

```

1  from abc import ABC, abstractmethod
2  from typing import Generic, Set, Tuple, TypeVar
3
4  #####
5  # An AdversarialSearchProblem is a representation of a game that is convenient
6  # for running adversarial search algorithms.
7  #
8  # A game can be put into this form by extending the AdversarialSearchProblem
9  # class. See ttproblem.py for an example of this.
10 #
11 # Every subclass of AdversarialSearchProblem has its game states represented
12 # as instances of a subclass of GameState. The only requirement that of a
13 # subclass of GameState is that it must implement that player_to_move(.)
14 # method,
15 # which returns the index (0-indexed) of the next player to move.
16 #####
17
18 class GameState(ABC):
19     @abstractmethod
20     def player_to_move(self) → int:
21         """
22         Output- Returns the index of the player who will move next.
23         """
24         pass
25
26
27 State = TypeVar("State", bound=GameState)
28
29 # Action represents the type of actions that an instance of
30 # AdversarialSearchProblem uses to
31 # cause a transition. It's generic because different games have different
32 # actions: TTT requires
33 # placing a piece on a *2D* grid, while Connect 4 just involves selecting a
34 # column.
35 Action = TypeVar("Action")
36
37 class AdversarialSearchProblem(ABC, Generic[State, Action]):
38     def get_start_state(self):
39         """
40         Output- Returns the state from which to start.
41         """
42         return self._start_state
43
44     def set_start_state(self, state: State):
45         """
46         Changes the start state to the given state.
47         Note to student: You should not need to use this.
48         This is only for running games.

```

```

47
48     Input:
49         state- a GameState
50     """
51     self._start_state = state
52
53     @abstractmethod
54     def get_available_actions(self, state: State) → Set[Action]:
55         """
56         Input:
57             state- a GameState
58         Output:
59             Returns the set of actions available to the player-to-move
60             from the given state
61         """
62         pass
63
64     @abstractmethod
65     def transition(self, state: State, action: Action) → State:
66         """
67         Input:
68             state- a GameState
69             action- the action to take
70         Output:
71             Returns the state that results from taking the given action
72             from the given state. (Assume deterministic transitions.)
73         """
74         assert not (self.is_terminal_state(state))
75         assert action in self.get_available_actions(state)
76         pass
77
78     @abstractmethod
79     def is_terminal_state(self, state: State) → bool:
80         """
81         Input:
82             state: a GameState
83         Output:
84             Returns a boolean indicating whether or not the given
85             state is terminal.
86         """
87         pass
88
89     # Used to be called evaluate_state
90     @abstractmethod
91     def evaluate_terminal(self, state: State) → Tuple[int, int]:
92         """
93         Should be called when determining which player benefits from a given
94         *terminal* state.
95         The range of values returned here should be synchronized with
96         heuristic_func.

```

```

96         Because we're evaluating terminal states, we're essentially evaluating
losing, winning, and
97         tying. You should make sure that the sum of the tuple you return sums
to a constant number,
98         like 1. If player 0 wins, then should their score be high or low
relative to player 1?
99
100        Final note: evaluate_terminal and heuristic_func do very similar
things. In fact, their
101        ranges are the same! However, we split these up because heuristic_func
should be used in
102        only the algorithm that uses a heuristic, whereas evaluate_terminal is
used across all
103        algorithms, since they all need to know how good or bad a terminal
state is.
104
105        Input:
106            state: a TERMINAL GameState
107        Output:
108            Returns a Tuple of player 0's value and player 1's value,
where each value
109            represents whether the player lost, tied, or won.
110        """
111        assert self.is_terminal_state(state)
112        pass
113
114
115    def HeuristicAdversarialSearchProblem(AdversarialSearchProblem):
116        @abstractmethod
117        def heuristic(self, state: State) → float:
118            """
119            Input:
120                state: The current game state.
121            Output:
122                Returns a heuristic evaluation of state (a float)
123            """
124            pass
125
126
127    #####
128    # GameUI is an abstraction that allows you to interact directly with
129    # an AdversarialSearchProblem (through gamerunner.py). See ttproblem or
130    # connect4problem for examples.
131    #
132    # Utilizing GameUI is NOT necessary for this assignment, although you can use
133    # it with any ASPs you may decide to create.
134    #####
135
136    class GameUI(ABC):
137        def update_state(self, state: GameState):
138            """

```

```

139         Updates the state currently being rendered.
140         """
141         self._state = state
142
143     @abstractmethod
144     def render(self):
145         """
146         Renders the GameUI instance's render (presumably this will be called
continuously).
147         """
148         pass
149
150     @abstractmethod
151     def get_user_input_action(self):
152         """
153         Output- Returns an action obtained through the GameUI input itself.
154         (It is expected that GameUI validates that the action is valid).
155         """
156         pass
157
158
159 from go_search_problem import GoProblem, GoState, Action
160 from adversarial_search_problem import GameState
161 from heuristic_go_problems import *
162 import random
163 from abc import ABC, abstractmethod
164 import numpy as np
165 import time
166 from game_runner import run_many
167 import pickle
168 import torch
169 from torch import nn
170 import matplotlib.pyplot as plt
171 import math
172
173 def run_many_wrapper(agent1, agent2, num_games):
174     # Import `run_many` inside the wrapper function to avoid circular import
175     from game_runner import run_many
176
177     agent1_score, agent2_score = run_many(agent1, agent2, num_games)
178     print(agent1_score, agent2_score)
179
180     return agent1_score, agent2_score
181
182
183 MAXIMIZER = 0
184 MINIMIZER = 1
185
186 class GameAgent():
187     # Interface for Game agents
188     @abstractmethod

```

```

189     def get_move(self, game_state: GameState, time_limit: float) → Action:
190         # Given a state and time limit, return an action
191         pass
192
193
194     class RandomAgent(GameAgent):
195         # An Agent that makes random moves
196
197         def __init__(self):
198             self.search_problem = GoProblem()
199
200         def get_move(self, game_state: GoState, time_limit: float) → Action:
201             """
202             get random move for a given state
203             """
204             actions = self.search_problem.get_available_actions(game_state)
205             return random.choice(actions)
206
207         def __str__(self):
208             return "RandomAgent"
209
210
211     class GreedyAgent(GameAgent):
212         def __init__(self, search_problem=GoProblemSimpleHeuristic()):
213             super().__init__()
214             self.search_problem = search_problem
215
216         def get_move(self, game_state: GoState, time_limit: float) → Action:
217             """
218             get move of agent for given game state.
219             Greedy agent looks one step ahead with the provided heuristic and
220             chooses the best available action
221             (Greedy agent does not consider remaining time)
222
223             Args:
224                 game_state (GameState): current game state
225                 time_limit (float): time limit for agent to return a move
226             """
227             # Create new GoSearchProblem with provided heuristic
228             search_problem = self.search_problem
229
230             # Player 0 is maximizing
231             if game_state.player_to_move() == MAXIMIZER:
232                 best_value = -float('inf')
233             else:
234                 best_value = float('inf')
235
236             best_action = None
237
238             # Get Available actions
239             actions = search_problem.get_available_actions(game_state)

```

```

239         # Compare heuristic of every reachable next state
240         for action in actions:
241             new_state = search_problem.transition(game_state, action)
242             value = search_problem.heuristic(new_state,
new_state.player_to_move())
243             if game_state.player_to_move() == MAXIMIZER:
244                 if value > best_value:
245                     best_value = value
246                     best_action = action
247             else:
248                 if value < best_value:
249                     best_value = value
250                     best_action = action
251
252         # Return best available action
253         return best_action
254
255     def __str__(self):
256         """
257         Description of agent (Greedy + heuristic/search problem used)
258         """
259         return "GreedyAgent + " + str(self.search_problem)
260
261
262     class MinimaxAgent(GameAgent):
263         def __init__(self, depth=1, search_problem=GoProblemSimpleHeuristic()):
264             super().__init__()
265             self.depth = depth
266             self.search_problem = search_problem
267
268         def get_move(self, game_state: GoState, time_limit: float) → Action:
269             """
270             Get move of agent for given game state using minimax algorithm
271
272             Args:
273                 game_state (GameState): current game state
274                 time_limit (float): time limit for agent to return a move
275             Returns:
276                 best_action (Action): best action for current game state
277             """
278             # TODO: implement get_move method of MinimaxAgent
279             states_expanded = 0
280
281             def max_value(depth, state):
282                 """
283                 Helper function for minimax. Computes the optimal action for the
maximizer.
284
285                 Input:
286                     depth - the current depth in the search tree
287                     state - the current state being evaluated
288
289                 Output: tuple containing,

```

```

288         - the maximum value that can be achieved from this state
289         - the corresponding action that leads to this value
290     """
291     nonlocal states_expanded
292     states_expanded += 1
293
294     if self.search_problem.is_terminal_state(state): # check if the
current state is terminal
295         reward = self.search_problem.evaluate_terminal(state)
296         if reward == 1:
297             return float('inf'), None
298         elif reward == -1:
299             return float('-inf'), None
300         else:
301             return 0, None # Tie
302
303     if depth == self.depth:
304         return self.search_problem.heuristic(state,
state.player_to_move()), None
305
306     max_eval = float('-inf')
307     actions = self.search_problem.get_available_actions(state) # get
available actions from the state
308     best_action = actions[0]
309
310     for action in actions: # check every available action
311         next_state = self.search_problem.transition(state, action)
312         curr_eval, _ = min_value(depth + 1, next_state) # calls
min_value to predict opponent's action for the next_state
313
314         if curr_eval > max_eval: # update max_eval and best_action if
a better value is found
315             max_eval = curr_eval
316             best_action = action
317
318     return max_eval, best_action
319
320
321     def min_value(depth, state):
322         """
323         Helper function for minimax. Computes the optimal action for the
minimizer.
324
325         Input:
326             depth - the current depth in the search tree
327             state - the current state being evaluated
328         Output: tuple containing,
329             - the minimum value that can be achieved from this state
330             - the corresponding action that leads to this value
331         """
332         # Similar logic to the one for max_value in minimax.
nonlocal states_expanded

```

```

333         states_expanded += 1
334
335         if self.search_problem.is_terminal_state(state): # check if the
current state is terminal
336             reward = self.search_problem.evaluate_terminal(state)
337             if reward == 1:
338                 return float('inf'), None
339             elif reward == -1:
340                 return float('-inf'), None
341             else:
342                 return 0, None # Tie
343
344         if depth == self.depth:
345             return self.search_problem.heuristic(state,
state.player_to_move()), None
346
347         min_eval = float('inf')
348         actions = self.search_problem.get_available_actions(state)
349         best_action = actions[0]
350
351         for action in actions:
352             next_state = self.search_problem.transition(state, action)
353             curr_eval, _ = max_value(depth + 1, next_state)
354
355             if curr_eval < min_eval:
356                 min_eval = curr_eval
357                 best_action = action
358
359         return min_eval, best_action
360
361         #####LOGIC FOR MINIMAX
362         player = game_state.player_to_move()
363
364         if player == 0: # maximizer plays
365             _, best_action = max_value(0, game_state)
366         else: # minimizer plays
367             _, best_action = min_value(0, game_state)
368
369         # after the first call to max_value/min_value, the helpers call each
other recursively until reaching the final_state
370
371         stats = {'states_expanded': states_expanded}
372
373         if best_action is None: # Check if no valid action was found
374             raise ValueError("No valid action found by the agent.")
375
376         return best_action
377
378     def __str__(self):
379         return f"MinimaxAgent w/ depth {self.depth} + " +
str(self.search_problem)

```



```

380
381
382 class AlphaBetaAgent(GameAgent):
383     def __init__(self, depth=1, search_problem=GoProblemSimpleHeuristic()):
384         super().__init__()
385         self.depth = depth
386         self.search_problem = search_problem
387
388     def get_move(self, game_state: GoState, time_limit: float) → Action:
389         """
390         Get move of agent for given game state using alpha-beta algorithm
391
392         Args:
393             game_state (GameState): current game state
394             time_limit (float): time limit for agent to return a move
395         Returns:
396             best_action (Action): best action for current game state
397         """
398         # TODO: implement get_move algorithm of AlphaBeta Agent
399         if self.search_problem.is_terminal_state(game_state):
400             print("Terminal state reached!")
401             return None
402
403         def max_value(depth, state, alpha, beta):
404             """
405             Helper function for alpha-beta pruning. Computes optimal action
406             for the maximizer.
407             Input:
408                 depth - the current depth in the search tree
409                 state - the current state being evaluated
410                 alpha - the best value that the maximizer can guarantee so far
411                 beta - the best value that the minimizer can guarantee so far
412             Output:
413                 Returns a tuple containing:
414                 - The maximum value that can be achieved from this state
415                 - The corresponding action that leads to this value
416             """
417             if self.search_problem.is_terminal_state(state): # check if the
current state is terminal
418                 reward = self.search_problem.evaluate_terminal(state)
419                 if reward == 1:
420                     return float('inf'), None
421                 elif reward == -1:
422                     return float('-inf'), None
423                 else:
424                     return 0, None # Tie
425
426             if depth == self.depth:
427                 return self.search_problem.heuristic(state,
state.player_to_move()), None

