Artificial Intelligence for Pokemon Showdown

Kush Khosla[1], Lucas Lin[2], Calvin Qi[3]

***Abstract*— We present a method to apply the TD-lambda learning algorithm to play competitive Pokemon. By modeling Pokemon as a state game, and using the TD-lambda learning algorithm to learn from the top players in the game, our AI was able to achieve around average human level of play.**

I. INTRODUCTION

A. Game Overview

Pokemon is one of the most popular games in the world. Most people know the basics of catching Pokemon and leveling up to make them stronger. However, unknown to most is the expansive competitive scene for Pokemon battling. There are forums, websites, damage calculators and databases all with the purpose of helping one improve their competitive Pokemon battling skills.

Pokemon battling consists of two players battling with six Pokemon each. The goal of the game is to be the first to knockout the six Pokemon on the other team first. There are various strategies to approach this. Some teams prefer to take a hyper-offensive approach, filling their team with strong attackers. Others prefer to have their team consist of very bulky Pokemon, and continuously chip damage off of the opponent in small amounts.

Today, most competitive Pokemon battling takes place on pokemonshowdown.com, an open source platform where players can create and battle using their teams. We construct an AI to play competitively with other players around the globe.

[1]K. Khosla is an undergraduate in the Math Department, Stanford University, 450 Serra Mall, Stanford, California kkhosla at stanford.edu

[2]L. Lin is an undergraduate in the Computer Science Department, Stanford University, 450 Serra Mall, Stanford, California lucaslin at stanford.edu

[3]C. Qi is an undergraduate in the Computer Science Department, Stanford University, 450 Serra Mall, Stanford, California calvinqi at stanford.edu

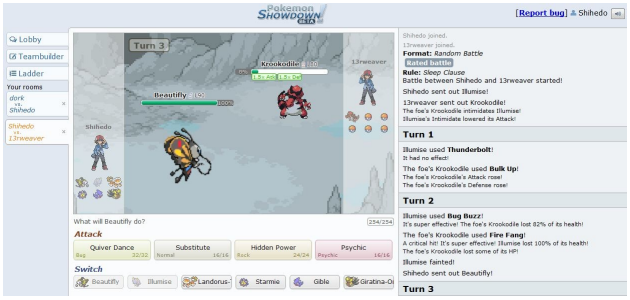

Fig. 1. Sample Pokemon Showdown battle

B. Related Work

There have been other attempts to create AIs to play Pokemon Showdown. Multiple groups in previous years of CS221 have created similar projects with slight variations. For example, one team created an AI that focused on the 'random tier.' That is, a game mode in which each team gets random Pokemon to battle.

In addition to this, one user on the Pokemon forums spent several months creating an AI to consistently play the OU tier (the most popular tier), based on MDPs. We use some of this code, such as the battle simulation that has the Pokemon logic embedded in it, as well as the code to play online. However, this code originally featured a very primitive evaluation function, using only hard-coded values based on intuition. Because the state space for Pokemon battling is so large, depth limited search can only go one or two levels before taking longer than a Pokemon battle allows for a single turn. Thus, an evaluation function is especially important. Here, we present our approach to improving this evaluation function, as well as results from this AI.

II. DATA COLLECTION

Originally, the bot was going to continuously play games to learn the evaluation function. However, we discovered that given that playing on-

line against other humans is too slow and unpredictable, and that it would be difficult to set up the bot to play against itself, we decided to have the bot learn the evaluation function from watching replays of battles from the top players. Specifically, we found the top 500 players in the tier, and downloaded their battles.

A. Match Data

The Pokemon Showdown website has a domain known as replays.pokemonshowdown.com. This site contains replays of any battle that was selected to be recorded (by replay, we mean a complete reenactment of the battle that one can watch). One can search for replays by players, or by tier, and watch the battle to learn from their mistakes, or see how other people play. By having the bot learn from these replays, specifically, replays from the top players, it can figure out what game states are advantageous situations, and which game states are not.

In total, from the 500 players, there were about 22,000 game replays that we could use for learning. More could be obtained if we searched based on the tier, but we wanted the bot to learn only from the best players in the tier. Thus, the bot was trained on these 22,000 games.

