get_empty_spaces(self):
        """
        return a 2D array of the empty spaces on the board
        The array will have shape (size, size) and will have 1's where the empty spaces are and 0's elsewhere.

        :return: 2D np array of the empty spaces on the board
        """
        return self.internal_state.observation_tensor(2)

    def action_index_to_coord(self, action: Action) → tuple[int, int]:
        """
        Convert an action index to a coordinate.
        :param action: action index
        :return: coordinate (x, y)
        """
        return (action % self.size, action // self.size)

    def __repr__(self):
        return str(self.internal_state)


class GoProblem(AdversarialSearchProblem[GoState, Action]):
    def __init__(self, size=DEFAULT_SIZE, state=None, player_to_move=0):
        """
        Create a new Go search problem.
        If no state is provided, a new game is created with the given size.
        """
        if state is None:
            game_state = go_utils.create_go_game(size)
        else:
            game_state = state
        self.start_state = GoState(game_state, player_to_move)

    def get_available_actions(self, state: GoState) → Sequence[Action]:
        """
        Get the available actions for the given state.
        Use this to get the list of available actions for a given state.
        Note: An action in this case is an integer in range [0, size^2].
        Each action index corresponds to a coordinate on the board (x, y) =
(action % size, action // size).
        With action=size**2 reserved for the pass action.

        :param state: current state
        :return: list of available actions
        """
        return state.legal_actions()

    def transition(self, state: GoState, action: Action) → GoState:
        """
        Return new_state resulting from applying action to state.

        :param state: current state
        :param action: action to apply
        :return: new state resulting from applying action to state
        """
        new_state = state.clone()
        new_state.apply_action(action)
        return new_state

    def is_terminal_state(self, state: GoState) -> bool:
        """
        Return if the given state is a terminal state.
        State is terminal if no legal actions are available or the players have passed twice in a row.

        :param state: current state
        :return: True if the state is terminal, False otherwise
        """
        return state.is_terminal_state()

    def evaluate_terminal(self, state: GoState) -> float:
        """
        Get the value of the terminal state.
        The value is 1 if BLACK wins and -1 if WHITE wins.

        :param state: current state
        :return: value of the terminal state
        """
        return state.terminal_value()[0]

    def action_index_to_string(self, action: Action) -> str:
        """
        Convert an Action (index) to a string.
        """
        return "(" + str(action % self.start_state.size) + ", " +  str(action // self.start_state.size) + ")"

import numpy as np
import pyspiel
import pygame
import sys


def create_go_game(size):
    """
    load open-spiel game with provided size
    """
    if size == 5:
        komi = 0.5
    elif size == 9:
        komi = 5.5
    else:
        komi = 7.5
    game = pyspiel.load_game("go", {"board_size": size, "komi": komi})
    state = game.new_initial_state()
    return state

```

```python

from go_search_problem import GoProblem

BLACK = 0
WHITE = 1

class GoProblemSimpleHeuristic(GoProblem):
    def __init__(self, state=None):
        super().__init__(state=state)

    def heuristic(self, state, player_index):
        """
        Very simple heuristic that just compares the number of pieces for each player

        Having more pieces (>1) than the opponent means that some were captured, capturing is generally good.
        """
        return len(state.get_pieces_coordinates(BLACK)) - len(state.get_pieces_coordinates(WHITE))

    def __str__(self) -> str:
        return "Simple Heuristic"


class GoProblemLearnedHeuristic(GoProblem):
    def __init__(self, model=None, state=None,):
        super().__init__(state=state)
        self.model = model

    def encoding(self, state):
        pass

    def heuristic(self, state, player_index):
        pass

    def __str__(self) -> str:
        return "Learned Heuristic"


class GoProblemAdvancedHeuristic(GoProblem):
    def __init__(self, state=None):
        super().__init__(state=state)

    def heuristic(self, state, player_index):
        """
        Advanced heuristic for evaluating a Go game state based on:
        1. Piece Count: Difference in the number of stones between the player and opponent.
        2. Territory Control: Influence over empty spaces, based on proximity to placed stones.
        3. Center Control: Control over the central area of the board.
        4. Liberties: Number of adjacent empty spaces around the player's stones.
        5. Weighted Scoring