```

```

428
429         max_eval = float('-inf')
430         actions = self.search_problem.get_available_actions(state)
431         best_action = actions[0]
432
433         np.random.shuffle(actions)
434
435         for action in actions:
436             next_state = self.search_problem.transition(state, action)
437             curr_eval, _ = min_value(depth + 1, next_state, alpha, beta)
438
439             if curr_eval > max_eval: # update max_eval, best_action, and
alpha if a better value is found
440                 max_eval = curr_eval
441                 best_action = action
442
443                 alpha = max(alpha, max_eval)
444
445                 if beta ≤ alpha: # stop exploring if beta ≤ alpha
446                     return max_eval, best_action
447
448         return max_eval, best_action
449
450     def min_value(depth, state, alpha, beta):
451         """
452         Helper function for alpha-beta pruning. Computes optimal action
453         for the minimizer.
454         Input:
455             depth - the current depth in the search tree
456             state - the current state being evaluated
457             alpha - the best value that the maximizer can guarantee so far
458             beta - the best value that the minimizer can guarantee so far
459
460         Output:
461             Returns a tuple containing:
462             - The minimum value that can be achieved from this state
463             - The corresponding action that leads to this value
464         """
465         if self.search_problem.is_terminal_state(state): # check if the
current state is terminal
466             reward = self.search_problem.evaluate_terminal(state)
467             if reward == 1:
468                 return float('inf'), None
469             elif reward == -1:
470                 return float('-inf'), None
471             else:
472                 return 0, None # Tie
473
474         if depth == self.depth:

```

```

475         return self.search_problem.heuristic(state,
state.player_to_move()), None
476
477         min_eval = float('inf')
478         actions = self.search_problem.get_available_actions(state)
479         best_action = actions[0]
480
481         np.random.shuffle(actions)
482
483         for action in actions:
484             next_state = self.search_problem.transition(state, action)
485             curr_eval, _ = max_value(depth + 1, next_state, alpha, beta)
486
487             if curr_eval < min_eval:
488                 min_eval = curr_eval
489                 best_action = action
490
491             beta = min(beta, min_eval)
492
493             if beta ≤ alpha:
494                 return min_eval, best_action
495
496         return min_eval, best_action
497
498         #####
499         alpha = float('-inf')
500         beta = float('inf')
501
502         player = game_state.player_to_move()
503
504         if player == 0:
505             _, best_action = max_value(0, game_state, alpha, beta)
506         else:
507             _, best_action = min_value(0, game_state, alpha, beta)
508
509         return best_action
510
511         def __str__(self):
512             return f"AlphaBeta w/ depth {self.depth} + " +
str(self.search_problem)
513
514
515         class IterativeDeepeningAgent(GameAgent):
516             def __init__(self, cutoff_time=1,
search_problem=GoProblemSimpleHeuristic()):
517                 super().__init__()
518                 self.cutoff_time = cutoff_time
519                 self.search_problem = search_problem
520
521             def get_move(self, game_state: GoState, time_limit: float):
522                 """

```

```

523     Get move of agent for given game state using iterative deepening
algorithm (+ alpha-beta).
524     Iterative deepening is a search algorithm that repeatedly searches for
a solution to a problem,
525     increasing the depth of the search with each iteration.
526
527     The advantage of iterative deepening is that you can stop the search
based on the time limit, rather than depth.
528     The recommended approach is to modify your implementation of Alpha-
beta to stop when the time limit is reached
529     and run IDS on that modified version.
530
531     Args:
532         game_state (GameState): current game state
533         time_limit (float): time limit for agent to return a move
534     Returns:
535         best_action (Action): best action for current game state
536     """
537     if self.search_problem.is_terminal_state(game_state):
538         return self.search_problem.get_available_actions(game_state)[0]
539
540     start_time = time.time()
541     time_buffer = 0.05 # Prevent exceeding time limit
542     end_time = start_time + time_limit - time_buffer
543
544     best_action = None
545     current_depth = 1
546
547     def max_value(depth, state, alpha, beta):
548         if time.time() ≥ end_time:
549             raise TimeoutError
550
551         if self.search_problem.is_terminal_state(state):
552             reward = self.search_problem.evaluate_terminal(state)
553             if reward == 1:
554                 return float('inf'), None
555             elif reward == -1:
556                 return float('-inf'), None
557             return 0, None
558
559         if depth ≤ 0:
560             return self.search_problem.heuristic(state,
state.player_to_move()), None
561
562         max_eval = float('-inf')
563         actions = self.search_problem.get_available_actions(state)
564         if not actions:
565             return max_eval, None
566
567         best_action = actions[0]
568         np.random.shuffle(actions)

```

```

569
570     for action in actions:
571         next_state = self.search_problem.transition(state, action)
572         curr_eval, _ = min_value(depth - 1, next_state, alpha, beta)
573
574         if curr_eval > max_eval:
575             max_eval = curr_eval
576             best_action = action
577
578         alpha = max(alpha, max_eval)
579         if beta ≤ alpha:
580             break
581
582     return max_eval, best_action
583
584 def min_value(depth, state, alpha, beta):
585     if time.time() ≥ end_time:
586         raise TimeoutError
587
588     if self.search_problem.is_terminal_state(state):
589         reward = self.search_problem.evaluate_terminal(state)
590         if reward == 1:
591             return float('inf'), None
592         elif reward == -1:
593             return float('-inf'), None
594         return 0, None
595
596     if depth ≤ 0:
597         return self.search_problem.heuristic(state,
state.player_to_move()), None
598
599     min_eval = float('inf')
600     actions = self.search_problem.get_available_actions(state)
601     if not actions:
602         return min_eval, None
603
604     best_action = actions[0]
605     np.random.shuffle(actions)
606
607     for action in actions:
608         next_state = self.search_problem.transition(state, action)
609         curr_eval, _ = max_value(depth - 1, next_state, alpha, beta)
610
611         if curr_eval < min_eval:
612             min_eval = curr_eval
613             best_action = action
614
615         beta = min(beta, min_eval)
616         if beta ≤ alpha:
617             break
618

```

```

619         return min_eval, best_action
620
621     actions = self.search_problem.get_available_actions(game_state)
622     if actions:
623         best_action = actions[0]
624     else:
625         return None
626
627     # Main IDS loop
628     while time.time() < end_time:
629         try:
630             alpha = float('-inf')
631             beta = float('inf')
632
633             if game_state.player_to_move() == 0: # MAX player
634                 _, current_action = max_value(current_depth, game_state,
alpha, beta)
635             else: # MIN player
636                 _, current_action = min_value(current_depth, game_state,
alpha, beta)
637
638             if current_action is not None:
639                 best_action = current_action
640
641                 current_depth += 1
642
643             except TimeoutError:
644                 break
645
646         return best_action
647
648     def __str__(self):
649         return f"IterativeDeepening + " + str(self.search_problem)
650
651
652
653     def load_dataset(path: str):
654         with open(path, 'rb') as f:
655             dataset = pickle.load(f)
656         return dataset
657
658     dataset_5x5 = load_dataset('dataset_5x5.pkl')
659     # dataset_9x9 = load_dataset('9x9_dataset.pkl')
660
661     def save_model(path: str, model, input_size=None):
662         """
663         Save model to a file
664         Input:
665             path: path to save model to
666             model: Pytorch model to save
667         """

```

```

668     torch.save({
669         'model_state_dict': model.state_dict(),
670     }, path)
671
672
673 def load_model(path: str, model):
674     """
675     Load model from file
676
677     Note: you still need to provide a model (with the same architecture as the
        saved model))
678
679     Input:
680         path: path to load model from
681         model: Pytorch model to load
682     Output:
683         model: Pytorch model loaded from file
684     """
685     checkpoint = torch.load(path)
686     model.load_state_dict(checkpoint['model_state_dict'])
687     return model
688
689 class ValueNetwork(nn.Module):
690     def __init__(self, input_size):
691         super(ValueNetwork, self).__init__()
692
693         # TODO: What should the output size of a Value function be?
694
695         ''' Handout: the goal is to classify each state as a future
696         win for one player or the other, or more generally, to
697         generate a prediction in the range [-1, +1] that is indicative
698         of which player will win the game.'''
699
700         output_size = 1
701
702         # TODO: Add more layers, non-linear functions, etc.
703
704         # Layers
705         self.fc1 = nn.Linear(input_size, 32)
706         self.fc2 = nn.Linear(32, 16)
707         self.fc3 = nn.Linear(16, output_size)
708
709         # Activation functions
710         self.relu = nn.ReLU()
711         self.tanh = nn.Tanh()
712
713     def forward(self, x):
714         """
715         Run forward pass of network
716
717         Input:

```

```

718         x: input to network
719         Output:
720         output of network
721         """
722         # TODO: Update as more layers are added
723         z1 = self.fc1(x)
724         a1 = self.relu(z1)
725
726         z2 = self.fc2(a1)
727         a2 = self.relu(z2)
728
729         z3 = self.fc3(a2)
730         a3 = self.relu(z3)
731
732         return a3
733
734
735 class GoProblemLearnedHeuristic(GoProblem):
736     def __init__(self, model=None, state=None):
737         super().__init__(state=state)
738         self.model = model
739
740     def __call__(self, model=None):
741         """
742         Use the model to compute a heuristic value for a given state.
743         """
744         return self
745
746     def encoding(self, state):
747         """
748         Get encoding of state (convert state to features)
749         Note, this may call get_features() from Task 1.
750
751         Input:
752             state: GoState to encode into a fixed size list of features
753         Output:
754             features: list of features
755         """
756         # TODO: get encoding of state (convert state to features)
757
758         return get_features(state)
759
760     def heuristic(self, state, player_index):
761         """
762         Return heuristic (value) of current state
763
764         Input:
765             state: GoState to encode into a fixed size list of features
766             player_index: index of player to evaluate heuristic for
767         Output:
768             value: heuristic (value) of current state

```



```

769     """
770     # TODO: Compute heuristic (value) of current state
771     value = 0
772
773     features = self.encoding(state)
774     features_tensor = torch.tensor(features, dtype=torch.float32)
775
776     with torch.no_grad():
777         value = self.model(features_tensor)
778
779     '''value = max(-1, min(1, value))
780     if player_index != state.player_to_move():
781         value = -value'''
782
783     # Note, your agent may perform better if you force it not to pass
784     # (i.e., don't select action #25 on a 5x5 board unless necessary)
785     return value
786
787     def __str__(self) → str:
788         return "Learned Heuristic"
789
790     import go_utils
791 def create_value_agent_from_model():
792     """
793     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.
794     """
795
796     model_path = "value_model.pt"
797     # TODO: Update number of features for your own encoding size
798
799     feature_size = len(get_features(dataset_5x5[0][0]))
800
801     model = load_model(model_path, ValueNetwork(feature_size))
802
803     heuristic_search_problem = GoProblemLearnedHeuristic(model)
804
805     # TODO: Try with other heuristic agents (IDS/AB/Minimax)
806     learned_agent = GreedyAgent(heuristic_search_problem)
807
808     return learned_agent
809
810
811 def get_features(game_state: GoState):
812     """
813     Map a game state to a list of features.
814
815     Some useful functions from game_state include:
816     game_state.size: size of the board

```

```

817         get_pieces_coordinates(player_index): get coordinates of all pieces of
a player (0 or 1)
818         get_pieces_array(player_index): get a 2D array of pieces of a player
(0 or 1)
819
820         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
821
822         Descriptions of these methods can be found in the GoState
823
824     Input:
825         game_state: GoState to encode into a fixed size list of features
826     Output:
827         features: list of features
828     """
829     board_size = game_state.size
830
831     # TODO: Encode game_state into a list of features
832     features = []
833
834     board = game_state.get_board()
835
836     for channel in range(4):
837         for row in range(board_size):
838             for col in range(board_size):
839                 features.append(board[channel][row][col])
840
841     return features
842
843
844 class PolicyNetwork(nn.Module):
845     def __init__(self, input_size, board_size=5):
846         super(PolicyNetwork, self).__init__()
847
848         # TODO: What should the output size of the Policy be?
849         self.output_size = board_size * board_size + 1
850
851         # TODO: Add more layers, non-linear functions, etc.
852         self.fc1 = nn.Linear(input_size, 512)
853         self.fc2 = nn.Linear(512, 128)
854         self.fc3 = nn.Linear(128, 64)
855         self.fc4 = nn.Linear(64, self.output_size)
856
857         self.relu = nn.ReLU()
858
859     def forward(self, x):
860         # TODO: Update as more layers are added
861         z1 = self.fc1(x)
862         a1 = self.relu(z1)
863         z2 = self.fc2(a1)

