Optimal Battle Strategy in Pokémon using Reinforcement Learning

Akshay Kalose, Kris Kaya, and Alvin Kim

Abstract— **Pokémon is a turn based video game where players send out their Pokémon to battle against the opponents Pokémon one at a time. Our project attempts to find an optimal battle strategy for the game utilizing a model-free Reinforcement Learning strategy. We found that a softmax exploration strategy with Q-Learning resulted in the best performance after qualitatively and quantitatively, using the win rate against a random agent, evaluating it against other approaches.**

Introduction

Pokémon is a popular video game franchise where players play as a trainer who owns monsters called Pokémon. Players can battle other trainers by having their Pokémon fight in a turn-based combat system. The game of Pokémon has evolved in major ways over the years, with each new iteration of the game making the game more and more complex. The first Pokémon game featured 151 unique Pokémon, but now there exists over 800 of them.

Pokémon battles contain an unique blend of strategy, domain knowledge, and luck that make them well-regarded amongst the video game community. In addition, due to the extremely large amount of both Pokémon and moves, there exists an incredible amount of variety to the battles.

We were interested in exploring this battle space by using reinforcement learning to create an agent that can optimally play Pokémon battles.

Motivation

The battling aspect of Pokémon is so popular that there is a relatively large competitive scene. Countless databases, forums, and other online resources exist to give players the information needed to increase battling skill. Moreover, there are sanctioned competitive battle tournaments in real life where the winners receive cash prizes. There also exists a popular battle simulator website called Pokémon Showdown, where play can create teams of Pokémon and battle others on the internet. Given the widespread popularity of Pokémon battling, we wished to further explore the competitive scene by developing a successful agent.

Additionally, we were interested in the projects applications in general game-playing. Attempts to find optimal strategies for various games such as chess or Go are common throughout the literature. Yet, given that Pokémon is a video game and that it has an immense state space, comparatively less research has been put into creating an optimal battle agent. Therefore, we were interested in seeing if it was indeed possible to create an AI agent for Pokémon that was able to match or even exceed human performance levels. By using game-playing algorithms explored in other games, we hoped to find the existing strategy that would have the best performance when applied to Pokémon battling.

Problem Definition

Our project attempts to find an optimal battle strategy in Pokémon. As mentioned before, a battle is a turn based game where players send out their Pokémon to battle against the opponents Pokémon one at a time. Each player has a team of [1,6] Pokémon, each with its own set of uniquely valued stats such as health, physical attack, special attack, physical defense, special defense, and speed which determine its strength and survivability. They each also have up to 4 unique moves which serve as the potential actions that the Pokémon can take during battle. Most moves function to reduce the health of the opponent Pokémon. However, some moves inflict special conditions (known as status conditions or status effects) on the opponent's Pokémon such as poison and sleep, which negatively impact a Pokémon in more subtle ways than having it just lose health. For example, the poison ailment reduces a Pokémon's health by $\frac{1}{16}$ every turn. Additionally, some moves can apply a stat multiplier to either Pokémon. For example, there is a move called Swords Dance which doubles the attack stat of the user.

We define a battle strategy as an ordered sequence of moves to be performed by a given user's Pokémon. An optimal strategy would be the sequence of moves that maximizes the probability of a given user winning the Pokémon battle. The winning player of a battle is the one who makes the other players Pokémon lose all their health while having at least one Pokémon on their team that is still alive. We sought to implement an AI agent whose goal was to defeat opponents by reducing all of the opponents Pokémon to zero health while maximally retaining the health of their own Pokémon.

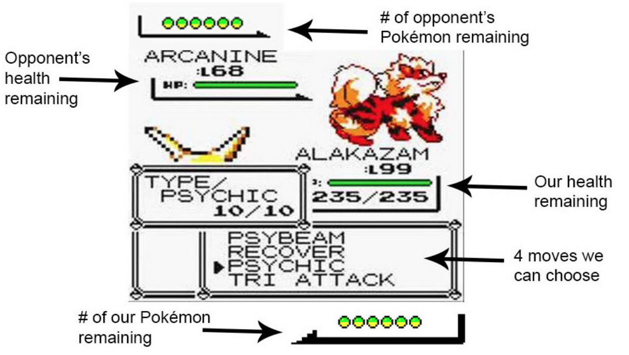

Fig. 1. An example of the key components in a Pokémon battle

Literature Review

Attempts to create an optimal Pokémon battler have been prevalent both at Stanford University and at other academic institutions. Students in advanced undergraduate classes at Stanford such as CS 221 and CS 229 have attempted to create AI agents to optimally play Pokémon Showdown, an online Pokémon battle simulator. A paper by Khosla, Lin, and Qi [1] used an expectimax AI with an evaluation function trained from 20,000 replays using a TD Lambda Learning strategy and was able to achieve an ELO rating of 1344. This indicates the bot was able to achieve a level of play equal to an average human. Moreover, a CS 221 project done by Ho and Ramesh [2] implemented a minimax agent (of search depth 2) with branch and bound, move ordering, and evaluation function that took into account speed and whether the current Pokémon could do super effective damage. This bot was able to achieve an ELO rating of 1270.

