Pokemon_Battle_PokeBob_team

January 22, 2025

The project focuses on the track proposed in the course, specifically the section related to competitions. Our team is named **PokeBob** and is formed by **Gemma Ragadini** and **Filippo Alessandro Sandoval Villarreal**. We selected the task involving the simulation of a Pokémon battle between two teams, each composed of three Pokémon. The battle consists of three matches, with the first player to knock out all three Pokémon of the opposing player declared the winner of the match. The battle is considered concluded when a player wins at least two out of three matches. The objective of the project was to develop a competitive AI agent in the Pokémon battle environment. We started by challenging a random player who selects its Pokémon moves arbitrarily without any strategic logic. Both players operate under the same conditions, with their teams assigned randomly and regenerated before each challenge, ensuring that any advantages or disadvantages are also determined by chance. In the following sections, we will explain our approach to solving the task and outline the methodology we adopted, along with the results we obtained.

The repository is available at this link: https://github.com/GemmaRagadini/Pokemon_AIF_24_25.git

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

```
!python3 -m venv amb
!source amb/bin/activate
```

Related Work

For this project, we decided to explore some existing approaches related to the concepts studied during the course, as well as to develop a custom approach of our own. These approaches were evaluated through a tournament, the details of which will be provided in the corresponding section.

The related work was derived from the course slides and the accompanying textbook, *Artificial Intelligence A Modern Approach - Fourth Edition. Stuart Russer, Peter Norvig.* Specifically, we focused on the section related to game theory (Chapter 6 of the textbook) and examined various approaches to the Minimax algorithm and its variations. This included the implementation of alpha-beta pruning and heuristics such as the killer move heuristic.

Methodology

To achieve our goal, we decided to implement several algorithms discussed in the related work. We focused our attention on various implementations of the Minimax algorithm using 2 different evaluation functions. In the following sections, we will provide a detailed explanation of each algorithm we implemented.

```
import utils
from utils import *
```

Implemented Policies

We implemented 3 algorithms: classic Minimax, Minimax with alpha-beta pruning and killer heuristic, and My Policy. The third one is the policy that performed the best. The first two were tested with two different evaluation functions, `game_state_eval` and `my_eval_fun`.

The two Minimax algorithms are located in the file `behaviour/otherPolicies.py`, while `MyPolicy` is located in the file `behaviour/myPolicy.py`.

Minimax

The first implementation is the one obout a classic minimax algorithm with a pre-existent evaluation function called `game_eval`. Here below there is our implementation. Minimax policy employs `game_state_eval` or `my_eval_fun` evaluation functions (described in the *Evaluation Functions* section). The Minimax implementation is straightforward, comprising a section for the maximizer player and another for the minimizer. The maximizer aims to transition to states where its Pokémon are healthier than the opponent's Pokémon, while the minimizer seeks to reduce this advantage. Each state is evaluated recursively. To support these computations, the algorithm uses the `n_defeated` function (in `behaviour/evalFunctions.py`, again in *Evaluation Functions* section), which counts the number of fainted (knocked-out) Pokémon. Additionally, the algorithm determines the next action from the maximizer player's perspective, as implemented in the `get_action` method.The default values for the search depth and weights in the evaluation function were determined empirically. Various configurations were tested, and the ones used here were found to deliver the best performance according to our evaluation metrics.

Minimax with Alpha-Beta Pruning and Killer Move Heuristic

Our second implementation extends the basic Minimax algorithm by incorporating alpha-beta pruning and the killer move heuristic. This implementation was developed to enhance both the performance and efficiency of the Minimax algorithm described above. The addition of alpha-beta pruning allows the algorithm to eliminate branches in the search tree that cannot influence the final decision, significantly reducing the number of nodes explored. Meanwhile, the killer move heuristic prioritizes moves that are likely to be effective, further optimizing the decision-making process by focusing on promising actions. The combined use of these techniques aims to not only improve the accuracy of the algorithm but also speed up its execution, enabling faster and more effective decision-making.

Evaluation Functions

The evaluation function plays a central role in determining the quality of a game state during a Pokémon battle simulation. The purpose of the evaluation function is to return a numerical score that represents the desirability of the current game state. Higher scores indicate more favorable conditions for the agent, while lower scores highlight disadvantages.

`game_state_eval` is the pre-existent evaluation function, it encourages states where the player's active Pokémon (mine) has higher HP relative to its maximum HP and penalizes states where the opponent's active Pokémon (opp) has high HP. Finally it adds a penalty proportional to the search depth to prioritize faster victories. Our results demonstrate that it provides a balanced implementation. However, it is not the most effective approach we encountered. To try to improve the performance of the algorithms, we implemented another evaluation function, `my_eval_fun` that builds upon the previous one. The function achieves this by evaluating three core aspects: Type

Compatibility, Health Points (HP) Analysis and Damage Potential. These values are combined into a single score using weighted contributions.