```

```

864         a2 = self.relu(z2)
865         z3 = self.fc3(a2)
866         a3 = self.relu(z3)
867         z4 = self.fc4(a3)
868
869         return z4
870
871 class PolicyAgent(GameAgent):
872     def __init__(self, search_problem, model_path, board_size=5):
873         super().__init__()
874         self.search_problem = search_problem
875         # self.model = load_model(model_path, PolicyNetwork)
876
877         input_size = len(get_features(dataset_5x5[0][0]))
878         model_template = PolicyNetwork(input_size, board_size)
879         self.model = load_model(model_path, model_template)
880
881         self.board_size = board_size
882
883     def encoding(self, state):
884         # TODO: get encoding of state (convert state to features)
885         return get_features(state)
886
887     def get_move(self, game_state: GoState, time_limit=1):
888         """
889         Get best action for current state using self.model
890
891         Input:
892             game_state: current state of the game
893             time_limit: time limit for search (This won't be used in this agent)
894         Output:
895             action: best action to take
896         """
897         legal_actions = self.search_problem.get_available_actions(game_state)
898
899         features = self.encoding(game_state)
900         features_tensor = torch.tensor(features,
dtype=torch.float32).unsqueeze(0)
901
902         with torch.no_grad():
903             action_logits = self.model(features_tensor)
904             action_probs = torch.softmax(action_logits, dim=1).squeeze(0)
905
906         all_probs = action_probs.tolist()
907
908         # Get probabilities for legal actions
909         legal_actions_probs = [
910             (action, all_probs[action-1] if 1 ≤ action ≤ len(all_probs) else
0)
911             for action in legal_actions
912         ]

```

```

913         # Sort legal actions by probability
914         sorted_legal_actions = sorted(legal_actions_probs, key=lambda x: x[1],
915 reverse=True)
916
917         # Return best legal action
918         return sorted_legal_actions[0][0] if sorted_legal_actions else None
919
920     def __str__(self) → str:
921         return "Policy Agent"
922
923 def create_policy_agent_from_model():
924     """
925     Create agent object from saved model. This (or other methods like this)
926     will be how your agents will be created in gradescope and in the final
927     tournament.
928     """
929     model_path = "policy_model.pt"
930     agent = PolicyAgent(GoProblem(size=5), model_path)
931     return agent
932
933
934 def plot_agent_comparisons(learned_agent_name, learned_agent):
935     """
936     Create a bar plot comparing the performance of learned agents against
937     other agents
938
939     :param value_agent: Learned value network agent
940     :param policy_agent: Learned policy network agent
941     """
942     random_agent = RandomAgent()
943     greedy_agent = GreedyAgent()
944     minimax_agent = MinimaxAgent()
945     alpha_beta_agent = AlphaBetaAgent()
946     iterative_deepening_agent = IterativeDeepeningAgent()
947     mcts_agent = MCTSAgent()
948
949     agents = [
950         ("Random", random_agent),
951         ("Greedy", greedy_agent),
952         ("Minimax", minimax_agent),
953         ("AlphaBeta", alpha_beta_agent),
954         ("IterativeDeepening", iterative_deepening_agent),
955         ("MCTS", mcts_agent)
956     ]
957
958     num_games = 5
959
960     value_scores = []

```

```

960     policy_scores = []
961     agent_names = []
962
963     for agent_name, agent in agents:
964         agent1_score, agent2_score = run_many_wapper(agent, learned_agent,
965             num_games)
966         agent_names.append(f"{learned_agent_name} vs {agent_name}")
967         value_scores.append(agent1_score)
968         policy_scores.append(agent2_score)
969
970     plt.figure(figsize=(12, 6))
971
972     bar_width = 0.35
973
974     r1 = np.arange(len(agents))
975     r2 = [x + bar_width for x in r1]
976
977     plt.bar(r1, value_scores, color='skyblue', width=bar_width,
978         label=learned_agent_name)
979     plt.bar(r2, policy_scores, color='lightgreen', width=bar_width,
980         label=f'Opponent {learned_agent_name}')
981
982     plt.xlabel('Opponent Agents')
983     plt.ylabel('Score')
984     plt.title(f'Performance of {learned_agent_name} Against Different
985         Opponents')
986     plt.xticks([r + bar_width/2 for r in range(len(agents))], agent_names,
987         rotation=45)
988
989     plt.legend()
990
991     for i, (v1, v2) in enumerate(zip(value_scores, policy_scores)):
992         plt.text(r1[i], v1, f'{v1:.2f}', ha='center', va='bottom')
993         plt.text(r2[i], v2, f'{v2:.2f}', ha='center', va='bottom')
994
995     plt.tight_layout()
996     plt.savefig(f'agent_comparison_{learned_agent_name}.png')
997     plt.close()
998
999
1000 class MCTSNode:
1001     def __init__(self, state, parent=None, action=None,
1002         prior_probability=0.0):
1003         self.state = state
1004         self.parent = parent
1005         self.children = []
1006         self.visits = 0
1007         self.value = 0
1008         self.action = action
1009         self.prior_probability = prior_probability

```

```

1005
1006 def is_leaf(self):
1007     """
1008     Checks if the node is a leaf (i.e., has no children).
1009     """
1010     return (len(self.children) == 0 or
1011             self.state.is_terminal_state() or
1012             len(self.children) < len(self.state.legal_actions()))
1013
1014 def __hash__(self):
1015     return hash(self.state)
1016
1017
1018 class MCTSAgent(GameAgent):
1019     def __init__(self, c=np.sqrt(2)):
1020         super().__init__()
1021         self.c = c
1022         self.search_problem = GoProblem()
1023         self.action_choices = []
1024
1025     def get_move(self, game_state: GoState, time_limit: float) → Action:
1026         root = MCTSNode(game_state)
1027         start_time = time.time()
1028
1029         # While time remains
1030         while time.time() - start_time < 0.9:
1031             leaf = self.select(root)
1032
1033             # Only expand if not terminal
1034             if not self.search_problem.is_terminal_state(leaf.state):
1035                 self.expand(leaf)
1036
1037             # Simulate and backpropagate for each child
1038             for child in leaf.children:
1039                 result = self.simulate(child)
1040                 self.backprop(result, child)
1041
1042         # Return action with most visits
1043         return max(root.children, key=lambda child: child.visits).action
1044
1045     def select(self, node):
1046         """SELECT: Find a leaf node using UCT policy."""
1047         while not node.is_leaf():
1048             node = max(node.children, key=lambda child: self.uct_value(child))
1049         return node
1050
1051     def expand(self, leaf):
1052         """EXPAND: Create all possible child nodes."""
1053         actions = leaf.state.legal_actions()
1054
1055         # Create a child node for each legal action

```

```

1056         for action in actions:
1057             child_state = self.search_problem.transition(leaf.state, action)
1058             child_node = MCTSNode(state=child_state, parent=leaf,
action=action)
1059             leaf.children.append(child_node)
1060
1061     def simulate(self, node):
1062         """SIMULATE: Run rollout from given node."""
1063         curr_state = node.state
1064
1065         while not self.search_problem.is_terminal_state(curr_state):
1066             actions = curr_state.legal_actions()
1067             action = np.random.choice(actions)
1068             curr_state = self.search_problem.transition(curr_state, action)
1069             self.action_choices.append(action)
1070
1071         return self.search_problem.evaluate_terminal(curr_state)
1072
1073     def backprop(self, result, node):
1074         """BACKPROPAGATE: Update statistics from leaf to root."""
1075         while node is not None:
1076             node.visits += 1
1077
1078             if result < 0 and node.state.player_to_move() == 0:
1079                 node.value += 1
1080             elif result > 0 and node.state.player_to_move() == 1:
1081                 node.value += 1
1082
1083             node = node.parent
1084
1085     def uct_value(self, node):
1086         """Calculate UCT value for node selection."""
1087         if node.visits == 0:
1088             return float('inf')
1089
1090         exploitation = node.value / node.visits
1091         exploration = (self.c * np.sqrt(np.log(node.parent.visits) /
node.visits) if node.parent else 0)
1092
1093         return exploitation + exploration
1094
1095     def plot_action_frequencies(self, actions):
1096         action_counts = {action: actions.count(action) for action in
set(actions)}
1097         actions = list(action_counts.keys())
1098         frequencies = list(action_counts.values())
1099
1100         plt.figure(figsize=(8, 6))
1101         plt.bar(actions, frequencies)
1102         plt.xlabel("Action")
1103         plt.ylabel("Frequency")

```

```

1104     plt.title("Action Frequencies in Rollouts")
1105     plt.savefig('action_frequencies_plot.png')
1106     plt.close()
1107
1108     def __str__(self):
1109         return "MCTS"
1110
1111
1112 class NeuralMCTSAgent(GameAgent):
1113     def __init__(self, policy_network, value_network, c=np.sqrt(2)):
1114         super().__init__()
1115         self.c = c
1116         self.policy_network = policy_network
1117         self.value_network = value_network
1118         self.search_problem = GoProblem()
1119         self.action_choices = []
1120
1121     def get_move(self, game_state: GoState, time_limit: float) → Action:
1122         root = MCTSNode(game_state)
1123         start_time = time.time()
1124
1125         # While time remains
1126         while time.time() - start_time < 0.9:
1127             leaf = self.select(root)
1128
1129             # Only expand if not terminal
1130             if not self.search_problem.is_terminal_state(leaf.state):
1131                 self.expand(leaf)
1132
1133             # Simulate and backpropagate for each child
1134             for child in leaf.children:
1135                 result = self.simulate(child)
1136                 self.backprop(result, child)
1137
1138             # Return action with most visits
1139             return max(root.children, key=lambda child: child.visits).action
1140
1141     def select(self, node):
1142         """SELECT: Find a leaf node using PUCT policy."""
1143         while not node.is_leaf():
1144             node = max(node.children, key=lambda child:
self.puct_value(child))
1145         return node
1146
1147     def expand(self, leaf):
1148         """EXPAND: Create all possible child nodes, guided by the policy
network."""
1149         actions = leaf.state.legal_actions()
1150
1151         # Get the policy distribution from the policy network for the current
state

```



```

1152     features = get_features(leaf.state) # Get features from the state
1153     features_tensor = torch.tensor(features, dtype=torch.float32)
1154
1155     with torch.no_grad():
1156         # Get policy logits and convert them to probabilities
1157         policy_logits = self.policy_network(features_tensor)
1158         policy_probs = torch.softmax(policy_logits, dim=0).tolist()
1159
1160     # Filter out illegal actions
1161     legal_actions_probs = {action: policy_probs[action - 1] for action in
actions}
1162
1163     # Sort actions based on their probability from the policy network
1164     sorted_actions = sorted(legal_actions_probs.items(), key=lambda x:
x[1], reverse=True)
1165
1166     # Create a child node for each legal action, prioritizing higher
probability actions
1167     for action, _ in sorted_actions:
1168         child_state = self.search_problem.transition(leaf.state, action)
1169         prior_probability = policy_probs[action - 1]
1170         child_node = MCTSNode(state=child_state, parent=leaf,
action=action, prior_probability=prior_probability)
1171         leaf.children.append(child_node)
1172
1173     def simulate(self, node):
1174         """SIMULATE: Use the value network to simulate the outcome from the
given node."""
1175         features = get_features(node.state)
1176         features_tensor = torch.tensor(features, dtype=torch.float32)
1177
1178         # Use the value network to evaluate the state
1179         with torch.no_grad():
1180             value = self.value_network(features_tensor).item()
1181
1182         return value
1183
1184     def backprop(self, result, node):
1185         """BACKPROPAGATE: Update statistics from leaf to root."""
1186         while node is not None:
1187             node.visits += 1
1188
1189             if result < 0 and node.state.player_to_move() == 0:
1190                 node.value += 1
1191             elif result > 0 and node.state.player_to_move() == 1:
1192                 node.value += 1
1193
1194             node = node.parent
1195
1196     def uct_value(self, node):
1197         """Calculate UCT value for node selection."""