III. METHODOLOGY

Match data extraction was achieved through the use of the Selenium WebDriver for Python, much like the other components of this project. First, Selenium was used to find the rankings ladder on Pokemon Showdown and extract the top 500 players from the webpage. After this was accomplished and stored in a text file, Selenium went to replays.pokemonshowdown.com, searched for every individual player, and found all of their battles in the tier. Once the program found all of the battles, it downloaded the html page source for each of the battles. As mentioned earlier, this resulted in about half a gigabyte of just html source code, from 22,000 different pages (each one representing one battle).

A. Match Data Extraction

Of course, after obtaining the html source code, the next step was to figure out what to do with the information that was given. The html for a page contained a complete log of the battle, a snippet of which might look something like this:

```
|turn|6
|callback|decision
|
|move|p1a: Volcarona|Fire Blast|p2a: Meow
|-damage|p2a: Meow|60\/369
|move|p2a: Meow|Stone Edge|p1a: Volcarona
|-supereffective|p1a: Volcarona
|-damage|p1a: Volcarona|0 fnt
|faint|p1a: Volcarona
|
|-heal|p2a: Meow|83\/369|[from] item: Leftovers
|callback|decision
|
|switch|p1a: Kingdra|Kingdra, F|297\/297
|-damage|p1a: Kingdra|260\/297|[from] Stealth Rock
```

There were certain keywords that indicated a change in the battle state. Some of these include 'switch,' for when a player switched out a Pokemon, 'damage,' for when a Pokemon took damage, and 'faint,' for when a Pokemon's health was reduced to zero. The key to extracting the data was to search the log for these words, and modify our game state variable: a variable that kept track of what was happening in the game at a given turn.

For specifics, we first constructed a Pokemon class that contained information about any given Pokemon, such as its current health, its base stats, its typing, and if it had a status ailment. From here, we constructed a Team class that held all the Pokemon on one team, as well as the functionality to keep track of the primary Pokemon on the team. Finally, we constructed a GameState class. This GameState class contained two teams, as well as auxiliary information about the state of the game. Some examples include Stealth Rocks (a move that damages a Pokemon every it switches in), barriers (to increase the defense of an entire team), and the weather during the battle.

To turn the html documents into usable information, we first constructed the initial GameState just from the teams on each side. Then, as we parsed the html document, we updated the corresponding information in the GameState. For example, in the snippet above, when we found a '-damage' flag, we would determine that it was Kingdra that was damaged, and would thus modify the GameState variable so that the Kingdra on the first team had its health changed to $260/297 = 87\%$.

Another important aspect was keeping track of the GameState throughout the entire match, so that the bot could learn from the progression. Thus,

we also had a JSON file that represented how the GameState variable changed throughout the game. When a turn ended, we would dump the contents into the JSON, for later analysis. This resulted in large JSON files for 22,000 games, corresponding to around 10GB of data.

B. Feature Selection

The specific features extracted from each game state would play a defining role in how we are able to model properties of the game. This would impact the weights gathered by our reinforcement learning algorithm, which in turn shapes the evaluation function that is the guiding force behind having a stronger estimation of which game policies are more advantageous than others. Thus, it was of great importance to us to select a vector of features that would represent all key characteristics of the game as well as relationships between these different characteristics (a basic linear model doesn't necessarily capture many relationships, but later on our addition of TD-lambda will enhance this). However, we also did not want to make the feature vector excessive since that could induce the drawbacks of the Curse of Dimensionality, in which the features are sparse and our space of possibilities is far more massive, which leads to difficulties in feature measures being numerically meaningful or discriminative.

Besides including a constant feature (1) to allow for a constant offset in our result, we divided our features into seven categories: *game condition, teams, stats, statuses, stat boost stage, hazards, and type advantages*. Also, note that many of these features come in pairs because one is required to describe the human player agent and the other for the opponent, so both are simply seeking to measure the same property in two different teams. To provide an idea of what some features in these categories represented, here are some examples:

- *Game condition* feature described the macroscopic situation of the current game. It included features such as the number of Pokemon alive on each team, and the turn number.
- *Stats* features involved properties of each Pokemon's stat breakdown. Since each Pokemon is endowed with base levels of each stat, we can take the active Pokemon for the user and create features for its attack, defense, special attack, etc. and conduct the same for the opponent's active Pokemon. In addition to having raw base stat numbers, it's also very important to be able to measure the stat advantages, since these are calculated based on one Pokemon's physical or special attack subtracting the opponents corresponding physical or special defense. So we also added these differences as features.
- The remaining categories followed very similarly, with features honing in on one measurable aspect of the game from both the bot's and the opponent's points of view, then repeating the process and adding more combined features describing how each value relates to others.

