Gotta Train 'Em All: Learning to Play Pokémon Showdown with Reinforcement Learning

Kevin Chen
kvchen@stanford.edu

Elbert Lin
el168@stanford.edu

Abstract

Pokémon Showdown is a faithful open-source recreation of the battle system in the original Pokémon franchise. Gameplay consists of arena-style battles in which players carry out moves simultaneously with the ultimate goal of defeating the opponent's team of Pokémon. We use deep reinforcement learning in conjunction with feature-space embeddings and test the performance of this model against various baseline models.

1. Introduction

Reinforcement learning has long been an area of interest for many areas of research that require high-level control without the use of explicit control or featurization. Recent advancements in the use of deep neural networks have accelerated both the learning speed and resulting quality of these agents. These unsupervised, end-to-end learning methods have proved promising, as evidenced by DeepMind's work in achieving superhuman performance in both Atari games [11] and Go [14]. Reinforcement learning has also found uses in real-world applications such as self-driving cars and robotics.

Pokémon battles are unique in that players are adversarial and move simultaneously; this is in contrast to turn-based games such as Chess. As an extremely complex game with many varied strategies, levels of play, and metagame theories, Pokémon provides an excellent testbed for gauging the performance of modern reinforcement learning algorithms.

We select proximal policy optimization (PPO) as our algorithm of choice. We train a deep neural net to represent the policy evaluation function. The input to our algorithm consists of battle state features; the output of our neural net are probabilities for each move index representing how good that move is in the overall policy. This move index is then fed back into the battle environment and used to progress the game.

2. Related Work

Reinforcement learning methods existed long before deep learning became popular, but previous game-playing agents typically made extensive use of manual featurization. With the rise of deep learning, these techniques were revisited in the context of utilizing machines to train complex nonlinear models.

One seminal paper in this field explores playing Atari games using deep-Q learning [11]. The authors utilize the same set of hyperparameters across all games, showing that the algorithm is able to generalize well across many different environments and conditions. Agents trained as such are quickly able to surpass humanlike performance in many games.

However, Q-learning still has difficulty learning complex value functions; if the target value function is extremely complex and not easily learned, the resulting policy will suffer as well. More recent approaches typically use policy gradient methods, which operate directly in the policy space and can still learn good policies without learning a value function directly.

Recent advancements in policy-gradient algorithms have solved many problems with vanilla policy gradient methods. Trust Region Policy Optimization (TRPO) is one such method that provides theoretical guarantees for monotonic improvements in the policy over time [12].

Asynchronous advantage actor-critic (A3C) [10] is another such approach in which multiple agents are launched in separate environments. This was developed with the intention of decorrelating experiences and stabilizing training. Although we do not explore distributed training in this paper due to financial constraints, we note that our experiments were largely bottlenecked by computational resources. Being able to leverage multiple machines may have enhanced our results.

A relatively new algorithm in the reinforcement learning space is that of proximal policiy optimization (PPO) [13]. Whereas previous policy gradient methods are slow and costly to train, PPO uses a simpler objective function while still ensuring useful results. The new, modified ob-

jective implements TRPO in such a way that it is compatible with stochastic gradient descent, thus achieving high performance with good sample efficiency.

Modern advancements in deep learning techniques have not yet found significant use in Pokémon as of yet. Many implementations instead focus on traditional techniques like minimax to explore a Pokémon battle game tree. Furthermore, the infrastructure for running large-scale simulations in a research environment has not been exposed in a clean, easy-to-use library; the closest we were able to find was a fairly limited environment for AI competitions [7].

3. Dataset and Features

3.1. Embeddings

Data for the embeddings was taken from a dataset on Kaggle [3]. It contains the stats of the first 721 Pokémon of the Pokédex, i.e. the ones corresponding to the first six generations of the RPG. Each row contains the name, type(s), numerical stat data (such as HP, Attack, Speed, etc.), and some other data such as color, height, and whether the Pokémon is considered legendary (in game).

To create embeddings for each Pokémon, we turned the data into a graph to be used with Node2Vec [6], which creates embeddings from graph data in a fashion similar to Word2Vec [9]. It first takes in a graph, then randomly samples that graph to create random walks of some number of nodes. Using these random walks, it creates a skip-gram model that can then be trained to generate embeddings.