```

```

1198         if node.visits == 0:
1199             return float('inf')
1200
1201         exploitation = node.value / node.visits
1202         exploration = (self.c * np.sqrt(np.log(node.parent.visits) /
node.visits) if node.parent else 0)
1203
1204         return exploitation + exploration
1205
1206
1207     def puct_value(self, node):
1208         """Calculate PUCT value for node selection."""
1209         if node.visits == 0:
1210             return float('inf')
1211
1212         exploitation = node.value / node.visits
1213
1214         '''
1215         UCT implementation:
1216         exploration = (self.c * np.sqrt(np.log(node.parent.visits) /
node.visits) if node.parent else 0)
1217         '''
1218         exploration = self.c * node.prior_probability *
np.sqrt(np.log(node.parent.visits) / node.visits) if node.parent else 0
1219
1220         return exploitation + exploration
1221
1222     def __str__(self):
1223         return "Neural MCTS"
1224
1225
1226
1227 class OpeningBook:
1228     def __init__(self):
1229         self.openings_5x5 = {
1230             'empty_board': [
1231                 (2, 2),      # Center
1232                 (1, 1),
1233                 (1, 3),
1234                 (3, 1),
1235                 (3, 3)
1236             ],
1237             'center_taken': [
1238                 (0, 0),      # Corner
1239                 (0, 4),
1240                 (4, 0),
1241                 (4, 4),
1242                 (1, 0),      # Edge
1243                 (1, 4),
1244                 (3, 0),
1245                 (3, 4)

```

```

1246     ]
1247 }
1248
1249 self.openings_9x9 = {
1250     'empty_board': [
1251         (4, 4),      # Center
1252         (2, 2),      # Star points (4-4 points)
1253         (2, 6),
1254         (6, 2),
1255         (6, 6),
1256         (2, 4),      # Side star points
1257         (4, 2),
1258         (4, 6),
1259         (6, 4)
1260     ],
1261     'center_taken': [
1262         (1, 1),      # 3-3 points
1263         (1, 7),
1264         (7, 1),
1265         (7, 7),
1266         (4, 1),      # Side approaches
1267         (1, 4),
1268         (4, 7),
1269         (7, 4)
1270     ],
1271     'star_point_taken': [
1272         (3, 3),      # 5-5 points
1273         (3, 5),
1274         (5, 3),
1275         (5, 5),
1276         (0, 0),      # Corner moves
1277         (0, 8),
1278         (8, 0),
1279         (8, 8)
1280     ]
1281 }
1282
1283 def get_opening_move(self, game_state: GoState, time_limit: float):
1284     """
1285     Get an opening move based on predefined strategies for specific board
1286     states.
1287     """
1288     board = game_state.get_board()
1289     size = board.shape[1]
1290
1291     # Select appropriate opening book based on board size
1292     openings = self.openings_5x5 if size == 5 else self.openings_9x9
1293
1294     # Check if the board is empty
1295     if self.is_empty_board(board):
1296         for move in openings['empty_board']:

```

```

1296         if self.is_legal_move(board, move):
1297             return self.move_to_index(move, size)
1298
1299     # For 9x9 board, check if any star points are taken
1300     if size == 9 and not self.is_empty_board(board):
1301         star_points = [(2, 2), (2, 6), (6, 2), (6, 6)]
1302         if any(not self.is_empty_point(board, point) for point in
star_points):
1303             for move in openings['star_point_taken']:
1304                 if self.is_legal_move(board, move):
1305                     return self.move_to_index(move, size)
1306
1307     # If the center is taken, play from 'center_taken' strategies
1308     if not self.is_empty_board(board):
1309         for move in openings['center_taken']:
1310             if self.is_legal_move(board, move):
1311                 return self.move_to_index(move, size)
1312
1313     # No suitable opening move found
1314     return None
1315
1316     def is_empty_board(self, board: np.ndarray) → bool:
1317         """
1318         Check if the board is empty by verifying the first three channels.
1319         An empty board has no black or white pieces, and all cells in the
EMPTY channel are 1.
1320         """
1321         return np.all(board[0] == 0) and np.all(board[1] == 0) and
np.all(board[2] == 1)
1322
1323     def is_legal_move(self, board: np.ndarray, move: tuple) → bool:
1324         """
1325         Check if the move is legal (i.e., within bounds and on an empty cell).
1326         """
1327         x, y = move
1328         size = board.shape[1]
1329
1330         # Move within bounds and on an empty cell
1331         empty_board = (board[2])
1332         return 0 ≤ x < size and 0 ≤ y < size and empty_board[x][y] == 1
1333
1334     def move_to_index(self, move: tuple, size: int) → int:
1335         """Convert a 2D (row, col) move to a 1D index."""
1336         x, y = move
1337         return x * size + y
1338
1339
1340     class HybridGoAgent5x5(GameAgent):
1341         def __init__(self, board_size=5):
1342             super().__init__()
1343

```

```

1344         self.opening_book = OpeningBook()
1345
1346         # input_size = len(get_features(dataset_5x5[0][0]))
1347
1348         # policy_model = PolicyNetwork(input_size)
1349         # value_model = ValueNetwork(input_size)
1350         # self.mcts_agent = NeuralMCTSAgent(policy_network=policy_model,
value_network=value_model)
1351         self.mcts_agent = MCTSAgent()
1352
1353         self.alphabeta_agent = AlphaBetaAgent()
1354         self.ids_agent = IterativeDeepeningAgent(1,
GoProblemAdvancedHeuristic())
1355         self.move_count = 0
1356         self.total_moves = board_size * board_size
1357
1358     def get_move(self, state: GoState, time_limit: float) → Action:
1359         self.move_count += 1
1360         if self.move_count ≤ 3:
1361             book_move = self.opening_book.get_opening_move(state, time_limit)
1362             if book_move is not None:
1363                 return book_move
1364
1365         endgame_threshold = int(self.total_moves * 0.75)
1366         if self.move_count ≥ endgame_threshold:
1367             # return self.alphabeta_agent.get_move(state, time_limit)
1368             return self.ids_agent.get_move(state, time_limit)
1369
1370         return self.mcts_agent.get_move(state, time_limit)
1371
1372     def __str__(self):
1373         return "HybridGoAgent5x5"
1374
1375
1376
1377 class HybridGoAgent9x9(GameAgent):
1378     def __init__(self, board_size=9):
1379         super().__init__()
1380
1381         self.opening_book = OpeningBook()
1382
1383         self.mcts_agent = MCTSAgent()
1384
1385         self.alphabeta_agent = AlphaBetaAgent()
1386         self.ids_agent = IterativeDeepeningAgent(1,
GoProblemAdvancedHeuristic())
1387         self.move_count = 0
1388         self.total_moves = board_size * board_size
1389
1390     def get_move(self, state: GoState, time_limit: float) → Action:
1391         self.move_count += 1

```

```

1392         if self.move_count ≤ 4:
1393             book_move = self.opening_book.get_opening_move(state, time_limit)
1394             if book_move is not None:
1395                 return book_move
1396
1397         endgame_threshold = int(self.total_moves * 0.7)
1398         if self.move_count ≥ endgame_threshold:
1399             return self.ids_agent.get_move(state, time_limit)
1400
1401         return self.mcts_agent.get_move(state, time_limit)
1402
1403     def __str__(self):
1404         return "HybridGoAgent9×9"
1405
1406
1407     def get_final_agent_5×5():
1408         """Called to construct agent for final submission for 5×5 board"""
1409         return HybridGoAgent5×5()
1410
1411     def get_final_agent_9×9():
1412         """Called to construct agent for final submission for 9×9 board"""
1413         return HybridGoAgent9×9()
1414
1415
1416     def plot_compare_hybrid_agent(hybrid_agent : HybridGoAgent5×5):
1417         """
1418         Create a bar plot comparing the performance of HybridGoAgent5×5 against
1419         other agents
1420         """
1421         hybrid_agent_name = str(hybrid_agent)
1422
1423         agents = [
1424             ("IterativeDeepening",
1425              IterativeDeepeningAgent(GoProblemSimpleHeuristic)),
1426             ("MCTS", MCTSAgent()),
1427             ("Random", RandomAgent()),
1428             ("Greedy", GreedyAgent()),
1429             ("Minimax", MinimaxAgent()),
1430             ("AlphaBeta", AlphaBetaAgent())
1431         ]
1432
1433         num_games = 5
1434         hybrid_scores = []
1435         opponent_scores = []
1436         agent_names = []
1437
1438         for agent_name, opponent in agents:
1439             # Run games with hybrid agent as both first and second player
1440             hybrid_first, opp_first = run_many(hybrid_agent, opponent, num_games)
1441             opp_second, hybrid_second = run_many(opponent, hybrid_agent,
1442             num_games)

```

```

1440
1441     # Average scores from both positions
1442     hybrid_avg = (hybrid_first + hybrid_second) / 2
1443     opp_avg = (opp_first + opp_second) / 2
1444
1445     print(f"{hybrid_agent_name}: {hybrid_avg}, {agent_name} Score:
{opp_avg}")
1446
1447     agent_names.append(f"vs {agent_name}")
1448     hybrid_scores.append(hybrid_avg)
1449     opponent_scores.append(opp_avg)
1450
1451     plt.figure(figsize=(12, 6))
1452     bar_width = 0.35
1453
1454     r1 = np.arange(len(agent_names))
1455     r2 = [x + bar_width for x in r1]
1456
1457     plt.bar(r1, hybrid_scores, color='skyblue', width=bar_width,
label=hybrid_agent_name)
1458     plt.bar(r2, opponent_scores, color='lightgreen', width=bar_width,
label='Opponent')
1459
1460     plt.xlabel('Opponent Agents')
1461     plt.ylabel('Average Score')
1462     plt.title(f'Performance of {hybrid_agent_name} Against Different
Opponents')
1463     plt.xticks([r + bar_width/2 for r in range(len(agent_names))], agent_names,
rotation=45)
1464
1465     plt.legend()
1466
1467     for i, (h_score, o_score) in enumerate(zip(hybrid_scores,
opponent_scores)):
1468         plt.text(r1[i], h_score, f'{h_score:.2f}', ha='center', va='bottom')
1469         plt.text(r2[i], o_score, f'{o_score:.2f}', ha='center', va='bottom')
1470
1471     plt.tight_layout()
1472     plt.savefig(f'{hybrid_agent_name}_comparison.png')
1473     plt.close()
1474
1475
1476 def main():
1477     agent5x5 = HybridGoAgent5x5()
1478     agent9x9 = HybridGoAgent9x9()
1479
1480     plot_compare_hybrid_agent(agent5x5)
1481     # plot_compare_hybrid_agent(agent9x9)
1482
1483     ...
1484     go_agent5x5 = HybridGoAgent5x5()

```

```

1485     go_agent9x9 = HybridGoAgent9x9()
1486
1487     random_agent = RandomAgent()
1488     greedy_agent = GreedyAgent()
1489     minimax_agent = MinimaxAgent()
1490     alpha_beta_agent = AlphaBetaAgent()
1491     iterative_deepening_agent = IterativeDeepeningAgent()
1492     mcts_agent = MCTSAgent()
1493     policy_agent = create_policy_agent_from_model()
1494     value_agent = create_value_agent_from_model()
1495
1496     agents = [
1497         #("Random", random_agent),
1498         ("Greedy", greedy_agent),
1499         #("Minimax", minimax_agent),
1500         #("AlphaBeta", alpha_beta_agent),
1501         #("IterativeDeepening", iterative_deepening_agent),
1502         #("MCTS", mcts_agent),
1503         #("Policy", policy_agent),
1504         #("Value", value_agent)
1505     ]
1506
1507     num_games = 5
1508
1509     for agent_name, agent in agents:
1510         go_agent_score9x9, simple_agent_score = run_many(go_agent9x9, agent,
1511 num_games)
1512         print(f"{str(go_agent9x9)}: {go_agent_score9x9}, {agent_name} Score:
1513 {simple_agent_score}")
1514
1515         go_agent_score5x5, simple_agent_score = run_many(go_agent5x5, agent,
1516 num_games)
1517         print(f"{str(go_agent5x5)}: {go_agent_score5x5}, {agent_name} Score:
1518 {simple_agent_score}")
1519     '''
1520
1521 if __name__ == "__main__":
1522     main()
1523
1524 import time
1525 from go_search_problem import GoProblem
1526 import abc
1527 import tqdm
1528 import numpy as np
1529 from go_gui import GoGUI
1530 # from agents import *
1531 import pygame
1532 import argparse

```



```

1532 pygame.init()
1533 clock = pygame.time.Clock()
1534
1535 BLACK = MAXIMIZER = 0
1536 WHITE = MINIMIZER = 1
1537
1538
1539 def run_game(agent1, agent2, time_limit=15, time_increment=1,
hard_time_cutoff=True, size=5):
1540     """
1541     Run a single game between two agents.
1542     :param agent1: The first agent
1543     :param agent2: The second agent
1544     :param time_limit: The time limit for each player (starting time)
1545     :param time_increment: The time increment for each player (additional time
per move)
1546     :param hard_time_cutoff: If true, will terminate the game when a player
runs out of time
1547                             If false, will continue to play until the game
is over.
1548     :return: The result of the game (1 for agent1 win, -1 for agent2 win)
1549     """
1550     my_go = GoProblem(size=size)
1551     state = my_go.start_state
1552     player1_time = time_limit
1553     player2_time = time_limit
1554     player1_durations = []
1555     player2_durations = []
1556     while (not my_go.is_terminal_state(state)):
1557         start_time = time.time()
1558         # Clone so as to avoid side effects from agents
1559         player1_action = agent1.get_move(state.clone(), player1_time)
1560         move_duration = time.time() - start_time
1561         player1_time -= move_duration
1562         player1_durations.append(move_duration)
1563         if (player1_time ≤ 0):
1564             print("Player 1 over time")
1565             if hard_time_cutoff:
1566                 info = {"Agent 1 End Time": player1_time, "Agent 2 End Time":
player2_time,
1567                       "Agent 1 Average Duration":
np.mean(player1_durations),
1568                       "Agent 2 Average Duration":
np.mean(player2_durations),
1569                       "Agent 1 Longest Duration": np.max(player1_durations),
1570                       "Agent 2 Longest Duration": np.max(player2_durations),
1571                       "Agent 1 Score": -1, "Agent 2 Score": 1}
1572             return -1, info
1573         player1_time += time_increment
1574         state = my_go.transition(state, player1_action)
1575         if (my_go.is_terminal_state(state)):