Beyond the above studies, other approaches to Pokémon AI have been undertaken by researchers outside Stanford. Panumate and Iida [3] tested four Pokémon AIs (Random AI, Attack AI, Smart-Attack AI and Smart-Defense AI) that each used a different strategy for the purposes of game refinement. They found that the four agents were optimal for play against agents of different skill levels, which reflects that the differing strategies influenced the agents' performance. Additionally, Lee and Togelius [7] found that a Pruned BFS agent, a Minimax agent, and a One Turn Look Ahead agent were able to outperform other agents such as a standard BFS. This study compared the performance of the AIs by having them battle against each other.

Challenges

There are a few challenges that make it difficult to create an intelligent agent that can win Pokémon battles. First of all, battles have an extremely large state space. There are over 800 Pokémon and over 700 potential moves. As a result, there is an enormous number of possible teams of Pokémon. In addition, while most moves purely do damage, there are also moves that have additional effects such as inflicting status condition or altering base stats. Whereas a naive agent would solely choose moves that do the most damage, a more intelligent player knows how to properly utilize these secondary types of moves in order to win. The presence of these secondary moves indicates the need for a strategy beyond simple damage maximization.

To combat the issue of a large state space, we decided to limit the amount of Pokémon and moves to just the first generation. This change meant only 151 Pokémon and 165 moves were available. The additional effects were a larger challenge but we will discuss later in our approach how we accounted for them.

In addition, there was the logistical challenge of modelling the game state and performing the necessary logic and damage calculations needed in a battle. Pokémon battles have surprisingly large amount of unique moves and special conditions that we had to account for.

Implementation

Due to the complexity of a Pokémon battle, we decided to simplify the process of battling and implemented a deterministic Pokémon battle simulator. Making our simulator deterministic allowed us to avoid some uncertainty involved in Pokémon battles such as the accuracy of a move or the chance of inflicting a status effect. As such, we were able to focus more on finding an optimized strategy.

We built our simulator from scratch in Python using the data from veekun's pokedex [5], and pokeapi.co [6]. This simulator supports all first generation Pokémon except Ditto and most first generation moves.

A. Data Structure

The game of Pokémon has a lot of data in its game state, which required us to structure our data very deliberately. There are essentially two main data structures; a data structure to represent each Pokémon and then a data structure to represent each potential move.

Pokémon

ID:	Index in list of Pokémon
Stats:	List of integers representing base stat values for HP, Attack, Defense, Special Attack, Special Defense, Speed
Level:	Level of Pokémon
Stat Stages:	List of integers ranging from -6 to 6 that determine stat stages
Type:	Indexes of types associated with the Pokémon
Current HP:	Current health points of the Pokémon
Moves:	List of up to 4 indexes of moves

Move

ID:	Index of list of moves
Type:	Index of type of the move
Power:	Integer used in damage calculation
Priority:	Integer that determines who attacks first (Overrides speed)
Target:	Integer determining who is affected by the move
Damage Class:	Integer determining whether move is status, physical, or special
Meta:	Information of status effects and stat stage modifications.

B. Damage Calculation

$$Damage = \left(\frac{(\frac{2 \times Level}{5} + 2) \times Power \times A/D}{50} + 2 \right) \times Modifier$$

$$Modifier = TypeModifier \times SameTypeAttackBonus$$

$$A = \text{(Special) Attack stat of attacking Pokémon}$$

$$D = \text{(Special) Defense stat of defending Pokémon}$$

The above equation is the formula utilized by the game and our simulator to calculate the amount of damage inflicted on the opponent when a Pokémon performs a damaging move. It takes into account the Pokémon's stats including level, Attack, and Defense as well as the Power of the move and any relevant modifiers.

The type modifier is calculated using the type of the move being used and the types of the defending Pokémon. The same type attack bonus is equal to 1.5 if the defending Pokémon shares a type with the attacking move's type, otherwise is equal to 1.

C. Stat Stage Multiplers

Every Pokémon stat has a stat staged value. Depending on the value, there is a multiplier applied to the stat as shown in the table below. Certain moves can manipulate the stat multiplier of the agent's Pokémon, the opponent's Pokémon, or both.