        :param state: Current game state
        :param player_index: Player to evaluate (0 for BLACK, 1 for WHITE)
        :return: Heuristic score favoring the current player
        """

        # if player_index=0 (BLACK), opponent_index=1 (WHITE)
        # if player_index=1 (WHITE), opponent_index=0 (BLACK)
        opponent_index = 1 - player_index

        board = state.get_board()
        size = state.size

        # 1. Piece count
        player_pieces = len(state.get_pieces_coordinates(player_index))
        opponent_pieces = len(state.get_pieces_coordinates(opponent_index))
        piece_difference = player_pieces - opponent_pieces

        # 2. Territory control
        # Use empty spaces as a proxy for potential territory
        empty_spaces = board[2]

        player_board = state.get_pieces_array(player_index)
        opponent_board = state.get_pieces_array(opponent_index)

        def count_potential_territory(player_board, empty_spaces):
            territory_score = 0
            for y in range(size):
                for x in range(size):
                    if empty_spaces[y, x] == 1:
                        # Count the number of player's stones in a 3x3 neighborhood around the empty space (y, x).
                        nearby_player_stones = player_board[max(0, y-1):min(size, y+2),
                                                            max(0, x-1):min(size, x+2)].sum()
                        territory_score += nearby_player_stones
            return territory_score

        player_territory = count_potential_territory(player_board, empty_spaces)
        opponent_territory = count_potential_territory(opponent_board, empty_spaces)
        territory_difference = player_territory - opponent_territory

        # 3. Center control
        def center_control_score(board):
```

```
      # Evaluate center control by counting the number of stones each player has in the central region of the board.
      # Note: center_range[0] gives the start of range and center_range[-1] gives the end of the range.
      center_range = range(size // 4, 3 * size // 4)
      center_board = board[center_range[0]:center_range[-1], center_range[0]:center_range[-1]]
      return center_board.sum()

    player_center_control = center_control_score(player_board)
    opponent_center_control = center_control_score(opponent_board)
    center_control_difference = player_center_control - opponent_center_control

    # 4. Liberties
    def count_liberties(board):
      liberties = 0

      # For each player's stone, count the number of adjacent empty spaces (liberties).
      for y in range(size):
        for x in range(size):
          if board[y, x] == 1:
            liberties += sum([
              (y > 0 and empty_spaces[y-1, x] == 1),
              (y < size-1 and empty_spaces[y+1, x] == 1),
              (x > 0 and empty_spaces[y, x-1] == 1),
              (x < size-1 and empty_spaces[y, x+1] == 1)
            ])
      return liberties

    player_liberties = count_liberties(player_board)
    opponent_liberties = count_liberties(opponent_board)
    liberty_difference = player_liberties - opponent_liberties

    # Weights for each component
    weights = {
      'pieces': 2.0,
      'territory': 1.5,
      'center_control': 1.0,
      'liberties': 1.0
    }

    # Combine scores with weights
    total_score = (
      weights['pieces'] * piece_difference +
      weights['territory'] * territory_difference +
      weights['center_control'] * center_control_difference +
      weights['liberties'] * liberty_difference
    )

```

```python
        normalized_score = total_score / (size * size)

        return normalized_score if player_index == BLACK else -normalized_score

    def __str__(self) -> str:
        return "Advanced Heuristic"

# %%
# Needed if running on Colab
!pip3 install open-spiel
!pip3 install torch

# %%
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import random
from go_search_problem import GoProblem, GoState
from heuristic_go_problems import GoProblemLearnedHeuristic, GoProblemSimpleHeuristic
from agents import GreedyAgent, RandomAgent, MCTSAgent, GameAgent
import matplotlib.pyplot as plt
from tqdm import tqdm
from game_runner import run_many
import pickle

torch.set_default_tensor_type(torch.FloatTensor)

# %%
def load_dataset(path: str):
    with open(path, 'rb') as f:
        dataset = pickle.load(f)
    return dataset

dataset_5x5 = load_dataset('dataset_5x5.pkl')
# dataset_9x9 = load_dataset('9x9_dataset.pkl')

