Otto-von-Guericke-University Magdeburg
Faculty of Computer Science
Chair of Computational Intelligence

Computational Intelligence in Games

Pokemon VGC AI Competition 2024

Submitted by:
MUHAMMAD AHAD - 232910
MOHAMMAD TAIF ARIF SHAMSI - 224365
July 8, 2024

Contents

1 Introduction

This report presents the design and implementation of our custom Pokémon bot, developed for the course computational intelligence in games. The bot was evaluated based on various metrics. The bot can be found at [1]. The competition can be found here [2].

2 Bot Overview

2.1 General Description

Our Pokémon bot employs a combination of heuristic and deep Q-learning approaches to effectively build teams and make battle decisions. The primary goal was to create a competitive bot capable of participating in the provided framework's tournaments.

2.2 Framework and Tools Used

The bot was developed using Python, with key libraries including PyTorch for deep learning and custom modules provided by the Pokemon VGC AI framework. The following files constitute the bot's implementation:

- `IncreaseStateBattlePolicy.py`: Implements the battle policy with deep Q-learning and heuristics.
- `HeuristicBattlePolicy.py`: Implements heuristic rules for battle decisions.
- `TBPolicy.py`: Implements the heuristic team selection policy.
- `mahadAgent.py`: Integrates the battle and team-building policies into a single agent.

3 Custom Policies

3.1 Team Build Policy

3.1.1 Heuristic Team Selection

The heuristic team selection policy, implemented in `TBPolicy.py`, aims to select a balanced and diverse team. The `MaxPowerMovesTeamBuildPolicy` class defines the team-building strategy, which involves selecting Pokémon based on HP and move power.

- `set_roster`: Initializes the roster.
- `get_action`: Returns a selected team based on the heuristic approach.

- `evaluate_pokemon`: Evaluates Pokémon based on their HP and move power.
- `heuristic_team_selection`: Selects the team using the evaluated scores.

3.2 Battle Policy

Our battle policy leverages a combination of Deep Q-Learning and heuristic-based decision-making to optimize the bot's performance. The integration of these approaches ensures a balance between learning from experience and applying domain-specific knowledge.

3.2.1 Deep Q-Learning Approach

The Deep Q-Learning (DQN) approach is implemented in `IncreasedStateBattlePolicy.py`. This policy uses a neural network to predict optimal actions based on the current game state.

Inputs The input to the DQN consists of a 32-dimensional feature vector extracted from the game state. These features include:

- Current weather condition.
- Change in HP of the active Pokémon.
- Types of the active and opposing Pokémon.
- Status changes of both active and opposing Pokémon.
- Types and power of the moves available to the active Pokémon and its party members.

Outputs The output of the DQN is a set of Q-values corresponding to the possible actions the bot can take. These actions include:

- Using one of the four available moves.
- Switching to one of the party members.

Learning Process The DQN is trained using experience replay and a target network. The key components of the learning process are:

- `ReplayMemory`: Stores past experiences as transitions $(state, action, reward, next\_state)$ to break the correlation between consecutive experiences.
- `DQN`: The neural network model that approximates the Q-function.
- `optimize_model`: Samples a batch of transitions from replay memory and updates the network parameters using the loss between predicted and target Q-values.

- `select_action`: Implements an epsilon-greedy policy to balance exploration and exploitation.

Implementation Details

- `get_state`: Extracts and returns the feature vector from the game state.
- `get_action`: Determines the best action to take based on the current state and Q-values.
- `select_action`: Chooses an action based on the epsilon-greedy strategy.
- `calculate_reward`: Computes the reward for transitioning from one state to another.
- `optimize_model`: Trains the neural network using batches of experiences from replay memory.
- `save_model` and `load_model`: Saves and loads the model parameters for persistent learning.

3.2.2 Heuristic Battle Policy

The heuristic battle policy is implemented in `HeuristicBattlePolicy.py`. This policy uses predefined rules and domain knowledge to make decisions.

Decision-Making Process The heuristic policy makes decisions based on several factors:

- `calculate_damage`: Estimates the potential damage of each move considering type advantages, weather conditions, and stat stages.
- `evaluate_matchup`: Assesses the matchup between the active and opposing Pokémon based on their types and potential moves.
- `should_switch`: Determines if a switch is beneficial based on type disadvantages, low HP, or status conditions.
- `get_action`: Combines the above evaluations to select the optimal move or switch.

3.2.3 Combining DQN and Heuristic Approaches

The integration of DQN and heuristic approaches is handled in `IncreasedStateBattlePolicy.py`. The policy leverages the strengths of both methods to enhance decision-making.

Policy Selection The combined policy decides which approach to use based on the confidence of the DQN and predefined conditions:

- The DQN predicts actions and their Q-values based on the current state.
- If the DQN suggests switching (actions 4 or 5), the heuristic policy (`heuristic_action`) is consulted to confirm or override the decision.
- The heuristic rules provide fallback decisions when the DQN's confidence is low, ensuring robust performance even in unfamiliar scenarios.

Implementation Details

- `get_action`: The primary function that determines the action to take by first querying the DQN and then applying heuristic rules if necessary.
- `heuristic_action`: Evaluates the need for switching Pokémon based on heuristic conditions.