Our resulting total had 55 features. We tried to avoid features that would be too particular and lead to sparsity (such as indicator variables for whether each Pokemon is present, which would introduce hundreds of possible new feature dimensions). Thus, features chosen are usually numerically measurable for every single game state and thus have meaningful values in most cases and aren't just equal to 0. For purposes of organization, we employed a naming convention to these features. Feature names would be all lowercase, underscore separated words, where the first word was always the agent the feature describes (player or opp, or something different if describing the game state in general) and the second word was always the feature category, and the remaining words would uniquely describe the rest. For example, if the feature measures the base physical attack stat of the opponent's active Pokemon, we name it:

```
opp_stat_active_patk
```

Weights for these features should be somewhat indicative of how helpful or detrimental they are to winning the game.

C. Epsilon-Greedy

As a first implementation of the AI, we had the bot play in a greedy/random hybrid fashion. The ϵ-greedy approach takes a random action with probability ϵ and otherwise chooses the greedy move, which we define as the active Pokemon's single

attack move with highest damage. This allows the bot to combine exploitation and exploration by choosing strong moves but still retaining an element ϵ of randomness to avoid getting locked into a bad greedy choice. However, this strategy does not account for the opponent's moves at all, so in a two player adversarial game it is not very competitively viable and serves as our baseline.

D. Minimax

Our next implementation strategy used a minimax algorithm. Minimax guards against the opponent's best option. That is, a minimax playing style will pick moves that maximize the player's reward, assuming that the opponent is playing exactly to minimize it. In order for minimax to have full effectiveness, the opponent must be playing optimally. Of course, this is rarely the case in most real-life situations, especially when facing lower-level opponents who may not know the optimal move. As we will discuss later, this strategy can be refined to work well in some cases, but in general it is too simplistic and idealistic to be the strongest overall strategy.

E. Expectimax

After minimax, we created our next bot to play with an expectimax strategy. In expectimax, the agent tries to maximize the expected value of the game. This usually pertains to situations where the opponent is either playing randomly, or even just optimally with an added element of chance. Given that a single Pokemon battle might take 30 or more turns, expectimax is effective at planning for where the game is likely to be headed. It would be very hard for a player to be able to think that far ahead and play optimally from the beginning of the game, so expectimax is useful since it accounts for the optimal strategy as well as the less optimal ones, combining aspects of strategy with those of chance, while altogether weighting them by likelihoods to plan which move would be best for the expected future.

F. TD-Lambda

Finally, we upgraded the expectimax bot by adding an improved evaluation function trained from our 20,000 replays using TD-lambda learning. This algorithm makes use of a lambda (λ) parameter that influences the decay of traces and their impact on the credit of states/actions for a reward. The update rule for this gradient descent method in standard form is

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(v_t - V_t(s_t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_{\vec{\theta}_t} V_t(s_t)$$

where $\vec{\theta}_i$ are the weights used the the evaluation function at the ith step, v_t is the target output, $V_t(s_t)$ is the output of the evaluation function using the model and state of time t, λ is the parameter controlling trace decay, and α is the step-size parameter [1].

We use this generalized algorithm in the specific case with a neural network as part of the evaluation function. Defining the variables below