```

```

1576         break
1577     start_time = time.time()
1578     player2_action = agent2.get_move(state.clone(), player2_time)
1579     duration = time.time() - start_time
1580     player2_durations.append(duration)
1581     player2_time -= duration
1582     if (player2_time ≤ 0):
1583         print("Player 2 over time")
1584         if hard_time_cutoff:
1585             info = {"Agent 1 End Time": player1_time, "Agent 2 End Time":
player2_time,
1586                     "Agent 1 Average Duration":
np.mean(player1_durations),
1587                     "Agent 2 Average Duration":
np.mean(player2_durations),
1588                     "Agent 1 Longest Duration": np.max(player1_durations),
1589                     "Agent 2 Longest Duration": np.max(player2_durations),
1590                     "Agent 1 Score": -1, "Agent 2 Score": 1}
1591             return 1, info
1592     else:
1593         player2_time += time_increment
1594         state = my_go.transition(state, player2_action)
1595         info = {"Agent 1 End Time": player1_time, "Agent 2 End Time":
player2_time,
1596                 "Agent 1 Average Duration": np.mean(player1_durations),
1597                 "Agent 2 Average Duration": np.mean(player2_durations),
1598                 "Agent 1 Longest Duration": np.max(player1_durations),
1599                 "Agent 2 Longest Duration": np.max(player2_durations),
1600                 "Agent 1 Score": -1, "Agent 2 Score": 1}
1601         return my_go.evaluate_terminal(state), info
1602
1603
1604 def run_many(agent1, agent2, num_games=10, verbose=True, size=5):
1605     print(f"Number of games: {num_games}")
1606     agent1_score = 0
1607     agent2_score = 0
1608     agent1_score_black = 0
1609     agent2_score_black = 0
1610     agent1_average_duration = 0
1611     agent2_average_duration = 0
1612
1613     agent1_longest_duration = 0
1614     agent2_longest_duration = 0
1615
1616     agent1_average_time_remaining = 0
1617     agent2_average_time_remaining = 0
1618
1619     agent1_min_time_remaining = float('inf')
1620     agent2_min_time_remaining = float('inf')
1621
1622     for _ in tqdm.tqdm(range(int(num_games / 2))):

```

```

1623         result, info = run_game(agent1, agent2)
1624         agent1_score += result
1625         agent2_score += -result
1626         agent1_score_black += result
1627
1628         agent1_average_duration += info["Agent 1 Average Duration"] /
num_games
1629         agent2_average_duration += info["Agent 2 Average Duration"] /
num_games
1630
1631         agent1_longest_duration = max(
1632             agent1_longest_duration, info["Agent 1 Longest Duration"])
1633         agent2_longest_duration = max(
1634             agent2_longest_duration, info["Agent 2 Longest Duration"])
1635
1636         agent1_average_time_remaining += info["Agent 1 End Time"] / num_games
1637         agent2_average_time_remaining += info["Agent 2 End Time"] / num_games
1638
1639         agent1_min_time_remaining = min(
1640             agent1_min_time_remaining, info["Agent 1 End Time"])
1641         agent2_min_time_remaining = min(
1642             agent2_min_time_remaining, info["Agent 2 End Time"])
1643
1644         result, info = run_game(agent2, agent1)
1645
1646         # Note that since player 2 goes first in the second game,
1647         # The stats will look backwards
1648         agent2_score_black += result
1649         agent1_score += -result
1650         agent2_score += result
1651
1652         agent1_average_duration += info["Agent 2 Average Duration"] /
num_games
1653         agent2_average_duration += info["Agent 1 Average Duration"] /
num_games
1654
1655         agent1_longest_duration = max(
1656             agent1_longest_duration, info["Agent 2 Longest Duration"])
1657         agent2_longest_duration = max(
1658             agent2_longest_duration, info["Agent 1 Longest Duration"])
1659
1660         agent1_average_time_remaining += info["Agent 2 End Time"] / num_games
1661         agent2_average_time_remaining += info["Agent 1 End Time"] / num_games
1662
1663         agent1_min_time_remaining = min(
1664             agent1_min_time_remaining, info["Agent 2 End Time"])
1665         agent2_min_time_remaining = min(
1666             agent2_min_time_remaining, info["Agent 1 End Time"])
1667
1668         if verbose:
1669             print("Agent 1: " + str(agent1) + " Score: " + str(agent1_score))

```

```

1670     print("Agent 2: " + str(agent2) + " Score: " + str(agent2_score))
1671     print("Agent 1: " + str(agent1) + " Score with Black (first move): " +
1672           str(agent1_score_black))
1673     print("Agent 2: " + str(agent2) + " Score with Black (first move): " +
1674           str(agent2_score_black))
1675     print("Agent 1: " + str(agent1) + " Average Duration: " +
1676           str(agent1_average_duration))
1677     print("Agent 2: " + str(agent2) + " Average Duration: " +
1678           str(agent2_average_duration))
1679     print("Agent 1: " + str(agent1) + " Longest Duration: " +
1680           str(agent1_longest_duration))
1681     print("Agent 2: " + str(agent2) + " Longest Duration: " +
1682           str(agent2_longest_duration))
1683     print("Agent 1: " + str(agent1) + " Average Time Remaining: " +
1684           str(agent1_average_time_remaining))
1685     print("Agent 2: " + str(agent2) + " Average Time Remaining: " +
1686           str(agent2_average_time_remaining))
1687     print("Agent 1: " + str(agent1) + " Min Time Remaining: " +
1688           str(agent1_min_time_remaining))
1689     print("Agent 2: " + str(agent2) + " Min Time Remaining: " +
1690           str(agent2_min_time_remaining))
1691
1692     return agent1_score, agent2_score
1693
1694
1695 def run_game_with_gui(agent, size=5):
1696     """
1697     Run a single game between a human and an agent with a GUI.
1698     :param agent: The agent to play against (must be a subclass of GameAgent)
1699     """
1700     my_go = GoProblem(size=size)
1701     state = my_go.start_state
1702     gui = GoGUI(my_go)
1703     while (not my_go.is_terminal_state(state)):
1704         player1_action = agent.get_move(state.clone(), 1)
1705         state = my_go.transition(state, player1_action)
1706         gui.update_state(state)
1707         gui.render()
1708         if (my_go.is_terminal_state(state)):
1709             break
1710         action = None
1711         while action is None:
1712             while action not in state.legal_actions():
1713                 action = gui.get_user_input_action()
1714                 gui.render()
1715                 clock.tick(60)
1716             print("Human Action:", action, ", which corresponds to coordinate", my_go.action_index_to_string(action))
1717             gui.render()
1718             clock.tick(60)
1719             state = my_go.transition(state, action)

```

```

1720     gui.update_state(state)
1721     gui.render()
1722     clock.tick(60)
1723     print("Done!")
1724     if my_go.evaluate_terminal(state) == 1:
1725         print("Agent wins!")
1726     else:
1727         print("You won!")
1728
1729 def create_agent(agent_type: str, **kwargs):
1730     """
1731     Factory function to create agents based on command line arguments
1732
1733     :param agent_type: The type of agent to create (string)
1734     :param kwargs: Additional arguments for the agent (e.g., depth,
1735     parameters, etc.)
1736     """
1737     if agent_type.lower() == "alphabeta":
1738         depth = kwargs.get('depth', 2)
1739         return AlphaBetaAgent(depth=depth)
1740     elif agent_type.lower() == "random":
1741         return RandomAgent()
1742     elif agent_type.lower() == "greedy":
1743         return GreedyAgent()
1744     elif agent_type.lower() == "mcts":
1745         return MCTSAgent()
1746     # Add more agent types here as needed
1747     else:
1748         raise ValueError(f"Unknown agent type: {agent_type}")
1749
1750 def parse_args():
1751     parser = argparse.ArgumentParser(description='Go Game Runner')
1752
1753     # Mode selection
1754     parser.add_argument('--mode', choices=['gui', 'vs', 'tournament'],
1755     default='gui',
1756     help='Run mode: gui (play against AI), vs (single game
1757     between agents), tournament (multiple games)')
1758
1759     # Agent configuration
1760     parser.add_argument('--agent1-type', default='alphabeta',
1761     help='Type of agent 1 (e.g., alphabeta)')
1762     parser.add_argument('--agent1-depth', type=int, default=2,
1763     help='Depth limit for agent 1 if applicable')
1764
1765     parser.add_argument('--agent2-type', default='alphabeta',
1766     help='Type of agent 2 (e.g., alphabeta)')
1767     parser.add_argument('--agent2-depth', type=int, default=2,
1768     help='Depth limit for agent 2 if applicable')
1769
1770     # Game settings

```

```

1768 parser.add_argument('--time-limit', type=float, default=15,
1769                      help='Time limit per player in seconds')
1770 parser.add_argument('--time-increment', type=float, default=1,
1771                      help='Time increment per move in seconds')
1772 parser.add_argument('--soft-time', action='store_true',
1773                      help='Continue game even if time limit is exceeded')
1774 parser.add_argument('--size', type=int, default=5,
1775                      help='Size of the Go board')
1776
1777 # Tournament settings
1778 parser.add_argument('--num-games', type=int, default=10,
1779                      help='Number of games to play in tournament mode')
1780 parser.add_argument('--quiet', action='store_true',
1781                      help='Suppress detailed output in tournament mode')
1782
1783 args = parser.parse_args()
1784 return args
1785
1786 def main():
1787     args = parse_args()
1788
1789     # Create agents based on arguments
1790     agent1 = create_agent(args.agent1_type, depth=args.agent1_depth)
1791
1792     if args.mode == 'gui':
1793         run_game_with_gui(agent1)
1794     else:
1795         agent2 = create_agent(args.agent2_type, depth=args.agent2_depth)
1796         if args.mode == 'vs':
1797             result, info = run_game(agent1, agent2,
1798                                     time_limit=args.time_limit,
1799                                     time_increment=args.time_increment,
1800                                     hard_time_cutoff=not args.soft_time,
1801                                     size=args.size)
1802             print("Game Info:", info)
1803         elif args.mode == 'tournament':
1804             run_many(agent1, agent2,
1805                     num_games=args.num_games,
1806                     verbose=not args.quiet,
1807                     size=args.size)
1808
1809
1810 if __name__ == "__main__":
1811     main()
1812
1813 import pygame
1814 import sys
1815 from go_search_problem import GoProblem, GoState
1816
1817
1818 class GoGUI:

```

```

1819 # Define GUI colors
1820 BOARD = (210, 180, 140) # brown
1821 EMPTY = (0, 0, 0) # black
1822 P1 = (0, 0, 0) # black
1823 P2 = (255, 255, 255) # white
1824 BUTTON = (200, 200, 200) # grey
1825 BUTTON_HOVER = (180, 180, 180) # darker grey
1826 BUTTON_TEXT = (0, 0, 0) # black
1827 COLOR_MAP = [EMPTY, P1, P2]
1828
1829 def __init__(self, problem: GoProblem):
1830     # Initialize Pygame
1831     print("Setting up Board... ")
1832     print("Use the arrow keys to navigate and the enter key to select an
action.")
1833     pygame.init()
1834
1835     # Constants
1836     self.WIDTH, self.HEIGHT = 600, 700 # Increased height for pass button
1837     self.BOARD_SIZE = problem.start_state.size
1838     self.CELL_SIZE = 600 // self.BOARD_SIZE # Using original width for
board
1839
1840     # Pass button dimensions
1841     self.BUTTON_WIDTH = 100
1842     self.BUTTON_HEIGHT = 40
1843     self.BUTTON_X = (self.WIDTH - self.BUTTON_WIDTH) // 2
1844     self.BUTTON_Y = 620 # Position below the board
1845     self.BUTTON_COLOR = self.BUTTON
1846
1847     # Set up the display
1848     self.screen = pygame.display.set_mode((self.WIDTH, self.HEIGHT))
1849     pygame.display.set_caption("Go Game")
1850
1851     # Initialize font
1852     self.font = pygame.font.Font(None, 36)
1853
1854     self.problem = problem
1855     self.state = problem.start_state
1856     self.cursor_pos = [self.BOARD_SIZE // 2, self.BOARD_SIZE // 2]
1857
1858 def render(self):
1859     self.screen.fill(self.BOARD)
1860     self.draw_board()
1861     self.draw_pieces()
1862     self.draw_cursor()
1863     self.draw_pass_button()
1864     pygame.display.flip()
1865
1866 def draw_pass_button(self):
1867     # Check if mouse is hovering over button