Custom Policy

The third and last approch was the one about a our custom Policy, implemented in `behaviour/myPolicy.py`.

```
class MyPolicy(BattlePolicy):
    def __init__(self):
        self.hail_used = False
        self.sandstorm_used = False
        self.name = "My Policy"
    def assess_damages(self, active_pkm: Pkm, opp_pkm_type: PkmType,␣
 ↪attack_stage: int, defense_stage: int, weather: WeatherCondition)-> int:
        # moves evaluation
        damages: List[float] = []
        for move in active_pkm.moves:
            damages.append(evalFunctions.estimate_damage(move.type, active_pkm.
 ↪type, move.power, opp_pkm_type, attack_stage,defense_stage, weather))
        return damages
    def get_action(self, g: GameState) -> int:
        # my team
        my_team = g.teams[0]
        active_pkm = my_team.active
        bench = my_team.party
        my_attack_stage = my_team.stage[PkmStat.ATTACK]
        # opposite team
        opp_team = g.teams[1]
        opp_active_pkm = opp_team.active
        opp_defense_stage = opp_team.stage[PkmStat.DEFENSE]
        # weather
        weather = g.weather.condition
        try:
            # estimate of the damage of each move
            damages = self.assess_damages(active_pkm, opp_active_pkm.type,␣
 ↪my_attack_stage, opp_defense_stage, weather)
            move_id = int(np.argmax(damages))
        except Exception as e:
            import traceback
            traceback.print_exc()
        # if it eliminates the opponent or the move type is super effective,␣
 ↪use it immediately
        if (damages[move_id] >= opp_active_pkm.hp) or (damages[move_id] > 0 and␣
 ↪TYPE_CHART_MULTIPLIER[active_pkm.moves[move_id].type][opp_active_pkm.type]␣
 ↪== 2.0) :
            return move_id
        try:
            defense_type_multiplier = evalFunctions.examine_matchup(active_pkm.
↪type, opp_active_pkm.type, list(map(lambda m: m.type, opp_active_pkm.moves)))
        except Exception as e:
            import traceback
            traceback.print_exc()
        if defense_type_multiplier <= 1.0:
            return move_id
        # Consider the Pokémon switch
        matchup: List[float] = []
        not_fainted = False
        try:
            for j in range(len(bench)):
                if bench[j].hp == 0.0:
                    matchup.append(0.0)
                else:
                    not_fainted = True
                    matchup.append(
                        evalFunctions.examine_matchup(bench[j].type,␣
↪opp_active_pkm.type, list(map(lambda m: m.type, bench[j].moves))))
            best_switch_matchup = int(np.max(matchup))
            best_switch = np.argmax(matchup)
            current_matchup = evalFunctions.examine_matchup(active_pkm.type,␣
↪opp_active_pkm.type,list(map(lambda m: m.type, active_pkm.moves)))
        except Exception as e:
            import traceback
            traceback.print_exc()
        if not_fainted and best_switch_matchup >= current_matchup+1:
            return best_switch + 4
        return move_id
```

The policy is designed to make decisions about the actions the player's team should take based on an evaluation of damage, the Pokémon's types, and various other battle conditions, such as weather. Here's an overview of its functionality:

Initialization: The class initializes two flags, hail_used and sandstorm_used, to track if certain weather conditions have already been activated during the battle.

Damage Assessment (`assess_damages`): This method evaluates the potential damage of each move available to the active Pokémon, using `evalFunctions.estimate_damage`. It calculates the damage based on various factors such as:

- The Pokémon's attack and the opponent's defense stages.
- The Pokémon types and move types.
- Weather conditions that might affect damage output.

It then returns a list of damage estimates for all available moves.

Action Selection (`get_action`): This is the primary method used to decide the next action. It follows a series of steps to make the decision:

- Team Setup: It first extracts the active Pokémon from the player's team and the opponent's team.
- Weather Condition: It retrieves the current weather condition, which can influence the effectiveness of moves.
- Damage Calculation: Using the `assess_damages` method, it calculates the potential damage for each move and selects the move with the highest estimated damage.

