A Sophisticated Pokemon Agent

Andrew Costenoble
Northeastern University, Boston, MA

Abstract

The game series *Pokemon* presents an interesting and novel challenge in game AI for a number of reasons. However, despite a large competitive following there doesn't seem to have been any serious attempt at creating an agent to play the game at a high level. In this paper I attempted to create such an agent using approximate Q-Learning with manual feature extraction. The agent was to play against itself in at least 10,000 games in order to train the weights of the features. Then the agent would be put on the public Showdown ladder to play against real people to evaluate its performance. Future iterations of the agent would play against the first trained one to determine whether they were an improvement or not. An agent would have been considered successful if it was able to achieve an Elo rating of at least 1500 within 100 games. Unfortunately, due to time constraints, I was unable to train and evaluate the agent, so its actual performance is unknown.

1 Introduction

Pokemon is one of the longest-running and most successful video game franchises of all time. There have been 76 games published in the series, selling over 300 million copies in total [1]. So it should be no surprise that the game hosts a healthy competitive scene in addition to its more casual audience. The 2017 Pokemon World Championships had a total prizepool of $500,000 and the livestream of the final match reached almost 50,000 viewers. There's also a player-run group known as Smogon that lays out their own rules for competitive battling; to many players, these rules are the premiere way to play the game. An online battle simulator, known as Pokemon Showdown, that can enforce the Smogon ruleset will frequently see over 20,000 simultaneous users. As a frequent player myself, I've often thought about what it would take to create an artificial intelligence that could play the game at a competitive level. At first glance it seems that it should be fairly simple: battles rarely last beyond ten to twenty turns, generally there are at most 9 actions available to a player on a turn, and to a naive player the optimal action may seem obvious. However, I believe that *Pokemon* actually present a rather novel challenge for AI. At a high level, the game relies almost entirely on prediction of an opponent's actions. Unlike most turn-based strategy games, actions happen simultaneously during a turn, so a player must choose not just the best action for the current state, but the best action for what they believe their opponent's action will be. Additionally, because the games are generally quite short, a single wrong action can put the player in an unwinnable position.

2 Background

Reinforcement learning is a form of machine learning that involves an agent receiving incremental rewards as it progresses through its environment. A common application for reinforcement learning is in finding an optimal policy for Markov Decision Processes (MDP). An MDP is a problem that assumes all future states can be predicted only from the current state, and that each action has some probability of moving the problem into a given next state. MDPs are useful for a wide variety of applications, including game-playing agents for certain games. Q-Learning is one algorithm that can solve an MDP; that is, find an optimal policy for the problem. Q-Learning relies on the intuition that it is state-action pairs that matter, not just states. It calculates the value for each action in each given state, and an optimal policy will be one which chooses the highest value action in a given state. A typical Q-Learning algorithm runs for an arbitrary number of episodes, and during each repeatedly chooses some action until it encounters a terminal state, recording the reward and subsequent state at each step. It uses these to update the q-values, so that actions with a high reward or good next state will produce a higher value in the future.

Naive Q-Learning does not scale well into large problems, because as the state and action spaces expand the amount of state-action pairs required to keep track of also increases rapidly. Therefore there is a variant known as Approximate Q-Learning which is designed to perform similarly to normal Q-Learning but scales better into large problems. Whereas normal Q-Learning learns an explicit value for each state-action pair, approximate Q-Learning reduces each state-action pair to a set of features, which it combines in a function approximation with some set of weights for each feature. It is these weights that approximate Q-Learning seeks to learn. Formally, if $Q(s, a)$ represents the value for a given sate-action pair, f is a feature in F, and w is a feature in W:

$$Q(s, a) = \sum_{n=1}^{|F|} w_n f_n$$

Each weight, w_i, is also updated on each step using the following formula:

$$w_i + \alpha([r + \gamma max_{a'}(Q(s', a')] - Q(s, a)) * f_i(s, a)$$

Where α represents some learning rate, r was the reward seen for going into the current state, and γ is a discount factor. Intuitively, this equation means that higher weights will be given to features that are seen in positive states, and features that are often seen in negative states will be given lower weights. A potential pitfall of approximate Q-Learning is that it requires a high number of training episodes in order to get close to approximating the true reward and transition functions.

3 Previous Research

Despite the popularity of Pokemon, there seem to be few serious attempts at creating an AI to play it at a high level. All of the examples I found used extremely simple algorithms and were very limited in their scope. One used a straightforward decision tree, playing only the first generation of games against their own AI[1] [2]. Another was somewhat more sophisticated, using a minimax agent to play on Pokemon Showdown. But it only participated in Random Battles[2], and boasted the performance of "an average player" at an elo of 1300-1350 [3]. The most promising lead was from a group of researchers at NYU [4]. They pit several different algorithms, including Q-Learning, against each other to determine which was most successful. They found that Q-Learning did not perform very well against most of the other agents, but I believe the "toy metagame" they describe emerging from their agents was not accurate to what is observed among human players. The most successful strategy among their agents was to greedily maximize damage ouput over one or two turns, which any high-level player will tell you is easily countered.