```



```

1868         mouse_pos = pygame.mouse.get_pos()
1869         button_rect = pygame.Rect(self.BUTTON_X, self.BUTTON_Y,
self.BUTTON_WIDTH, self.BUTTON_HEIGHT)
1870         button_color = self.BUTTON_HOVER if
button_rect.collidepoint(mouse_pos) else self.BUTTON_COLOR
1871
1872         # Draw button
1873         pygame.draw.rect(self.screen, button_color, button_rect)
1874         pygame.draw.rect(self.screen, self.BUTTON_TEXT, button_rect, 2) #
Border
1875
1876         # Draw text
1877         text = self.font.render("PASS", True, self.BUTTON_TEXT)
1878         text_rect = text.get_rect(center=button_rect.center)
1879         self.screen.blit(text, text_rect)
1880
1881     def process_window_event(self, event):
1882         if event.type == pygame.QUIT:
1883             pygame.quit()
1884             sys.exit()
1885
1886     def is_pass_button_clicked(self, pos):
1887         button_rect = pygame.Rect(self.BUTTON_X, self.BUTTON_Y,
self.BUTTON_WIDTH, self.BUTTON_HEIGHT)
1888         return button_rect.collidepoint(pos)
1889
1890     def get_user_input_action(self):
1891         for event in pygame.event.get():
1892             self.process_window_event(event)
1893
1894             if event.type == pygame.MOUSEBUTTONDOWN:
1895                 if event.button == 1: # Left click
1896                     if self.is_pass_button_clicked(event.pos):
1897                         return self.BOARD_SIZE * self.BOARD_SIZE # Pass move
1898
1899             if event.type == pygame.KEYDOWN:
1900                 if event.key == pygame.K_UP:
1901                     self.cursor_pos[1] = max(0, self.cursor_pos[1] - 1)
1902                 elif event.key == pygame.K_DOWN:
1903                     self.cursor_pos[1] = min(
1904                         self.BOARD_SIZE - 1, self.cursor_pos[1] + 1)
1905                 elif event.key == pygame.K_LEFT:
1906                     self.cursor_pos[0] = max(0, self.cursor_pos[0] - 1)
1907                 elif event.key == pygame.K_RIGHT:
1908                     self.cursor_pos[0] = min(
1909                         self.BOARD_SIZE - 1, self.cursor_pos[0] + 1)
1910                 elif event.key == pygame.K_RETURN:
1911                     return self.cursor_pos[1] * self.BOARD_SIZE +
self.cursor_pos[0]
1912                 elif event.key == pygame.K_SPACE: # Added space as
alternative for pass

```



```

1913         return self.BOARD_SIZE * self.BOARD_SIZE
1914     return None
1915
1916     def update_state(self, action):
1917         if action is not None and action in
self.problem.get_available_actions(self.state):
1918             self.state = self.problem.transition(self.state, action)
1919         elif action is not None:
1920             self.state = action
1921
1922     def draw_cursor(self):
1923         x, y = self.cursor_pos
1924         pygame.draw.rect(self.screen, (255, 0, 0),
1925             (x * self.CELL_SIZE, y * self.CELL_SIZE,
self.CELL_SIZE, self.CELL_SIZE), 3)
1926
1927     def draw_board(self):
1928         for i in range(self.BOARD_SIZE):
1929             # Draw horizontal lines
1930             pygame.draw.line(self.screen, self.EMPTY, (0, i * self.CELL_SIZE),
1931                 (600, i * self.CELL_SIZE))
1932             # Draw vertical lines
1933             pygame.draw.line(self.screen, self.EMPTY, (i * self.CELL_SIZE, 0),
1934                 (i * self.CELL_SIZE, 600))
1935             # Draw bottom line
1936             pygame.draw.line(self.screen, self.EMPTY, (0, self.BOARD_SIZE *
self.CELL_SIZE),
1937                 (600, self.BOARD_SIZE * self.CELL_SIZE))
1938
1939     def draw_pieces(self):
1940         board = self.state.get_board()
1941         for y in range(self.BOARD_SIZE):
1942             for x in range(self.BOARD_SIZE):
1943                 if board[0][y][x] == 1:
1944                     self.draw_piece(x, y, self.P1)
1945                 elif board[1][y][x] == 1:
1946                     self.draw_piece(x, y, self.P2)
1947
1948     def draw_piece(self, x, y, color):
1949         center = (x * self.CELL_SIZE + self.CELL_SIZE // 2,
1950             y * self.CELL_SIZE + self.CELL_SIZE // 2)
1951         pygame.draw.circle(self.screen, color, center, self.CELL_SIZE // 2 -
2)
1952
1953
1954     def main():
1955         problem = GoProblem()
1956         gui = GoGUI(problem)
1957         clock = pygame.time.Clock()
1958
1959         while True:

```

```

1960         action = gui.get_user_input_action()
1961         gui.update_state(action)
1962         gui.render()
1963         clock.tick(60)
1964
1965
1966 if __name__ == "__main__":
1967     main()
1968
1969 from typing import Sequence, Type
1970 # import go_utils
1971 import numpy as np
1972 from adversarial_search_problem import AdversarialSearchProblem, GameState
1973 import copy
1974 import go_utils
1975 from pyspiel import Game
1976
1977 Action = int
1978
1979 DEFAULT_SIZE = 9
1980
1981 class GoState(GameState):
1982     """
1983     A state of the game of Go.
1984     Includes methods and properties for the state of the board, player to
1985     move, and other useful methods
1986     """
1987     def __init__(self, pyspiel_state: Game, player_to_move: int = 0):
1988         """
1989         Initialize GoState with pyspiel as backend Go engine.
1990         The initial state is created with a call to create_go_game() in
1991         go_utils.py
1992         Every other state will be generated from applying actions to the
1993         initial state.
1994         This essentially functions as a wrapper class to conver pyspiel game
1995         states to
1996         The ASP interface used previously.
1997
1998         :param pyspiel_state: pyspiel state of the game
1999         :param player_to_move: player to move
2000         """
2001         self.internal_state = pyspiel_state
2002         self.size = int(np.sqrt(len(pyspiel_state.observation_tensor())) / 4))
2003
2004     def player_to_move(self) → int:
2005         """
2006         Get the current player to move
2007         :return: player to move BLACK (0) or WHITE (1)
2008         """
2009         return self.internal_state.current_player()

```

```

2007
2008     def get_board(self) → np.ndarray:
2009         """
2010         Return the current board as a numpy array
2011         The board will have shape (4, size, size)
2012         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.
2013         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.
2014         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.
2015         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.
2016
2017         This is the default observation tensor used by pyspiel.
2018         """
2019         return np.array(self.internal_state.observation_tensor(0)).reshape(-1,
self.size, self.size)
2020
2021     def terminal_value(self) → float:
2022         """
2023         Return the terminal value of the game.
2024         :return: 1 if BLACK wins, -1 if WHITE wins
2025         """
2026         return self.internal_state.returns()
2027
2028     def clone(self) → GameState:
2029         """
2030         Create a copy of the current game state.
2031         This is used for safety with the game runner.
2032         We don't want search algorithms to be able to directly modify the game
state,
2033         so we only pass a copy of the state to the search algorithms.
2034         :return: a copy of the current game state
2035         """
2036         return GoState(self.internal_state.clone(),
self.internal_state.current_player())
2037
2038     def is_terminal_state(self) → bool:
2039         """
2040         Checks if the game is in a terminal state.
2041         The state is if there are no legal actions left or the players have
passed twice in a row.
2042
2043         :return: True if the game is in a terminal state, False otherwise
2044         """
2045         return self.internal_state.is_terminal()
2046
2047     def legal_actions(self) → Sequence[Action]:
2048         """
2049         Return all possible legal actions for the given state.

```

```

2050     Note: Actions are represented as integers, by default.
2051     For a more human-readable representation, use action_index_to_coord()
2052
2053     NOTE: It is preferable to get the available actions from the search
problem,
2054     not this state.
2055
2056     :return: list of legal actions
2057     """
2058     return self.internal_state.legal_actions()
2059
2060     def apply_action(self, action: Action):
2061         """
2062         Apply action and update internal state.
2063         Action must be an int, not a coordinate.
2064
2065         NOTE: It is preferable to use the transition function from the search
problem,
2066         not this method to apply actions.
2067         """
2068         self.internal_state.apply_action(action)
2069
2070     def get_pieces_coordinates(self, player_index: int):
2071         """
2072         Get the indices of the pieces of the given player.
2073         :param player_index: 0 for BLACK, 1 for WHITE
2074         :return: list of coordinates of the pieces of the given player
2075         """
2076         player_board = np.array(self.internal_state.observation_tensor(
2077             0)).reshape((-1, self.size, self.size))[player_index]
2078         return np.argwhere(player_board == 1)
2079
2080     def get_pieces_array(self, player_index):
2081         """
2082         Get the 2D array of the pieces of the given player.
2083         The array will have shape (size, size) and will have 1's where the
pieces are and 0's elsewhere.
2084
2085         :param player_index: 0 for BLACK, 1 for WHITE
2086         :return: 2D np array of the pieces of the given player
2087         """
2088         player_board = np.array(self.internal_state.observation_tensor(
2089             0)).reshape((-1, self.size, self.size))[player_index]
2090         return player_board
2091
2092     def get_empty_spaces(self):
2093         """
2094         return a 2D array of the empty spaces on the board
2095         The array will have shape (size, size) and will have 1's where the
empty spaces are and 0's elsewhere.
2096

```

```

2097         :return: 2D np array of the empty spaces on the board
2098         """
2099         return self.internal_state.observation_tensor(2)
2100
2101     def action_index_to_coord(self, action: Action) → tuple[int, int]:
2102         """
2103         Convert an action index to a coordinate.
2104         :param action: action index
2105         :return: coordinate (x, y)
2106         """
2107         return (action % self.size, action // self.size)
2108
2109     def __repr__(self):
2110         return str(self.internal_state)
2111
2112
2113 class GoProblem(AdversarialSearchProblem[GoState, Action]):
2114     def __init__(self, size=DEFAULT_SIZE, state=None, player_to_move=0):
2115         """
2116         Create a new Go search problem.
2117         If no state is provided, a new game is created with the given size.
2118         """
2119         if state is None:
2120             game_state = go_utils.create_go_game(size)
2121         else:
2122             game_state = state
2123         self.start_state = GoState(game_state, player_to_move)
2124
2125     def get_available_actions(self, state: GoState) → Sequence[Action]:
2126         """
2127         Get the available actions for the given state.
2128         Use this to get the list of available actions for a given state.
2129         Note: An action in this case is an integer in range [0, size^2].
2130         Each action index corresponds to a coordinate on the board (x, y) =
2131         (action % size, action // size).
2132         With action=size**2 reserved for the pass action.
2133
2134         :param state: current state
2135         :return: list of available actions
2136         """
2137         return state.legal_actions()
2138
2139     def transition(self, state: GoState, action: Action) → GoState:
2140         """
2141         Return new_state resulting from applying action to state.
2142
2143         :param state: current state
2144         :param action: action to apply
2145         :return: new state resulting from applying action to state
2146         """
2147         new_state = state.clone()

```

```

2147         new_state.apply_action(action)
2148         return new_state
2149
2150     def is_terminal_state(self, state: GoState) → bool:
2151         """
2152         Return if the given state is a terminal state.
2153         State is terminal if no legal actions are available or the players
2154         have passed twice in a row.
2155
2156         :param state: current state
2157         :return: True if the state is terminal, False otherwise
2158         """
2159         return state.is_terminal_state()
2160
2161     def evaluate_terminal(self, state: GoState) → float:
2162         """
2163         Get the value of the terminal state.
2164         The value is 1 if BLACK wins and -1 if WHITE wins.
2165
2166         :param state: current state
2167         :return: value of the terminal state
2168         """
2169         return state.terminal_value()[0]
2170
2171     def action_index_to_string(self, action: Action) → str:
2172         """
2173         Convert an Action (index) to a string.
2174         """
2175         return "(" + str(action % self.start_state.size) + ", " + str(action
2176         // self.start_state.size) + ")"
2177
2178     import numpy as np
2179     import pyspiel
2180     import pygame
2181     import sys
2182
2183     def create_go_game(size):
2184         """
2185         load open-spiel game with provided size
2186         """
2187         if size == 5:
2188             komi = 0.5
2189         elif size == 9:
2190             komi = 5.5
2191         else:
2192             komi = 7.5
2193         game = pyspiel.load_game("go", {"board_size": size, "komi": komi})
2194         state = game.new_initial_state()
2195         return state

```

```

2196
2197 from go_search_problem import GoProblem
2198
2199 BLACK = 0
2200 WHITE = 1
2201
2202 class GoProblemSimpleHeuristic(GoProblem):
2203     def __init__(self, state=None):
2204         super().__init__(state=state)
2205
2206     def heuristic(self, state, player_index):
2207         """
2208         Very simple heuristic that just compares the number of pieces for each
2209         player
2210
2211         Having more pieces (>1) than the opponent means that some were
2212         captured, capturing is generally good.
2213         """
2214         return len(state.get_pieces_coordinates(BLACK)) -
2215            len(state.get_pieces_coordinates(WHITE))
2216
2217     def __str__(self) → str:
2218         return "Simple Heuristic"
2219
2220 class GoProblemLearnedHeuristic(GoProblem):
2221     def __init__(self, model=None, state=None,):
2222         super().__init__(state=state)
2223         self.model = model
2224
2225     def encoding(self, state):
2226         pass
2227
2228     def heuristic(self, state, player_index):
2229         pass
2230
2231     def __str__(self) → str:
2232         return "Learned Heuristic"
2233
2234 class GoProblemAdvancedHeuristic(GoProblem):
2235     def __init__(self, state=None):
2236         super().__init__(state=state)
2237
2238     def heuristic(self, state, player_index):
2239         """
2240         Advanced heuristic for evaluating a Go game state based on:
2241         1. Piece Count: Difference in the number of stones between the player
2242         and opponent.
2243         2. Territory Control: Influence over empty spaces, based on proximity
2244         to placed stones.