- $\phi(s^{(t)})$ as the feature vector of inputs to the neural network coming from the state s at time t
- $\theta^{(t)}$ as output layer weights in the neural network at time t
- $w^{(t)}$ as the hidden layer weights in the neural network at time t where $w_{i,j}$ is the weight value of the jth feature for the ith hidden layer node
- $y^{(t)}$ as the output value for the neural network output node at time t
- $h_i^{(t)}$ as the output value for the ith hidden layer node in the neural network at time t

and using σ as the sigmoid function, we can represent the neural network as

$$f(s) = \sigma(\theta \cdot \sigma(w \cdot \phi(s)))$$

Calculating the gradient of this neural network with respect to the weights of the hidden layer and output node (see Appendix), we obtain the following update rules

$$\theta_i^{(t+1)} = \theta_i^{(t)} + \alpha(y^{(t+1)} - y^{(t)}) \sum_{k=1}^{t} [\lambda^{t-k} \mu(i,k)]$$

$$\mu(i,k) = h_i^{(k)} y^{(k)} (1 - y^{(k)})$$

$$w_{i,j}^{(t+1)} = w_{i,j}^{(t)} + \alpha(y^{(t+1)} - y^{(t)}) \sum_{k=1}^{t} [\lambda^{t-k} \tau(i,j,k)]$$

$$\tau(i,j,k) = \theta_i^{(k)} h_i^{(k)} y^{(k)} \phi(s^{(k)})_j (1 - y^{(k)})(1 - h_i^{(t)})$$

which we use to generate the weights for the neural network and generate the evaluation function. The specific choice of certain parameters is discussed below.

G. Optimizations and Hyperparameters

At each turn in Pokemon Showdown, each player has at most 9 actions at their disposal (4 Pokemon moves and up to 5 Pokemon to switch to), giving us a branching factor of 9. These games can involve many turns and lead to a large space of possibilities, and playing online against real players certainly has a time constraint, so we wanted to limit our search space to improve speed. To do so, we used *depth-limited search* with *alpha beta pruning*. We experimented with different depths and found that $d = 4$ was adequate for looking quite a bit into the game's future while also taking only a second or two to estimate the optimal move. We also tried using *Beam Search* and *Monte Carlo tree search* to speed up searches even more, but later on with the implementation of TD-learning to produce a very descriptive evaluation function, the estimates from a limited depth search became much more accurate and the speed already became very reasonable.

Two other hyperparameters that were important in the design of our TD-lambda algorithm were α and λ. Since training weights for reinforcement learning isn't a direct prediction problem, we chose to use qualitative judgment and some experimental tuning to choose hyperparameters. The first, α, represents our step size, which dictates how large our updates are upon reading in each new data point. We chose a small step size of 0.001 because we are reading in hundreds of thousands of states, which will be more than enough to reach convergence, so a lower step size helps ensure greater accuracy. Next, we needed to choose λ, which is our trace decay parameter that represents the proportion of credit given to distant states. Since each individual move can deal damage or change Pokemon to hugely affect the state of the game in the future, we wanted a high value of λ to represent this, so we took $\lambda = 0.9$.

IV. RESULTS

A. Finished Product

Our finished AI bot takes full control of the browser and can engage in Pokemon battles online against real players, all while reading moves from the webpage and updating game states, then running the corresponding game playing algorithm to determine and execute the next move. It also outputs parts of its "thought process" into the console, which looks like the following:

```
My primary: Scizor(344)
Their primary: Stoutland(202)
Their moves:
['Protect', 'Last Resort', u'Crunch', u'Return']
Their item:  Life Orb
Their ability:  Sand Rush
My move:
I think you are going to use
Switch[Hippowdon(420)](2, False, None) and I will use
Bullet Punch(1, True, None).
INFO:showdown.browser:Making a move...
INFO:showdown.browser:Waiting for move...
INFO:showdown.browser:Backup switching Pokemon...
INFO:showdown:Updating with latest information...
```

Fig. 2. Sample console output from the bot in action

B. Performance results

To evaluate the strength of each algorithm, we created separate accounts for the respective bots and ran them to play in actual online Pokemon Showdown games.They would be pitted against real players starting at base ELO of 1000. Winning a game would cause the account to move up to a higher rating and face more challenging opponents, and losing a game would cause the account to move to a lower rating and face less challenging opponents. We ran each version of the bot for 50 games to get a fairly stable reading of its ELO rating (and since each game takes around 10-15 minutes, this would require quite a few hours of monitoring). To standardize the results, we gave every bot the same fixed team of six Pokemon so that they would begin each game in the same state with the same arsenal. The numerical results of ELO and win rate are shown in Figure 3 below.