4 Implementation Details

4.1 File Descriptions

- `IncreaseStateBattlePolicy.py`: Contains the `ISDQNBattlePolicy` class, which extends `BattlePolicy`. This class uses a neural network (DQN) to determine actions based on game states and integrates heuristic rules for switching Pokémon when advantageous.
- `HeuristicBattlePolicy.py`: Contains the `HeuristicBattlePolicy` class, providing heuristic-based decision making for battles. It calculates damage, evaluates matchups, and determines optimal moves and switches.
- `TBPolicy.py`: Contains the `MaxPowerMovesTeamBuildPolicy` class, which selects a team based on the power of Pokémon's moves and their HP. It uses functions like `evaluate_pokemon` and `heuristic_team_selection`.
- `mahadAgent.py`: Defines the `MahadAgent` class, integrating the team-building and battle policies. It sets up the agent's name and policies and serves as the main competitor class.

5 Testing and Evaluation

5.1 Tournament Participation

We participated in the Friday and Tuesday tournaments to test our bot's performance and make iterative improvements. We also tested our bot against the championship winners of the year 2023.

5.2 Performance Metrics

The bot's performance was evaluated using metrics such as win rate, ELO rating and effectiveness of team composition. The figure 1 shows the training progress during one of the sessions.

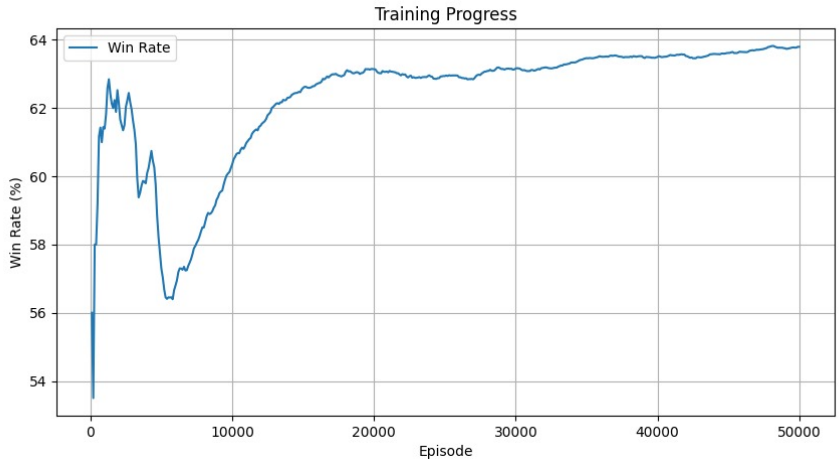

Figure 1: The figure illustrates the bot's training progress. The y-axis represents the win rate against randomly selected championship bots from the year 2023, while the x-axis indicates the number of battles the bot was trained on.

5.3 Results

Our bot showed consistent improvement in performance through during the testing and training processes, reflecting the effectiveness of our combined heuristic and deep Q-learning approaches.

6 Challenges

Throughout the development and training of our Pokémon bot, we encountered several challenges:

6.1 Training Time and Computational Resources

Whenever we wanted to train our reinforcement learning deep Q-learning approach, we had to train it for a minimum of 50,000 battles. This required approximately 8 hours, with an average rate of 6,250 battles per hour on our current limited hardware capabilities. The significant time required for training posed a major constraint, limiting the frequency of testing and iteration.

6.2 Hyperparameter Optimization

Optimal hyperparameters for training were difficult to obtain due to the extensive time required for each training cycle. This made it challenging to experiment with different hyperparameter configurations and identify the most effective settings.

6.3 Balancing Exploration and Exploitation

The epsilon-greedy strategy used in our DQN required careful tuning to balance exploration of new strategies and exploitation of known successful actions. Achieving this balance was critical for effective learning but proved to be challenging.

6.4 Integration of Heuristic and Learning Approaches

Combining heuristic rules with the DQN-based approach required extensive testing to ensure that the integration did not compromise the performance of either method. Determining when to rely on heuristic decisions versus learned policies was particularly challenging.

6.5 Testing and Validation

Given the complex nature of Pokémon battles and the stochastic elements involved, testing and validating the bot's performance required extensive simulations. Ensuring consistent performance across different tournaments and scenarios demanded significant effort and resources.

Despite these challenges, the combination of heuristic and deep Q-learning approaches allowed us to develop a competitive and robust Pokémon bot.

7 Conclusion

In this report, we detailed our approach to developing a competitive Pokémon bot. The integration of heuristic and deep Q-learning strategies proved to be effective, balancing com-

plexity and performance. The project provided valuable insights into both reinforcement learning and heuristic-based decision-making.

References

[1] Muhammad Ahad and Mohammad Taif Arif Shamsi. Pokemon reinforcement learning bot. `https://github.com/taifarif94/pokemon_reinforcement_learning_bot`, 2024. Accessed: 2024-07-08.

[2] Simão Reis, Luís Paulo Reis, and Nuno Lau. Vgc ai competition - a new model of meta-game balance ai competition. In *2021 IEEE Conference on Games (CoG)*, pages 01–08, 2021.