# %%
def save_model(path: str, model, input_size=None):
    """
    Save model to a file
    Input:
        path: path to save model to
        model: Pytorch model to save
    """

    torch.save({
        'model_state_dict': model.state_dict(),
    }, path)
```

```

def load_model(path: str, model):
    """
    Load model from file

    Note: you still need to provide a model (with the same architecture as the saved model))

    Input:
        path: path to load model from
        model: Pytorch model to load
    Output:
        model: Pytorch model loaded from file
    """
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint['model_state_dict'])
    return model

# %% [markdown]
# # Task 1: Convert GameState to Features

# %%
def get_features(game_state: GoState):
    """
    Map a game state to a list of features.

    Some useful functions from game_state include:
        game_state.size: size of the board
        get_pieces_coordinates(player_index): get coordinates of all pieces of a player (0 or 1)
        get_pieces_array(player_index): get a 2D array of pieces of a player (0 or 1)

        get_board(): get a 2D array of the board with 4 channels (player 0, player 1, empty, and player to move). 4 channels means the array will be of size 4 x n x n

        Descriptions of these methods can be found in the GoState

    Input:
        game_state: GoState to encode into a fixed size list of features
    Output:
        features: list of features
    """
    board_size = game_state.size

    # TODO: Encode game_state into a list of features
    features = []

    board = game_state.get_board()

    for channel in range(4):
        for row in range(board_size):
            for col in range(board_size):
                features.append(board[channel][row][col])

    return features

# %%
#TESTING
class MockGoState:
    def __init__(self, size, board, player_to_move):
        """
        Mock implementation of the GoState class for testing.
        :param size: Size of the board (n x n).
        :param board: A 3D list (4 x n x n) representing the board state.
        :param player_to_move: The current player to move (0 or 1).
        """
        self.size = size
        self.board = board
        self.player_to_move = player_to_move

    def get_board(self):
        return self.board

def test_get_features():
    # Test Case 1: Empty 3×3 Board with Player 0 to move
    size = 3
    board = [
        # Player 0 (white)
        [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
        # Player 1 (black)
        [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
        # Empty spaces
        [[1, 1, 1], [1, 1, 1], [1, 1, 1]],
        # Player to move (0 for Player 0, 1 for Player 1)
        [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
    ]
    game_state = MockGoState(size, board, player_to_move=0)
    features = get_features(game_state)
    assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got {len(features)}"
    assert features == [0] * 9 + [0] * 9 + [1] * 9 + [1] * 9, "Feature vector does not match expected solution."

    # Test Case 2: Filled 3×3 Board with Player 1 to move
    board = [
        [[0, 1, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [1, 0, 0], [0, 0, 0]],
        [[1, 0, 1], [0, 1, 1], [1, 1, 1]],
        [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
    ]
```

```python
    game_state = MockGoState(size, board, player_to_move=1)
    features = get_features(game_state)
    assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got {len(features)}"
    expected_features = [
        0, 1, 0,  0, 0, 0,  0, 0, 0,
        0, 0, 0,  1, 0, 0,  0, 0, 0,
        1, 0, 1,  0, 1, 1,  1, 1, 1,
        0, 0, 0,  0, 0, 0,  0, 0, 0
    ]
    assert features == expected_features, f"Feature vector does not match expected solution."

    # Test Case 3: Filled 2x2 Board with Player 0 to move
    size = 2
    board = [
        [[1, 0], [0, 1]],
        [[0, 1], [1, 0]],
        [[0, 0], [0, 0]],
        [[1, 1], [1, 1]]
    ]
    game_state = MockGoState(size, board, player_to_move=0)
    features = get_features(game_state)
    assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got {len(features)}"
    expected_features = [
        1, 0,  0, 1,
        0, 1,  1, 0,
        0, 0,  0, 0,
        1, 1,  1, 1
    ]
    assert features == expected_features, f"Feature vector does not match expected solution."

    print("All tests passed!")

test_get_features()

# %%
# Print information about first data point
data_point = dataset_5x5[0]
features = get_features(data_point[0])
action = data_point[1]
result = data_point[2]
print(data_point[0])
print("features", features)
print("Action #", action)
print("Game Result", result)