I chose to pursue Q-Learning for this agent because Pokemon can be represented well as a MDP. There are two features of the game that make it well-suited to Q-Learing: firstly, previous decisions do not affect future turns except for the direct effects they have on the state; that is to say, a single gamestate contains all information an agent would require to make its decision. Secondly, the rewards within a game can be delayed significantly. Forgoing damage on one turn to apply a status to the opponent may pay off much more significantly in the long run. Q-Learning is able to incorporate delayed rewards because it will see states that eventually lead to victory as having a high value, as long as the reward function is designed well.

4 Methodology

Firstly, a formal definition of the problem. A Pokemon battle is a two-agent competitive environment that is: partially observable, a player cannot see any information about an opponent's team other than what pokemon are on it; stochastic, there are many random elements to the game in addition to the opponent's actions; sequential, each turn affects those after it; static, the environment does not change as a player decides what to do; and discrete, there are only ever some distinct number of possible actions an agent can take.

[1]The in-game battling AI is notoriously easy among competitive players.

[2]A format significantly shallower than most others.

4.1 Algorithms

As previously mentioned, the main algorithm used for this agent was a form of approximate Q-Learning. However, in order to perform the battles themselves, the agent had to communicate with an outside server (as the execution of a Pokemon battle is immensely complicated). This meant that all updates had to be done by communicating with the server, using the information it sent back. The point at which this was most problematic was in the implementation of the gamestate. Because the full gamestate is enormous, the server does not send a full report on each turn. Instead, it sends a series of messages informing the client about what changed between the previous state and the current one. This meant that the agent had to keep its own internal representation of the state, interpret each of the messages as it received them, and change its state accordingly. The largest downside of this implementation was that the Q-Learning and communication algorithms had to be intertwined; when the agent received a message from the server it meant that the game was progressing to another turn and therefore the agent must both update its internal state accordingly as well as perform all the Q-Learning updates.

Once those algorithms were successfully implemented together, I was able to work on actually performing the Q-Learning. The first step was extracting features from the state, which I chose to do manually using my own knowledge of the domain. This quickly proved more difficult than anticipated; there are several useful features of a static gamestate that can be extracted without excessive effort, but calculating the outcome of an action and extracting features from that was exceedingly difficult. That being said, these are the features I was able to implement at the time of writing:

1. Total damage taken by the user's team by the entry hazards currently on the field
2. Total damage taken by the opponent's team by the entry hazards that are expected to be on the field after the user's move
3. The user's active pokemon's current health
4. The opponent's ˆ
5. The user's active pokemon's current status[3], encoded as a k-hot vector
6. The opponent's ˆ
7. The current field conditions, encoded as a k-hot vector
8. The current weather condition, encoded as a one-hot vector
9. The current side conditions on the user's side, encoded as a k-hot vector
10. The current side conditions on the opponent's side, ˆ
11. The pokemon on the user's team, encoded as a 6-hot vector
12. The pokemon on the opponent's team ˆ

The next step was to create a reward function. In the situation the tandem server communication actually proved to be a small benefit, as the reward could be built up from the messages received. I chose the values for the rewards somewhat arbitrarily, based on how important I believed each action is. The following are the rewards:

- A very large reward (∞) for winning, and equally large penalty for losing
- Up to +10 points for learning what moves an opponent has, proportional to the prior probability they had that move
- +10 points for knocking out a pokemon, -10 for having a pokemon be knocked out
- Up to +10 points for dealing damage, proportional to how much was done; inversely, up to -10 points for taking damage
- 3 points for inflicting a status on the opponent, -3 for getting a status inflicted
- +1 point for curing your own status, -1 for the opponent curing their status (curing a status is generally less useful than inflicting one)
- Points equal to twice the number of stages of a stat boost, up to 8 (gain for boosting yourself, lose for the opponent getting boosted)

4.2 Training

Unfortunately, I was unable to fully integrate the agent with the server in the alotted time frame. As of writing, it is able to fully play one game