```

```

2242     3. Center Control: Control over the central area of the board.
2243     4. Liberties: Number of adjacent empty spaces around the player's
stones.
2244     5. Weighted Scoring
2245
2246     :param state: Current game state
2247     :param player_index: Player to evaluate (0 for BLACK, 1 for WHITE)
2248     :return: Heuristic score favoring the current player
2249     """
2250
2251     # if player_index=0 (BLACK), opponent_index=1 (WHITE)
2252     # if player_index=1 (WHITE), opponent_index=0 (BLACK)
2253     opponent_index = 1 - player_index
2254
2255     board = state.get_board()
2256     size = state.size
2257
2258     # 1. Piece count
2259     player_pieces = len(state.get_pieces_coordinates(player_index))
2260     opponent_pieces = len(state.get_pieces_coordinates(opponent_index))
2261     piece_difference = player_pieces - opponent_pieces
2262
2263     # 2. Territory control
2264     # Use empty spaces as a proxy for potential territory
2265     empty_spaces = board[2]
2266
2267     player_board = state.get_pieces_array(player_index)
2268     opponent_board = state.get_pieces_array(opponent_index)
2269
2270     def count_potential_territory(player_board, empty_spaces):
2271         territory_score = 0
2272         for y in range(size):
2273             for x in range(size):
2274                 if empty_spaces[y, x] == 1:
2275                     # Count the number of player's stones in a 3x3
neighborhood around the empty space (y, x).
2276                     nearby_player_stones = player_board[max(0, y-
1):min(size, y+2),
2277                                                         max(0, x-
1):min(size, x+2)].sum()
2278                     territory_score += nearby_player_stones
2279                 return territory_score
2280
2281     player_territory = count_potential_territory(player_board,
empty_spaces)
2282     opponent_territory = count_potential_territory(opponent_board,
empty_spaces)
2283     territory_difference = player_territory - opponent_territory
2284
2285     # 3. Center control
2286     def center_control_score(board):

```



```

2287         # Evaluate center control by counting the number of stones each
player has in the central region of the board.
2288         # Note: center_range[0] gives the start of range and
center_range[-1] gives the end of the range.
2289         center_range = range(size // 4, 3 * size // 4)
2290         center_board = board[center_range[0]:center_range[-1],
center_range[0]:center_range[-1]]
2291         return center_board.sum()
2292
2293         player_center_control = center_control_score(player_board)
2294         opponent_center_control = center_control_score(opponent_board)
2295         center_control_difference = player_center_control -
opponent_center_control
2296
2297         # 4. Liberties
2298         def count_liberties(board):
2299             liberties = 0
2300
2301             # For each player's stone, count the number of adjacent empty
spaces (liberties).
2302             for y in range(size):
2303                 for x in range(size):
2304                     if board[y, x] == 1:
2305                         liberties += sum([
2306                             (y > 0 and empty_spaces[y-1, x] == 1),
2307                             (y < size-1 and empty_spaces[y+1, x] == 1),
2308                             (x > 0 and empty_spaces[y, x-1] == 1),
2309                             (x < size-1 and empty_spaces[y, x+1] == 1)
2310                         ])
2311             return liberties
2312
2313         player_liberties = count_liberties(player_board)
2314         opponent_liberties = count_liberties(opponent_board)
2315         liberty_difference = player_liberties - opponent_liberties
2316
2317         # Weights for each component
2318         weights = {
2319             'pieces': 2.0,
2320             'territory': 1.5,
2321             'center_control': 1.0,
2322             'liberties': 1.0
2323         }
2324
2325         # Combine scores with weights
2326         total_score = (
2327             weights['pieces'] * piece_difference +
2328             weights['territory'] * territory_difference +
2329             weights['center_control'] * center_control_difference +
2330             weights['liberties'] * liberty_difference
2331         )
2332

```

```

2333         normalized_score = total_score / (size * size)
2334
2335         return normalized_score if player_index == BLACK else -
normalized_score
2336
2337     def __str__(self) → str:
2338         return "Advanced Heuristic"
2339
2340 # %%
2341 # Needed if running on Colab
2342 !pip3 install open-spiel
2343 !pip3 install torch
2344
2345 # %%
2346 import numpy as np
2347 import torch
2348 import torch.nn as nn
2349 import torch.optim as optim
2350 import random
2351 from go_search_problem import GoProblem, GoState
2352 from heuristic_go_problems import GoProblemLearnedHeuristic,
GoProblemSimpleHeuristic
2353 from agents import GreedyAgent, RandomAgent, MCTSAgent, GameAgent
2354 import matplotlib.pyplot as plt
2355 from tqdm import tqdm
2356 from game_runner import run_many
2357 import pickle
2358
2359 torch.set_default_tensor_type(torch.FloatTensor)
2360
2361 # %%
2362 def load_dataset(path: str):
2363     with open(path, 'rb') as f:
2364         dataset = pickle.load(f)
2365     return dataset
2366
2367 dataset_5×5 = load_dataset('dataset_5×5.pkl')
2368 # dataset_9×9 = load_dataset('9×9_dataset.pkl')
2369
2370 # %%
2371 def save_model(path: str, model, input_size=None):
2372     """
2373     Save model to a file
2374     Input:
2375         path: path to save model to
2376         model: Pytorch model to save
2377     """
2378
2379     torch.save({
2380         'model_state_dict': model.state_dict(),
2381         }, path)

```

```

2382
2383 def load_model(path: str, model):
2384     """
2385     Load model from file
2386
2387     Note: you still need to provide a model (with the same architecture as the
saved model))
2388
2389     Input:
2390         path: path to load model from
2391         model: Pytorch model to load
2392     Output:
2393         model: Pytorch model loaded from file
2394     """
2395     checkpoint = torch.load(path)
2396     model.load_state_dict(checkpoint['model_state_dict'])
2397     return model
2398
2399 # %% [markdown]
2400 # # Task 1: Convert GameState to Features
2401
2402 # %%
2403 def get_features(game_state: GoState):
2404     """
2405     Map a game state to a list of features.
2406
2407     Some useful functions from game_state include:
2408         game_state.size: size of the board
2409         get_pieces_coordinates(player_index): get coordinates of all pieces of
a player (0 or 1)
2410         get_pieces_array(player_index): get a 2D array of pieces of a player
(0 or 1)
2411
2412         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
2413
2414         Descriptions of these methods can be found in the GoState
2415
2416     Input:
2417         game_state: GoState to encode into a fixed size list of features
2418     Output:
2419         features: list of features
2420     """
2421     board_size = game_state.size
2422
2423     # TODO: Encode game_state into a list of features
2424     features = []
2425
2426     board = game_state.get_board()
2427

```

```

2428     for channel in range(4):
2429         for row in range(board_size):
2430             for col in range(board_size):
2431                 features.append(board[channel][row][col])
2432
2433     return features
2434
2435 # %%
2436 #TESTING
2437 class MockGoState:
2438     def __init__(self, size, board, player_to_move):
2439         """
2440         Mock implementation of the GoState class for testing.
2441         :param size: Size of the board (n x n).
2442         :param board: A 3D list (4 x n x n) representing the board state.
2443         :param player_to_move: The current player to move (0 or 1).
2444         """
2445         self.size = size
2446         self.board = board
2447         self.player_to_move = player_to_move
2448
2449     def get_board(self):
2450         return self.board
2451
2452 def test_get_features():
2453     # Test Case 1: Empty 3x3 Board with Player 0 to move
2454     size = 3
2455     board = [
2456         # Player 0 (white)
2457         [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
2458         # Player 1 (black)
2459         [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
2460         # Empty spaces
2461         [[1, 1, 1], [1, 1, 1], [1, 1, 1]],
2462         # Player to move (0 for Player 0, 1 for Player 1)
2463         [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
2464     ]
2465     game_state = MockGoState(size, board, player_to_move=0)
2466     features = get_features(game_state)
2467     assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got {len(features)}"
2468     assert features == [0] * 9 + [0] * 9 + [1] * 9 + [1] * 9, "Feature vector does not match expected solution."
2469
2470     # Test Case 2: Filled 3x3 Board with Player 1 to move
2471     board = [
2472         [[0, 1, 0], [0, 0, 0], [0, 0, 0]],
2473         [[0, 0, 0], [1, 0, 0], [0, 0, 0]],
2474         [[1, 0, 1], [0, 1, 1], [1, 1, 1]],
2475         [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
2476     ]

```

```

2477     game_state = MockGoState(size, board, player_to_move=1)
2478     features = get_features(game_state)
2479     assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got
{len(features)}"
2480     expected_features = [
2481         0, 1, 0, 0, 0, 0, 0, 0, 0,
2482         0, 0, 0, 1, 0, 0, 0, 0, 0,
2483         1, 0, 1, 0, 1, 1, 1, 1, 1,
2484         0, 0, 0, 0, 0, 0, 0, 0, 0
2485     ]
2486     assert features == expected_features, f"Feature vector does not match
expected solution."
2487
2488     # Test Case 3: Filled 2x2 Board with Player 0 to move
2489     size = 2
2490     board = [
2491         [[1, 0], [0, 1]],
2492         [[0, 1], [1, 0]],
2493         [[0, 0], [0, 0]],
2494         [[1, 1], [1, 1]]
2495     ]
2496     game_state = MockGoState(size, board, player_to_move=0)
2497     features = get_features(game_state)
2498     assert len(features) == 4 * size * size, f"Expected {4 * size * size}, got
{len(features)}"
2499     expected_features = [
2500         1, 0, 0, 1,
2501         0, 1, 1, 0,
2502         0, 0, 0, 0,
2503         1, 1, 1, 1
2504     ]
2505     assert features == expected_features, f"Feature vector does not match
expected solution."
2506
2507     print("All tests passed!")
2508
2509 test_get_features()
2510
2511 # %%
2512 # Print information about first data point
2513 data_point = dataset_5x5[0]
2514 features = get_features(data_point[0])
2515 action = data_point[1]
2516 result = data_point[2]
2517 print(data_point[0])
2518 print("features", features)
2519 print("Action #", action)
2520 print("Game Result", result)
2521
2522 # %% [markdown]
2523 # # Task 2: Supervised Learning of a Value Network

```

```

2524
2525 # %%
2526 class ValueNetwork(nn.Module):
2527     def __init__(self, input_size):
2528         super(ValueNetwork, self).__init__()
2529
2530         # TODO: What should the output size of a Value function be?
2531
2532         ''' Handout: the goal is to classify each state as a future
2533         win for one player or the other, or more generally, to
2534         generate a prediction in the range [-1, +1] that is indicative
2535         of which player will win the game.'''
2536
2537         output_size = 1
2538
2539         # TODO: Add more layers, non-linear functions, etc.
2540
2541         # Layers
2542         self.fc1 = nn.Linear(input_size, 32)
2543         self.fc2 = nn.Linear(32, 16)
2544         self.fc3 = nn.Linear(16, output_size)
2545
2546         # Activation functions
2547         self.relu = nn.ReLU()
2548         self.tanh = nn.Tanh()
2549
2550     def forward(self, x):
2551         """
2552         Run forward pass of network
2553
2554         Input:
2555         x: input to network
2556         Output:
2557         output of network
2558         """
2559         # TODO: Update as more layers are added
2560         z1 = self.fc1(x)
2561         a1 = self.relu(z1)
2562
2563         z2 = self.fc2(a1)
2564         a2 = self.relu(z2)
2565
2566         z3 = self.fc3(a2)
2567         a3 = self.relu(z3)
2568
2569         return a3
2570
2571 # %%
2572 # This will not produce meaningful outputs until trained, but you can test for
2573 # syntax errors
2574 features_tensor = torch.Tensor(features)

```

```

2574 value_net = ValueNetwork(len(features))
2575 print("predicted Value", value_net(features_tensor))
2576
2577 # %%
2578 def train_value_network(dataset, num_epochs, learning_rate):
2579     """
2580     Train a value network on the provided dataset.
2581
2582     Input:
2583         dataset: list of (state, action, result) tuples
2584         num_epochs: number of epochs to train for
2585         learning_rate: learning rate for gradient descent
2586     Output:
2587         model: trained model
2588     """
2589     # Make sure dataset is shuffled for better performance
2590     random.shuffle(dataset)
2591     # You may find it useful to create train/test sets to better track
performance/overfit/underfit
2592     train_size = int(0.8 * len(dataset))
2593     train_dataset = dataset[:train_size]
2594     test_dataset = dataset[train_size:]
2595
2596     # Get input size
2597     sample_features = get_features(dataset[0][0])
2598     input_size = len(sample_features)
2599
2600     # TODO: Create model
2601     model = ValueNetwork(input_size)
2602
2603     # TODO: Specify Loss Function
2604     loss_function = nn.MSELoss()
2605
2606     # You can use Adam, which is stochastic gradient descent with ADAPtive
Momentum
2607     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
2608
2609     batch_size = 32
2610
2611     batch_loss = 0
2612     batch_counter = 0
2613
2614     for epoch in range(num_epochs):
2615         total_train_loss = 0.0
2616         for data_point in train_dataset:
2617             state = data_point[0]
2618             features = get_features(state)
2619             features_tensor = torch.tensor(features, dtype=torch.float32)
2620
2621             # TODO: What should the desired output of the value network be?