# %% [markdown]
# # Task 2: Supervised Learning of a Value Network

# %%
class ValueNetwork(nn.Module):
    def __init__(self, input_size):
        super(ValueNetwork, self).__init__()

        # TODO: What should the output size of a Value function be?

        ''' Handout: the goal is to classify each state as a future
        win for one player or the other, or more generally, to
        generate a prediction in the range [-1, +1] that is indicative
        of which player will win the game.'''

        output_size = 1

        # TODO: Add more layers, non-linear functions, etc.

        # Layers
        self.fc1 = nn.Linear(input_size, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, output_size)

        # Activation functions
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        """
        Run forward pass of network

        Input:
        x: input to network
        Output:
        output of network
        """
        # TODO: Update as more layers are added
        z1 = self.fc1(x)
        a1 = self.relu(z1)

        z2 = self.fc2(a1)
        a2 = self.relu(z2)

        z3 = self.fc3(a2)
        a3 = self.relu(z3)

        return a3

# %%
# This will not produce meaningful outputs until trained, but you can test for syntax errors
features_tensor = torch.Tensor(features)
```

```python
value_net = ValueNetwork(len(features))
print("predicted Value", value_net(features_tensor))

# %%
def train_value_network(dataset, num_epochs, learning_rate):
    """
    Train a value network on the provided dataset.

    Input:
        dataset: list of (state, action, result) tuples
        num_epochs: number of epochs to train for
        learning_rate: learning rate for gradient descent
    Output:
        model: trained model
    """
    # Make sure dataset is shuffled for better performance
    random.shuffle(dataset)
    # You may find it useful to create train/test sets to better track performance/overfit/underfit
    train_size = int(0.8 * len(dataset))
    train_dataset = dataset[:train_size]
    test_dataset = dataset[train_size:]

    # Get input size
    sample_features = get_features(dataset[0][0])
    input_size = len(sample_features)

    # TODO: Create model
    model = ValueNetwork(input_size)

    # TODO: Specify Loss Function
    loss_function = nn.MSELoss()

    # You can use Adam, which is stochastic gradient descent with ADAptive Momentum
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    batch_size = 32

    batch_loss = 0
    batch_counter = 0

    for epoch in range(num_epochs):
        total_train_loss = 0.0
        for data_point in train_dataset:
            state = data_point[0]
            features = get_features(state)
            features_tensor = torch.tensor(features, dtype=torch.float32)

            # TODO: What should the desired output of the value network be?
            # Note: You will have to convert the label to a torch tensor to use with torch's loss functions
            label = torch.tensor(data_point[2], dtype=torch.float32)

            # TODO: Get model prediction of value
            prediction = model(features_tensor)

            # TODO: Compute Loss for data point
            train_loss = loss_function(prediction, label)
            batch_loss += train_loss
            batch_counter += 1
            total_train_loss += train_loss

            if batch_counter % batch_size == 0:
                # Call backward to run backward pass and compute gradients
                batch_loss.backward()

                # Run gradient descent step with optimizer
                optimizer.step()

                # Reset gradient for next batch
                optimizer.zero_grad()

                batch_loss = 0

        total_test_loss = 0
        with torch.no_grad():
            for data_point in test_dataset:
                state = data_point[0]
                features = get_features(state)
                features_tensor = torch.tensor(features, dtype=torch.float32)
                label = torch.tensor(data_point[2], dtype=torch.float32)

                prediction = model(features_tensor)
                test_loss = loss_function(prediction, label)
                total_test_loss += test_loss

        avg_train_loss = total_train_loss / len(train_dataset)
        avg_test_loss = total_test_loss / len(test_dataset)

        print(f'Epoch {epoch+1}/{num_epochs}:')
        print(f'  Training Loss: {avg_train_loss:.4f}')
        print(f'  Testing Loss: {avg_test_loss:.4f}')

    return model

value_model = train_value_network(dataset_5x5, 10, 1e-4)
save_model("value_model.pt", value_model)

# %% [markdown]
# ## Comparing Learned Value function against other Agents
```

```python

# %%
class GoProblemLearnedHeuristic(GoProblem):
    def __init__(self, model=None, state=None):
        super().__init__(state=state)
        self.model = model

    def __call__(self, model=None):
        """
        Use the model to compute a heuristic value for a given state.
        """
        return self

    def encoding(self, state):
        """
        Get encoding of state (convert state to features)
        Note, this may call get_features() from Task 1.