[3]See Appendix A

against itself, but is unable to issue another challenge after the game ends for reasons unclear to me. Because of this, I was unable to train and evaluate the agent. Outlined here is how those steps would have been done.
Team composition plays a hugely important role in competitive pokemon, but it was not the focus of this project. However, in order to get a wide range of training data the agent would need to see several different pokemon and different team archetypes. Therefore, I selected several valid, pre-made teams for the agent to battle with. During training there would be two active agents. The first one would issue challenges, and the second would accept. For each battle the agents would choose a team at random from those provided. The two agents would run concurrently, battling against each other until one emerged victorious (it is possible for a battle to go on indefinitely, but is extremely unlikely). Once the battle ended the first agent would issue a new challenge and the process would repeat for some number of episodes. The first iteration would run them for 10,000 episodes, as I approximate that it would take about that many battles to see most of the possible outcomes, but that is largely unfounded. The specific number of episodes can be increased if the agent is still showing improvement at that point, or decreased if it converges before then. Once this first training instance is finished, the weights of the two agents would be averaged to ensure that the experiences of both are considered, and then these weights would be passed to a third, separate agent for evaluation.

4.3 Evaluation

To evaluate the performance of the first agent, there is no better solution than to put it on a public ladder against actual players. Generally speaking, an Elo rating of 1500 is considered an above-average player, so I would consider the agent successful if it can achieve this rating within a limit of 100 battles (the number of battles is chosen arbitrarily; in theory the agent should converge quickly to its actual Elo rank but with the way the Showdown ladder works it may take some time to reach it). To evaluate changes made to the agent, it would play against that first agent (regardless of what rank it achieved) for some number of battles. The win/loss record of the new agent against the old one would provide a comparison to determine whether the new agent is an improvement or not. From then on, any new agents would play against the last best agent for comparison.

5 Conclusion

I set out to create an agent that can play Pokemon at a competitive level, as this seems to be a task few people have attempted and none have succeeded at. This was a more novel challenge that it first appeared, due to some features of the game itself. For the agent to perform well it would have to predict the actions of the opponent, an open area of AI research. Although I would not be implementing something that sophisticated, I hoped to lay a groundwork for future improvements to build upon. I chose to use approximate Q-Learning for the core of the agent, because the game of Pokemon is represented well as a Markov Decision Process. The most difficult part of the project came in implementing the agent to communicate with the server handling the game logic, and having it work in tandem with the Q-Learning algorithm. Although I was unable to actually execute the agent, I was able to implement most of the core algorithm and came very close to a full training and evaluation implementation.

5.1 Future Work

I do plan on continuing work on this program in the future; it's a topic I've been interested in for some time and I hope to at least get an actual implementation. Besides getting the agent to work, there are two large portions for future work to target. First would be a more sophisticated manual feature extractor, incorporating things like damage done by a move and planning around an opponent's expected action. Alternatively, the core algorithm could be altered to incorporate neural nets in a Deep Q-Learning solution, and attempt to extract the features programmatically. As evidenced by this project, there are a huge number of features in a pokemon state to consider; perhaps too many to extract by hand. Besides these, the next target for improvement would be a more sophisticated model to predict the opponent's actions. If an agent is able to consistently predict what its opponent will do, then it will perform significantly better than any other agent.

References

[1] *Pokemon games sold over 300 million games worldwide - WholesGame* (2017, November 24). Retrieved October 26, 2018 from https://wholesgame.com/news/pokemon-games-sold-over-300-million-games-worldwide/

[2] *Pokemon Artificial Intelligence is Offically Smarter Than You | Hackaday* (2014, May 9). Retrieved October 26, 2018 from https://hackaday.com/2014/05/19/pokemon-artificial-intelligence-is-smarter-than-you/

[3] *Pokemon Showdown bot.* (2014, Oct 22). Retrieved October 26, 2018 from https://github.com/rameshvarun/showdownbot

[4] *Showdown AI Competition.* (2017). Retrieved from: http://game.engineering.nyu.edu/wp-content/uploads/2017/02/CIG_2017_paper_87-1.pdf

A Game State

A.1 Pokemon Statuses

A pokemon has two different types of status: volatile and non-volatile. A non-volatile status can be exactly one of the following:

- Healthy
- Faint
- Poison
- Toxic
- Paralysis
- Burn
- Freeze
- Sleep

A volatile status can be any number of the following:

- Bound
- Trapped
- Confused
- Cursed
- Embargo
- Encore
- Heal Block
- Identified
- Infatuated
- Leech Seed

A.2 Field Conditions

The field has two conditions: terrain and room. The terrain is exactly one of the following:

- None

- Psychic Terrain
- Electric Terrain
- Misty Terrain
- Grassy Terrain

The field can have any number of the following rooms:

- Trick Room
- Magic Room
- Wonder Room

A.3 Weather

There can be exactly one of the following weathers active at a time:

- None
- Rain
- Sandstorm
- Hail
- Sun

A.4 Side Conditions

There are three conditions a side can have, in any combination:

- Reflect
- Light Screen
- Tailwind