-6	-5	-4	-3	-2	-1
$\frac{25}{100}$	$\frac{28}{100}$	$\frac{33}{100}$	$\frac{40}{100}$	$\frac{50}{100}$	$\frac{66}{100}$

0	1	2	3	4	5	6
$\frac{100}{100}$	$\frac{150}{100}$	$\frac{200}{100}$	$\frac{250}{100}$	$\frac{300}{100}$	$\frac{350}{100}$	$\frac{400}{100}$

Considering Approaches

Multiple approaches exist in trying to build an AI to play Pokémon battles including both model-based and model-free as seen in the Literature Review. Since most of the AI's available are model-based and use game trees to perform some form of search such as minimax or expectimax, we explore model-free approaches to optimally win battles. One feature of model-free approaches that we found advantageous was less computation and more efficient data usage.

Initial Approach

In a model-free approach to reinforcement learning, we don't need to define transition and reward functions. These will be approximated by the learning algorithm. One of the most popular model-free learning algorithms is Q-learning, which attributes Q values to state and action pairs. In the context of our problem, given a current game state, we will have Q values associated with each of the user's Pokémon moves. The trained AI would then select the move with the highest Q value. We implement the Q-learning algorithm, as shown below. We use $\alpha = 0.10$ and $\gamma = 0.95$.

Algorithm 1 Q-learning algorithm as described in [4]

```
function QLearning
    $t \leftarrow 0$
    $s_0 \leftarrow$ initial state
    Initialize Q
    loop
        Choose action $a_t$ based on Q and some exploration strategy
        Observe new state $s_{t+1}$ and reward $r_t$
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
        $t \leftarrow$ t + 1
```

We decided to choose an epsilon-greedy strategy as our initial exploration strategy. We chose epsilon to be 0.10 so that we would choose the optimal move based off our Q-values 90% of the time and a random move the other 10%. Therefore, we would not be stuck in a local optima yet still could explore enough of the state space.

The next decision point was what to include in our state vector. Pokémon battles have an extremely high number of potential features that would be useful to keep track of. First of all, we could keep track of features of the general game state such as how many Pokémon each trainer has remaining. Then we could keep track of the specific Pokémon on the field. These features would include the type of the Pokémon, how much health each one has remaining, and the stats of both Pokémon. There are then additional features such as whether either Pokémon are currently inflicted with a status effect. Beyond the Pokémon that are currently on the field, each player has other Pokémon on their teams, whose stats, moves, and ailments could be useful to consider.

For the purpose of our initial implementation of Q-learning, we decided to utilize a simple feature vector that encoded the necessary basic information. We initially planned on tracking the sum of the total health of both trainer's respective teams of Pokémon, the amount of Pokémon left on each side, and the type of both Pokémon currently on the field. When we kept track of health, we wanted to have an accurate account of remaining health while also trying to avoid having too many possible states for each health total. Therefore, we instead split the health into 10 buckets of percentages that dictated what level of health the team had left. For example, if a team only had 6% health left, it would be in the first bucket but if it had 33% it would in the fourth.

We quickly made a change however to remove the amount of Pokémon left on each side. We realized that this inclusion would exponentially grow the state space, and since in our initial implementation we did not have function approximation, this would then lead to a lot of unknown states and random behavior. We also reasoned that our agent should still yield promising results if optimizing its actions to play against just a single opponent.

Our next step was then to determine what the reward should be. We wanted to strongly reward winning and penalize losing, so we set the rewards for reaching those particular game states to be very high. We then had a intermediary reward where we compared how much health we had left to how much our opponent had. If we had more health, we would receive a small reward scaled to how much more health we had and were penalized in the same way otherwise.

D. State Definition

player_hp_bucket:	Index of player's health bucket
opponent_hp_bucket:	Index of opponent's health bucket
player_type_1:	First type of player's Pokémon
player_type_2:	Second type of player's Pokémon
opponent_type_1:	First type of opponent Pokémon
opponent_type_2:	Second type of opponent Pokémon

E. Action Definition

Each Pokémon has at most 4 unique moves that it can perform at a given state. Therefore, we defined an action as the index of the move that was performed by the Pokémon.

Initial Results

We performed self-play by having two Q-learning agents play each other. The first agent would re-evaluate its Q-values after each move and choose its moves according to the exploration strategy, while the second agent would only have its Q-values updated after each game and choose the move with the best Q-value, breaking ties randomly. We trained our agent with 5,000 games and then had our Q-learning agent play against a random agent with the fixed Q-values found after our training. We ended up with a win-rate of 60%.