        Input:
            state: GoState to encode into a fixed size list of features
        Output:
            features: list of features
        """
        # TODO: get encoding of state (convert state to features)

        return get_features(state)

    def heuristic(self, state, player_index):
        """
        Return heuristic (value) of current state

        Input:
            state: GoState to encode into a fixed size list of features
            player_index: index of player to evaluate heuristic for
        Output:
            value: heuristic (value) of current state
        """
        # TODO: Compute heuristic (value) of current state
        value = 0

        features = self.encoding(state)
        features_tensor = torch.tensor(features, dtype=torch.float32)

        with torch.no_grad():
            value = self.model(features_tensor)

        '''value = max(-1, min(1, value))
        if player_index ≠ state.player_to_move():
            value = -value'''

        # Note, your agent may perform better if you force it not to pass
        # (i.e., don't select action #25 on a 5×5 board unless necessary)
        return value

    def __str__(self) -> str:
        return "Learned Heuristic"

import go_utils
def create_value_agent_from_model():
    """
    Create agent object from saved model. This (or other methods like this) will be how your agents will be created in gradescope and in the final tournament.
    """

    model_path = "value_model.pt"
    # TODO: Update number of features for your own encoding size

    feature_size = len(get_features(dataset_5x5[0][0]))

    model = load_model(model_path, ValueNetwork(feature_size))

    heuristic_search_problem = GoProblemLearnedHeuristic(model)

    # TODO: Try with other heuristic agents (IDS/AB/Minimax)
    learned_agent = GreedyAgent(heuristic_search_problem)

    return learned_agent

# learned_agent = create_value_agent_from_model(value_net)
learned_agent = create_value_agent_from_model()
agent2 = GreedyAgent(GoProblemSimpleHeuristic)
print("Greedy Agent", agent2)
print("Learned Agent", learned_agent)

run_many(learned_agent, GreedyAgent(), 40)

# %% [markdown]
# # Task 3: Supervised Learning of a Policy Network

# %%
class PolicyNetwork(nn.Module):
    def __init__(self, input_size, board_size=5):
      super(PolicyNetwork, self).__init__()

      # TODO: What should the output size of the Policy be?
      self.output_size = board_size * board_size + 1

      # TODO: Add more layers, non-linear functions, etc.
      self.fc1 = nn.Linear(input_size, 512)
      self.fc2 = nn.Linear(512, 128)
      self.fc3 = nn.Linear(128, 64)
```

```python
        self.fc4 = nn.Linear(64, self.output_size)

        self.relu = nn.ReLU()

    def forward(self, x):
        # TODO: Update as more layers are added
        z1 = self.fc1(x)
        a1 = self.relu(z1)
        z2 = self.fc2(a1)
        a2 = self.relu(z2)
        z3 = self.fc3(a2)
        a3 = self.relu(z3)
        z4 = self.fc4(a3)

        return z4

# %%
# This will not produce meaningful outputs until trained, but you can test for syntax errors
features_tensor = torch.Tensor(features)
policy_net = PolicyNetwork(len(features))
print("Predicted Action Probabilities", policy_net(features_tensor))

# %%
def train_policy_network(dataset, num_epochs, learning_rate):
    """
    Train a policy network on the provided dataset.