Move Selection Logic: If a move will eliminate the opponent or if it is "super effective" (based on the type chart), it is selected immediately. If the active Pokémon is at a disadvantage in terms of move effectiveness, the policy evaluates to switch to a Pokémon from the bench (the reserve Pokémon) that has a better matchup against the opponent's active Pokémon. This is determined by evaluating the compatibility between each Pokémon on the bench and the opponent's active Pokémon. The policy uses a comparison between the matchups of the active Pokémon and each bench Pokémon, selecting the Pokémon that has a significantly better type advantage.

Pokémon Switch Consideration: If there is at least one Pokémon on the bench that has a favorable matchup compared to the active Pokémon, the policy will switch to that Pokémon. This is done by comparing the matchup values, and if the bench Pokémon's matchup score is sufficiently better (greater than the current active Pokémon's matchup by at least 1), it will choose to switch to that Pokémon. When switching Pokémon, "time" is lost in battle, so the policy chooses to do so only if the gain is considerable.

The policy uses the `evaluate_matchup` function to assess the compatibility between Pokémon types based on their moves, making it a dynamic policy that adapts to the strengths and weaknesses of the opposing team.

This battle policy uses a combination of damage estimation, type advantages, and Pokémon switching strategies to make optimal decisions, ensuring that the player can both maximize damage and strategically manage their team for better performance in battle.

Performance Evaluation

We implemented several approaches and first thing we decided to have each agent battle against the Random agent. This allowed us to demonstrate that each policy outperforms the Random agent. We chose to use a random team for each player, generated for every match, so that with a large number of executions, the influence of the chosen team on the evaluation of the algorithm's performance decreases. *In the code below, the functions `SingleCombat` and `Tournament`, located in the `utils.py` file, are called due to space constraints.*

For the second evaluation, we organized a tournament involving these players:

- *My Policy Player*
- *Random Player*
- *Minimax Player*
- *Minimax with Alpha-Beta Pruning and Killer Move Heuristic Player*
- *Minimax Player using `my_eval_fun`*
- *Minimax with Alpha-Beta Pruning and Killer Move Heuristic Player using `my_eval_fun`*

```
if __name__ == '__main__':
    turnament=sys.argv[-1]
    if turnament== 't':
        utils.Tournament()
    else:
        utils.main()
```

Tournament

We organized a round-robin tournament, in which each player fights against every other player and earns one point for each match won. Each confrontation consists of N matches, each of which is made up of 3 individual battles. The number of matches won is counted, and the player rankings are created. We show the results for the tournament with N = 10.

```
utils.Tournament()
```

This is the result of a previous execution in textual form, as the tournament's execution time is approximately 3 hours: - My Policy - 40 punti
- My Minimax - 28 punti
- My Minimax with my eval - 26 punti
- Minimax with pruning alpha beta killer - 24 punti
- Minimax with pruning alpha beta killer and my eval - 24 punti
- Random Player - 8 punti

As we can from the result our policy wins the tournament with the most winned matches. Also Minimax and Minimax with alpha beta pruning and killer move heuristic perform very well and they beat the minimax with our evaluation function whuch consider also the power of a move of a pokemon. In this tournament the fact that the implementation of minimax with our evaluation function dosen't perform so well can be by the fact that the team assigned to the two player (Minimax and Minimax with my eval) were to favorable to the Minimax player (even if they are selected in a random way)

Conclusion

The result from the the simulations of the battle was allineated with what we thought. In fact every player with a policy different from the random one was able to defeat the random player, not always with outstanding results but they culd beat the random player. We implemented an algorithm that wins a very high percentage of matches (almost 100%) against the random agent and, as seen from the tournament, achieves very satisfactory results against the other implemented algorithms. The difference in performance with the change of evaluation function in the other two algorithms does not seem to be very impactful, as the two functions do not have fundamental differences in their approach. The major difference is with MyPolicy, as can be seen from the tournament results.
We believe that MyPolicy performs better than the various Minimax algorithms in this context because, with few game variables and possible strategies, this allowed us to be very specific in the implementation of the algorithm, which adapts well to the very specific situations of the game. MyPolicy performs well even against some of the agents from the VGC competitions of 2023 and 2024, we have shown these results in Appendix.

Appendix

We tested the performance of MyPolicy with challenges consisting of 50 epochs against 3 agents from the 2023 and 3 from 2024 competitions. Here are the results obtained:

- MyPolicy vs MySubmissionMR-M.Ruppert: 38-12
- MyPolicy vs vgc_weiyi_yen-Wei Yi Yen: 49-1

- MyPolicy vs WiktorBukowski-Wiktor Bukowski: 35-15
- MyPolicy vs campiao-Pedro Campião: 30-20
- MyPolicy vs Bot4TeamBuildPolicy-Anja Ka: 23-27
- MyPolicy vs MyPokemon-hgvbhjvcfg gh: 34-16