Softmax Exploration

To improve our agent, we decided to utilize a softmax exploration strategy instead. With softmax, our exploration strategy will now utilize information from our previous games. We now chose an action with probability proportional to exp($\lambda\rho_i$), with λ = 1 and ρ_i is equal to the normalized distribution of Q values at the current state and possible actions.

Softmax Results

We saw improvements solely by changing the exploration strategy to softmax. Whereas before we only had a win rate of 60% against random, after implementing softmax and training our agent with 5,000 games, we saw our agent win against a random agent with a higher win-rate of 65%. We noticed softmax had also converged to the optimum faster. After only 2,000 games for training, softmax resulted in a win rate of 60% against the random agent. After training for 20,000 games, softmax resulted in a win rate of 70% against the random agent.

Results

5000 Training Games

Epsilon-greedy vs Random agent win rate	60%
Softmax vs Random agent win rate	65%

Analysis

Our simulator was designed to create completely random battles, so a group of 6 random Pokémon vs an opponent team of 6 random Pokémon. In a similar vein, each Pokémon also has a random moveset. As a result, there are a few unwinnable games and we do not expect even the perfect agent to have a 100% win rate. A win rate of 65% against random shows that our agent is at times choosing the best action but not at a frequent enough rate to more consistently win games. For comparison, when we implemented a minimax agent for our CS 221 project, we were able to achieve a winrate of around 90% against a random opponent.

Future Work

There exists a high level of other potential improvement in order to improve our win rate. A simple method of improvement would be to run more trials during our training so that we can make sure we are getting Q-values for even more states.

Another method that we attempted was to implement eligibility traces. In our current implementation, we do not implement it and therefore our large reward of winning the game is only associated with the action and state directly before. While given enough simulations this reward would still be useful, it does not give enough credit to previous action, state pairs that helped lead to a win result. However, after we implemented eligibility traces our win-rate decreased so we decided to remove it.

There also exists high room for improvement in terms of improving our state vector. We explained that we initially chose our vector to have the simplest state space as possible while still encoding the necessary information in order to win the game. However, there are far more features that we could have included that would have helped provide insight. Some of these features include whether either Pokémon was inflicted with a status condition and what the statistics such as attack and defense were for each Pokémon.

We had to simplify our state space as we did not have a form of local approximation implemented. As a result, if during our game we reached a stage that we had not trained for, then we would have been choosing a random move. Therefore we had to keep our state vector simple and did not implement the additional features as mentioned above. If we had local approximation implemented, then we would be able to both improve the state vector by adding more descriptive features but also we would in general choose better moves as even in new states we would have a general approximation of what a good action is.

Conclusion

While we did end up having positive results of having a win rate of 65% with 5,000 training games, and 70% with 20,000 training games, against a random agent, a Q-learning implementation requires significantly more effort to create an optimal player when compared to other versions such as minimax. In our final project in CS 221 we approached a similar problem but used a minimax agent instead, and that agent's win rate against random was close to around 90%. We believe that similar win rates can be attained but more work needs to be done as detailed in our future works section. Also, Q-Learning requires a lot more training to learn Q values to estimate rewards and transitions. Therefore, it makes sense that most other Pokémon related research we have found used a model-based minimax agent, as it was a simpler process and received good results.

Group Member Contribution

Akshay implemented the battle simulator from scratch using data by veekun and pokeapi. The entire group worked together to come up with agent strategies and implementation. We split the writing of the paper equally.

Acknowledgement

We would like to thank Mykel Kochenderfer and the rest of the CS 238 staff.

References

[1] Kush Khosla, Lucas Li, Calvin Qi. Artificial Intelligence for Pokémon Showdown. Stanford, CA: n.d. Web. 6 Dec. 2018.

[2] Harrison Ho, Varun Ramesh. Percymon: A Pokémon Showdown Artificial Intelligence/ Stanford, CA: 2014. Web. 6 Dec. 2018

[3] Hiroyuki Iida, Chetprayoon Panumate. Developing Pokémon AI for Finding Comfortable Settings. Aug. 2016. Web. 6 Dec. 2018

[4] Mykel J. Kochenderfer, Decision Making Under Uncertainty: Theory and Application, MIT Press, 2015.

[5] GitHub. veekun/pokedex. https://github.com/veekun/pokedex/. 7 Dec. 2018.

[6] GitHub. PokeAPI/api-data. https://github.com/PokeAPI/api-data/. 7 Dec. 2018.

[7] S. Lee and J. Togelius, ”Showdown AI competition,” 2017 IEEE Conference on Computational Intelligence and Games (CIG), New York, NY, 2017, pp. 191-198. 7 Dec. 2018