    Input:
        dataset: list of (state, action, result) tuples
        num_epochs: number of epochs to train for
        learning_rate: learning rate for gradient descent
    Output:
        model: trained model
    """
    random.shuffle(dataset)

    input_size = len(get_features(dataset[0][0]))
    # input_size = 100


    # TODO: Create model
    model = PolicyNetwork(input_size, 5)
    print(f"Output size: {model.output_size}")

    # TODO: Specify Loss Function
    loss_function = nn.CrossEntropyLoss()

    # You can use Adam, which is stochastic gradient descent with ADAptive Momentum
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    batch_size = 32
    batch_loss = 0
    batch_counter = 0

    for epoch in range(num_epochs):
        total_train_loss = 0
        num_train_correct = 0

        for data_point in dataset:
            # TODO: Get features from state and convert features to torch tensor
            state = data_point[0]

            features = get_features(state)
            features_tensor = torch.tensor(features, dtype=torch.float32)

            # TODO: What should the desired output of the value network be?
            # Note: You will have to convert the label to a torch tensor to use with torch's loss functions
            action = data_point[1]
            label = torch.tensor(action, dtype=torch.long)

            # TODO: Get model estimate of value
            prediction = model(features_tensor)

            # TODO: Compute Loss for data point
            train_loss = loss_function(prediction, label)
            batch_loss += train_loss
            batch_counter += 1
            total_train_loss += train_loss

            if batch_counter % batch_size == 0:
                # Call backward to run backward pass and compute gradients
                batch_loss.backward()

                # Run gradient descent step with optimizer
                optimizer.step()

                optimizer.zero_grad()

                batch_loss = 0


        train_accuracy = num_train_correct / len(dataset)
        avg_train_loss = total_train_loss / len(dataset)

        print(f'Epoch {epoch+1}/{num_epochs}:')
        print(f'  Training Loss: {avg_train_loss:.4f}')

    # torch.save(model.state_dict(), "policy_model.pt")
    return model

policy_net = train_policy_network(dataset_5x5, 10, 1e-4)
save_model("policy_model.pt", policy_net)

# %% [markdown]
# ## Comparing Learned Policy against other Agents

# %%
class PolicyAgent(GameAgent):
    def __init__(self, search_problem, model_path, board_size=5):
        super().__init__()
        self.search_problem = search_problem
        # self.model = load_model(model_path, PolicyNetwork)

        input_size = len(get_features(dataset_5x5[0][0]))
        model_template = PolicyNetwork(input_size, board_size)
        self.model = load_model(model_path, model_template)

        self.board_size = board_size

    def encoding(self, state):
        # TODO: get encoding of state (convert state to features)
        return get_features(state)

    def get_move(self, game_state, time_limit=1):
      """
      Get best action for current state using self.model

      Input:
        game_state: current state of the game
        time_limit: time limit for search (This won't be used in this agent)
      Output:
        action: best action to take
      """
      legal_actions = self.search_problem.get_available_actions(game_state)

      features = self.encoding(game_state)
      features_tensor = torch.tensor(features, dtype=torch.float32).unsqueeze(0)

      with torch.no_grad():
          action_logits = self.model(features_tensor)
          action_probs = torch.softmax(action_logits, dim=1).squeeze(0)

      all_probs = action_probs.tolist()

      # Get probabilities for legal actions
      legal_actions_probs = [
          (action, all_probs[action-1] if 1 <= action <= len(all_probs) else 0)
```

```
            for action in legal_actions
        ]

        # Sort legal actions by probability
        sorted_legal_actions = sorted(legal_actions_probs, key=lambda x: x[1], reverse=True)

        # Return best legal action
        return sorted_legal_actions[0][0] if sorted_legal_actions else None

    def __str__(self) -> str:
        return "Policy Agent"

def create_policy_agent_from_model():
    """
    Create agent object from saved model. This (or other methods like this) will be how your agents will be created in gradescope and in the final tournament.
    """

    model_path = "policy_model.pt"
    agent = PolicyAgent(GoProblem(size=5), model_path)
    return agent

# %%
# policy_agent = PolicyAgent(GoProblem(size=5), policy_net)
policy_agent = create_policy_agent_from_model()
print("Policy Agent", policy_agent)
run_many(policy_agent, GreedyAgent(), 40)

# %% [markdown]
# # Submitting
#
# After you've completed all the tasks in this notebook, you'll want to add your agents to your agents.py file. You'll want to copy the necessary function and class definitions for PolicyAgent, GoProblemLearnedHeuristic, PolicyNetwork, ValueNetwork, and any other methods you referenced. Your agents will ultimately be tested on gradescope by calling create_value_agent_from_model or by create_policy_agent_from_model.



```