For the Pokémon graph, we take each Pokémon and create nodes from its name, type(s), and each of its numerical attributes (Total, HP, Attack, Defense, Sp. Atk., Sp. Def., and Speed). There are also two special nodes, Legendary and Mega. Then, we add edges between the Pokémon's name and all values associated with it: its type(s), its numerical data, and Legendary and/or Mega only if the Pokémon is legendary or has a Mega evolution, respectively. Finally, we apply Node2Vec to this graph.

Pokémon	Most similar Pokémon
Bulbasaur	Chikorita, Turtwig, Nuzleaf, Petilil, Exeggcute, Skiploom, Jumpluff, Oddish, Budew
Caterpie	Wurmple, Weedle, Kakuna, Metapod, Paras, Ledyba, Spinarak, Venonat, Silcoon
Mewtwo	Lugia, Mesprit, Mew, Victini, Celebi, Cresselia, Volcanion, Ho-Oh, Uxie

Table 1. Similar Pokémon within the embedding space.

With the generated embeddings from Node2Vec, we check the most similar nodes of some Pokémon to observe the validity of the results. For example, the most similar Pokémon to Bulbasaur are Chikorita, Jumpluff, Exeggcute, Skiploom, and Gloom, which are all relatively weaker Grass types. To better visualize the embeddings, they were flattened to 2 dimensions and plotted, with each Pokémon colored based on their type, generating Figure 1.

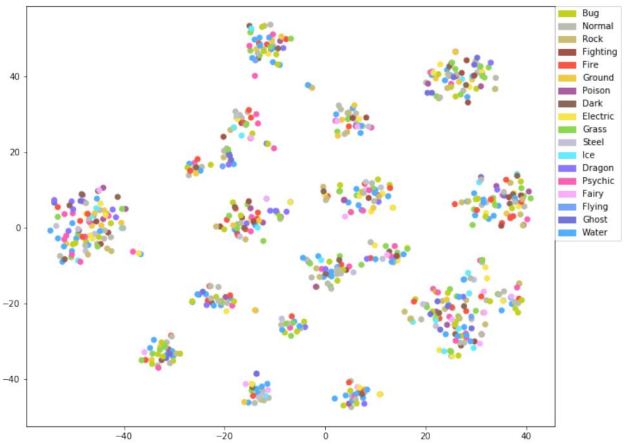

Figure 1. Embedding clusters projected into a 2-dimensional space

While the clusters appear to be random, we are confident that they are mostly correct, based on the nearby nodes as seen above. The graph also does not entirely capture the information contained in the embeddings, since it is a flattened projection. If needed, we could further improve the effectiveness of our embeddings by tuning the hyperparameters used to train the Node2Vec model (e.g. dimensions of embeddings, number of nodes in random walk, number of random walks).

3.2. Offline Simulator

3.2.1 Proxy Server

We built our simulation environment on top of Pokémon Showdown [8]. Pokémon battles are incredibly complex and typically involve a multitude of factors including Pokémon, their attributes, their interactions with each other, any active statuses, and the battle environment. As such, the official server implementation is similarly convoluted.

However, the official implementation is also fairly restricted in that it doesn't support taking "snapshots" of game states, which makes tree-search and rollout-based methods much more difficult. To work around this issue, we implemented our own proxy server built on NodeJS that imports only the battle system from the Pokémon Showdown repository. Our server API is intended to work in tandem with an OpenAI Gym and run on the same machine as the RL client; HTTP requests and responses are used to bridge the gap between our primary language (Python) and the Pokémon Showdown simulator environment (JavaScript). In the future we may switch to a more efficient transport, but we are

largely bottlenecked by the speed at which we can clone game states.

Our proxy server exposes the following functionality over an HTTP interface:

- Creating a new game. This returns a UUID for uniquely identifying the initial state of the game.
- Making a move. Takes in a UUID of a game state and both the player 1 and player 2 moves. Performs an immutable update and returns a UUID that uniquely identifies the subsequent battle state after taking that move. Also returns features from the new battle.
- Cleaning up. Takes in the UUID of the initial game state and releases all subsequent battle states from memory.

For efficiency, we memoize game trees such that nothing is recomputed if taking an action already taken from some game state.