```

```

2622         # Note: You will have to convert the label to a torch tensor to
use with torch's loss functions
2623         label = torch.tensor(data_point[2], dtype=torch.float32)
2624
2625         # TODO: Get model prediction of value
2626         prediction = model(features_tensor)
2627
2628         # TODO: Compute Loss for data point
2629         train_loss = loss_function(prediction, label)
2630         batch_loss += train_loss
2631         batch_counter += 1
2632         total_train_loss += train_loss
2633
2634         if batch_counter % batch_size == 0:
2635             # Call backward to run backward pass and compute gradients
2636             batch_loss.backward()
2637
2638             # Run gradient descent step with optimizer
2639             optimizer.step()
2640
2641             # Reset gradient for next batch
2642             optimizer.zero_grad()
2643
2644             batch_loss = 0
2645
2646         total_test_loss = 0
2647         with torch.no_grad():
2648             for data_point in test_dataset:
2649                 state = data_point[0]
2650                 features = get_features(state)
2651                 features_tensor = torch.tensor(features, dtype=torch.float32)
2652                 label = torch.tensor(data_point[2], dtype=torch.float32)
2653
2654                 prediction = model(features_tensor)
2655                 test_loss = loss_function(prediction, label)
2656                 total_test_loss += test_loss
2657
2658         avg_train_loss = total_train_loss / len(train_dataset)
2659         avg_test_loss = total_test_loss / len(test_dataset)
2660
2661         print(f'Epoch {epoch+1}/{num_epochs}:')
2662         print(f'  Training Loss: {avg_train_loss:.4f}')
2663         print(f'  Testing Loss: {avg_test_loss:.4f}')
2664
2665         return model
2666
2667 value_model = train_value_network(dataset_5x5, 10, 1e-4)
2668 save_model("value_model.pt", value_model)
2669
2670 # %% [markdown]
2671 # ## Comparing Learned Value function against other Agents

```



```

2672
2673 # %%
2674 class GoProblemLearnedHeuristic(GoProblem):
2675     def __init__(self, model=None, state=None):
2676         super().__init__(state=state)
2677         self.model = model
2678
2679     def __call__(self, model=None):
2680         """
2681         Use the model to compute a heuristic value for a given state.
2682         """
2683         return self
2684
2685     def encoding(self, state):
2686         """
2687         Get encoding of state (convert state to features)
2688         Note, this may call get_features() from Task 1.
2689
2690         Input:
2691             state: GoState to encode into a fixed size list of features
2692         Output:
2693             features: list of features
2694         """
2695         # TODO: get encoding of state (convert state to features)
2696
2697         return get_features(state)
2698
2699     def heuristic(self, state, player_index):
2700         """
2701         Return heuristic (value) of current state
2702
2703         Input:
2704             state: GoState to encode into a fixed size list of features
2705             player_index: index of player to evaluate heuristic for
2706         Output:
2707             value: heuristic (value) of current state
2708         """
2709         # TODO: Compute heuristic (value) of current state
2710         value = 0
2711
2712         features = self.encoding(state)
2713         features_tensor = torch.tensor(features, dtype=torch.float32)
2714
2715         with torch.no_grad():
2716             value = self.model(features_tensor)
2717
2718         '''value = max(-1, min(1, value))
2719         if player_index != state.player_to_move():
2720             value = -value'''
2721
2722         # Note, your agent may perform better if you force it not to pass

```

```

2723         # (i.e., don't select action #25 on a 5x5 board unless necessary)
2724         return value
2725
2726     def __str__(self) → str:
2727         return "Learned Heuristic"
2728
2729 import go_utils
2730 def create_value_agent_from_model():
2731     """
2732     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.
2733     """
2734
2735     model_path = "value_model.pt"
2736     # TODO: Update number of features for your own encoding size
2737
2738     feature_size = len(get_features(dataset_5x5[0][0]))
2739
2740     model = load_model(model_path, ValueNetwork(feature_size))
2741
2742     heuristic_search_problem = GoProblemLearnedHeuristic(model)
2743
2744     # TODO: Try with other heuristic agents (IDS/AB/Minimax)
2745     learned_agent = GreedyAgent(heuristic_search_problem)
2746
2747     return learned_agent
2748
2749 # learned_agent = create_value_agent_from_model(value_net)
2750 learned_agent = create_value_agent_from_model()
2751 agent2 = GreedyAgent(GoProblemSimpleHeuristic)
2752 print("Greedy Agent", agent2)
2753 print("Learned Agent", learned_agent)
2754
2755 run_many(learned_agent, GreedyAgent(), 40)
2756
2757 # %% [markdown]
2758 # # Task 3: Supervised Learning of a Policy Network
2759
2760 # %%
2761 class PolicyNetwork(nn.Module):
2762     def __init__(self, input_size, board_size=5):
2763         super(PolicyNetwork, self).__init__()
2764
2765         # TODO: What should the output size of the Policy be?
2766         self.output_size = board_size * board_size + 1
2767
2768         # TODO: Add more layers, non-linear functions, etc.
2769         self.fc1 = nn.Linear(input_size, 512)
2770         self.fc2 = nn.Linear(512, 128)
2771         self.fc3 = nn.Linear(128, 64)

```

```

2772         self.fc4 = nn.Linear(64, self.output_size)
2773
2774         self.relu = nn.ReLU()
2775
2776     def forward(self, x):
2777         # TODO: Update as more layers are added
2778         z1 = self.fc1(x)
2779         a1 = self.relu(z1)
2780         z2 = self.fc2(a1)
2781         a2 = self.relu(z2)
2782         z3 = self.fc3(a2)
2783         a3 = self.relu(z3)
2784         z4 = self.fc4(a3)
2785
2786         return z4
2787
2788 # %%
2789 # This will not produce meaningful outputs until trained, but you can test for
2790 # syntax errors
2791 features_tensor = torch.Tensor(features)
2792 policy_net = PolicyNetwork(len(features))
2793 print("Predicted Action Probabilities", policy_net(features_tensor))
2794
2795 # %%
2796 def train_policy_network(dataset, num_epochs, learning_rate):
2797     """
2798     Train a policy network on the provided dataset.
2799
2800     Input:
2801         dataset: list of (state, action, result) tuples
2802         num_epochs: number of epochs to train for
2803         learning_rate: learning rate for gradient descent
2804     Output:
2805         model: trained model
2806     """
2807     random.shuffle(dataset)
2808
2809     input_size = len(get_features(dataset[0][0]))
2810     # input_size = 100
2811
2812     # TODO: Create model
2813     model = PolicyNetwork(input_size, 5)
2814     print(f"Output size: {model.output_size}")
2815
2816     # TODO: Specify Loss Function
2817     loss_function = nn.CrossEntropyLoss()
2818
2819     # You can use Adam, which is stochastic gradient descent with ADaptive
2820     # Momentum
2821     optimizer = optim.Adam(model.parameters(), lr=learning_rate)

```

```

2821
2822     batch_size = 32
2823     batch_loss = 0
2824     batch_counter = 0
2825
2826     for epoch in range(num_epochs):
2827         total_train_loss = 0
2828         num_train_correct = 0
2829
2830         for data_point in dataset:
2831             # TODO: Get features from state and convert features to torch
2832             tensor      state = data_point[0]
2833
2834             features = get_features(state)
2835             features_tensor = torch.tensor(features, dtype=torch.float32)
2836
2837             # TODO: What should the desired output of the value network be?
2838             # Note: You will have to convert the label to a torch tensor to
2839             use with torch's loss functions
2840             action = data_point[1]
2841             label = torch.tensor(action, dtype=torch.long)
2842
2843             # TODO: Get model estimate of value
2844             prediction = model(features_tensor)
2845
2846             # TODO: Compute Loss for data point
2847             train_loss = loss_function(prediction, label)
2848             batch_loss += train_loss
2849             batch_counter += 1
2850             total_train_loss += train_loss
2851
2852             if batch_counter % batch_size == 0:
2853                 # Call backward to run backward pass and compute gradients
2854                 batch_loss.backward()
2855
2856                 # Run gradient descent step with optimizer
2857                 optimizer.step()
2858
2859                 optimizer.zero_grad()
2860
2861                 batch_loss = 0
2862
2863         train_accuracy = num_train_correct / len(dataset)
2864         avg_train_loss = total_train_loss / len(dataset)
2865
2866         print(f'Epoch {epoch+1}/{num_epochs}:')
2867         print(f'  Training Loss: {avg_train_loss:.4f}')
2868
2869     # torch.save(model.state_dict(), "policy_model.pt")

```

```

2870     return model
2871
2872 policy_net = train_policy_network(dataset_5x5, 10, 1e-4)
2873 save_model("policy_model.pt", policy_net)
2874
2875 # %% [markdown]
2876 # ## Comparing Learned Policy against other Agents
2877
2878 # %%
2879 class PolicyAgent(GameAgent):
2880     def __init__(self, search_problem, model_path, board_size=5):
2881         super().__init__()
2882         self.search_problem = search_problem
2883         # self.model = load_model(model_path, PolicyNetwork)
2884
2885         input_size = len(get_features(dataset_5x5[0][0]))
2886         model_template = PolicyNetwork(input_size, board_size)
2887         self.model = load_model(model_path, model_template)
2888
2889         self.board_size = board_size
2890
2891     def encoding(self, state):
2892         # TODO: get encoding of state (convert state to features)
2893         return get_features(state)
2894
2895     def get_move(self, game_state, time_limit=1):
2896         """
2897         Get best action for current state using self.model
2898
2899         Input:
2900             game_state: current state of the game
2901             time_limit: time limit for search (This won't be used in this agent)
2902         Output:
2903             action: best action to take
2904         """
2905         legal_actions = self.search_problem.get_available_actions(game_state)
2906
2907         features = self.encoding(game_state)
2908         features_tensor = torch.tensor(features,
dtype=torch.float32).unsqueeze(0)
2909
2910         with torch.no_grad():
2911             action_logits = self.model(features_tensor)
2912             action_probs = torch.softmax(action_logits, dim=1).squeeze(0)
2913
2914         all_probs = action_probs.tolist()
2915
2916         # Get probabilities for legal actions
2917         legal_actions_probs = [
2918             (action, all_probs[action-1] if 1 ≤ action ≤ len(all_probs) else
0)

```

```

2919         for action in legal_actions
2920     ]
2921
2922     # Sort legal actions by probability
2923     sorted_legal_actions = sorted(legal_actions_probs, key=lambda x: x[1],
reverse=True)
2924
2925     # Return best legal action
2926     return sorted_legal_actions[0][0] if sorted_legal_actions else None
2927
2928     def __str__(self) → str:
2929         return "Policy Agent"
2930
2931 def create_policy_agent_from_model():
2932     """
2933     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.
2934     """
2935
2936     model_path = "policy_model.pt"
2937     agent = PolicyAgent(GoProblem(size=5), model_path)
2938     return agent
2939
2940 # %%
2941 # policy_agent = PolicyAgent(GoProblem(size=5), policy_net)
2942 policy_agent = create_policy_agent_from_model()
2943 print("Policy Agent", policy_agent)
2944 run_many(policy_agent, GreedyAgent(), 40)
2945
2946 # %% [markdown]
2947 # # Submitting
2948 #
2949 # 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.
2950
2951
2952

```