From these results, we see that Expectimax with TD-lambda achieved the best outcome with an ELO rating of 1344. This fits with our hypothesis

Algorithm	Win/Loss	ELO
Baseline (ϵ-greedy)	10W 40L	1150
Minimax	23W 27L	1247
Expectimax	28W 22L	1301
Expectimax w/ TD-lambda	31W 19L	1344
Oracle	36W 14L	1466

Fig. 3. Our bot's performance using different algorithms

because we intended for Expectimax to account for elements of randomness that improve upon the overly-idealistic minimax, and we intended for TD-lambda to add capabilities to use prior learned information to evaluate the reward value of a particular state as a complement to tree search in Expectimax. With the addition of each new algorithm we do see incremental improvements, and Expectimax with TD-lambda performs the best. If we plot the specific ELO levels of the best performing bot marked every 10 games against our ϵ-greedy baseline, we get the following:

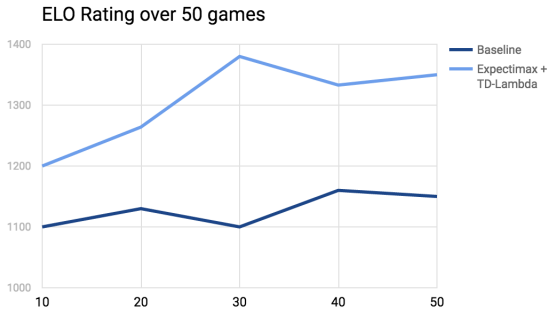

Fig. 4. ELO rating performance of Expectimax with TD-lambda bot over 50 games

We can compare this to the ELO plot of the performance of the minimax bot:

Both bots hit a sort of peak or upper bound at which point they cannot consistently defeat players of that ELO level, which leads to convergence at that certain level of performance. Clearly the TD-lambda bot reached a much better rating, but fundamentally they also behaved very differently. The Minimax bot struggled more to defeat lower-level players, whereas the Expectimax TD-lambda bot quickly rose in ranking and reached its peak much earlier on. This is due to Minimax having

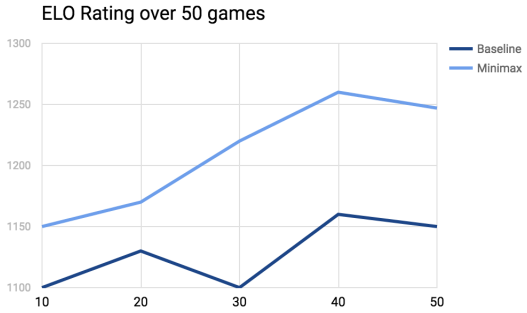

Fig. 5. ELO rating performance of Minimax bot over 50 games

a fixed assumption that the opponent will play optimally, which is not necessarily applicable to lower level players who make sub-optimal or unpredictable moves; hence it wasn't as immediate in defeating them. Meanwhile, the Expectimax with TD-lambda bot took expected values which better accounts for different random possibilities, and had a more general evaluation function that can gauge any general state of the game without assuming either agent's action patterns.

C. Comparisons to baseline and oracle

All three AI methods employed were able to perform noticeably better than the ϵ-greedy baseline. The baseline stood at 1150 ELO and Minimax, Expectimax, and Expectimax with TD-lambda produced ELO results of 1247, 1301, and 1344, respectively. However, our oracle was experienced human player Kush Khosla of our project group, who also played 50 games in the same period and achieved a rating of 1466. This is still far from the very top players (whose games we scraped to train our TD-lambda weights) who have ELOs as high as 1841. Especially considering that rating points become much more competitive and difficult to obtain at higher levels, it's evident that this learning method is still nowhere near sufficient to reach the level of an experienced human player.

V. DISCUSSION

After training, we noticed that there were a few weights that the bot considered to be especially important. One of these weights was for the move known as Stealth Rock. As mentioned earlier, Stealth Rock is a move that damages the

opponent's Pokemon every time they switch into the battle. The bot noticed that in almost every single game, this is one of the first things that top players do. As such, the bot decided that one of its top priorities is to send out Stealth Rocks.

After Stealth Rocks, the bot determined the other most important thing to do was keep its health high. Therefore, it was not uncommon to see the bot send out Stealth Rocks, and then use moves just to keep its health high. This is very much an effective strategy, often called a 'stall' strategy. Sometimes the opponent would not be able to defeat the bot, and would have all of their Pokemon faint, and sometimes this would cause the opponent to quit before the game technically ended. Because quitting is considered a loss, this was not an insignificant source of wins for the bot.

However, there were some decisions the bot made that, although in theory were a good idea, did not work out due to the rules of Pokemon. Again, one such example is Stealth Rocks. Even if the opponent already had Stealth Rocks on their side of the field, the bot would try to continuously send out more of them, despite the fact that Stealth Rocks can only be placed once once. As another example, the bot really valued boosting stats of its Pokemon by using moves such as 'Dragon Dance,' to simultaneously boost the Pokemon's speed and attack attributes. However, in the game, there is a limit to how many times a Pokemon's stats can be boosted. Even when the bot's Pokemon hit this limit, it would still try to boost the stat which would have no effect, resulting in a wasted turn.

Overall, the bot picked up many of the important aspects of play from top players but still made some noticeable mistakes; one such mistake came from to its inability to understand the nuances of repeated moves.

VI. FUTURE WORK

There are multiple ways in which we could continue this project. First of all, we could have the neural network update live as it plays more games against people of different ranks. This way, it can be constantly learning from both its own mistakes and those of other players.

Second of all, there is a possibility that this framework can be extended in order to not only play the game, but construct the most strategic Pokemon team that should be used. It could do this by, for a given possible team, running the evaluation function on this given team and a variety of opponent teams. Then, a metric could be created that would give either a probability or a score of how well the given possible team would do against some common teams and battle styles.

Thirdly, the evaluation function could be extended by equipping the bot to play other types of Pokemon battles - like doubles and triples (where players have two or three Pokemon out at once, respectively). This would likely just require more features in the feature extraction function, and use the same learning algorithm to learn how to evaluate game states.

Finally, we could implement an adaptable AI that changes strategies throughout the game, or even based on the opponent's ranking. Towards the end of a game, because a player has more limited options and is more likely to act optimally, minimax would be more appropriate. However, at the beginning of a game, the bot might benefit from using expectimax more so than minimax, considering there are a lot of different ways in which the game can play out. Expectimax would allow the bot to cover lots of those bases. Similarly, given the ELO ranking of the player, the bot might switch to an minimax strategy sooner rather than later, because players with higher rankings will more than likely play optimally earlier than a player with a lower ranking.

APPENDIX

The TD-Lambda update rule is derived by taking the gradient of our neural network representation

$$f(s) = \sigma(\theta \cdot \sigma(w \cdot (\phi(s))$$

We take the partial derivative of this function with respect to the hidden layer weights and output

layer weights–

$$\begin{aligned}
\frac{\delta f(s)}{\delta w_{i,j}} &= \frac{\delta \sigma(\theta \cdot \sigma(w \cdot (\phi(s))}{\delta w_{i,j}} \\
&= y(1-y)\frac{\delta(\theta \cdot \sigma(w \cdot (\phi(s))))}{\delta w_{i,j}} \\
&= y(1-y)\theta_i h_i(1-h_i)\frac{\delta(w \cdot (\phi(s)))}{\delta w_{i,j}} \\
&= y(1-y)\theta_i h_i(1-h_i)\phi(s)_j \\
\frac{\delta f(s)}{\delta \theta_i} &= \frac{\delta \sigma(\theta \cdot \sigma(w \cdot (\phi(s))}{\delta \theta_i} \\
&= y(1-y)\frac{\delta(\theta \cdot \sigma(w \cdot (\phi(s))))}{\delta \theta_i} \\
&= y(1-y)h_i
\end{aligned}$$

ACKNOWLEDGEMENTS

We would like to thank Professor Percy Liang for teaching us the foundational material that allowed us to work on this project as well as our TA, Michael Chen, for giving us feedback and tips for improving our model/project.

REFERENCES

[1] R.S. Sutton and A.G. Barto, *Reinforcement learning: An introduction*, Vol. 1, no. 1, Cambridge: MIT press, 1998.

[2] G. Tesauro, "Temporal difference learning and TD-Gammon." *Communications of the ACM*, 38(3), 1995, pp. 58-68.

[3] vasumv (2015). pokemon_ai [software]. Retrieved from, `https://github.com/vasumv/pokemon_ai`.