3.2.2 Gym Environment

We expose the proxy server as a clientside library through the OpenAI gym API [5]. This exposes the entire battle system as a clean abstraction through a few methods: `step`, `reset`, `close`, and `seed`. This environment is reusable for any game-playing agent, and has additional lower-level bindings for accessing the proxy server directly (used in the minimax agent).

3.2.3 Feature Engineering

Features are derived from the battle state in the simulator. At a high level, battles consist of two sides. Each side consists of a team of Pokémon, and each Pokémon has some set of moves. Each of these objects (battle, side, Pokémon, and move) has attributes that can be used to derive a feature vector.

3.3. Opponent Agents

For online training of our agent, we created three distinct opponent agents, all playing with a fixed strategy.

- Random agent. Selects any available move at random.
- Default agent. Always tries to pick a non-switching move.
- Minimax agent. Uses a heuristic that prioritizes damaging the opponent (which is a strategy strong at lower levels). Computes the difference between team healths; larger margins represent a stronger game state heuristic.

4. Methods

4.1. Proximal Policy Optimization

The proximal policy optimization algorithm uses the following objective function [13]:

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t],$$

where θ is the policy parameters, $\hat{E}_t$ is the empirical expectation over time, r_t is the ratio of probability of actions under the new and old policies, $\hat{A}_t$ is the estimated advantage at time t, and ϵ is a hyperparameter. Essentially, at each timestep the algorithm chooses a new policy based on experimental data that maximizes the agent's advanatge while maintaining a small deviation from the previous policy. This limit on deviation is encoded by ϵ and the clipping function. For our experiments, we modify a reference implementation from OpenAI [2].

4.2. Network Architecture

We experimented with multiple network architectures, but we settled on a simple multi-layer perceptron for our final experiments. This consisted of a three-layer network with fully-connected layers, each with 512 units and a ReLU activation.

4.3. Move Invalidation

Because only a subset of moves are valid for each player at each battle state, we need to ensure that our agent doesn't select an invalid move (otherwise the battle simulator will error out). To accomplish this, we slightly modify the RL algorithm loop; instead of just observing a state, the agent observes a pair of $(state, actions)$, where $actions$ is a tuple of valid move indices.

We pass in $state$ as the input to our neural net as usual. However, before taking the softmax of our output layer logits, we use the $actions$ as an additional input to force the logits of the invalid moves to negative infinity. This causes the final probability of invalid moves to become 0 after taking softmax.

5. Experiments

We train our RL agent against each of the three opposing agents. Due to computational constraints, we restrict the minimax agent to 1-ply. This turns it into a greedy agent that seeks to maximize damage to the opponent during its turn. Our training was performed on a single machine with an NVIDIA GTX1070. Each agent was trained until its reward curve stabilized (usually around 100 epochs).

Hyperparameter	Value
Steps per epoch	4000
γ	0.99
Clip ratio	0.2
π learning rate	3e-4
Value function learning rate	1e-3
λ	0.97
Target KL divergence	0.01

Table 2. Hyperparameters used for training.

6. Results and Discussion

6.1. Evaluation Metric

We evaluate our model based on the average reward per epoch, which represents the performance of the agent against its opponent. Our agent obtains a reward of $+1$ if it wins the battle, -1 if it loses, and 0 for all other battle states. We also perform qualitative analysis over our agent and analyze the game states at a higher level to obtain insight into our agent's decisions.

6.2. Performance

Opponent	Average epoch reward
Random	0.85
Default	0.58
Minimax	-0.9

Table 3. Average epoch rewards after training convergence for opponent agents.

We found that the agent was able to learn to defeat both the random and default agents, achieving an average reward of around 0.85 after training for about 70 epochs (see Figure 2). On the other hand, our agent failed to perform against the minimax agent (Figure 3), showing almost no improvement even after the same number of epochs. We believe that this is due to the minimax agent being too strong, preventing the RL agent from learning anything useful at a reasonable rate. Likely, no matter which move our agent chose, the minimax agent would choose the best move to counter, leading our agent to believe that there is no move that would clearly be more beneficial in that game state.

Somewhat surprisingly, our minimax-trained agent was able to perform well against the random and default agents despite often losing to minimax. This indicates that the RL agent is indeed learning to attempt moves to defeat the opponent, even if it has not yet learned any particularly strong strategies to defeat more difficult opponents.

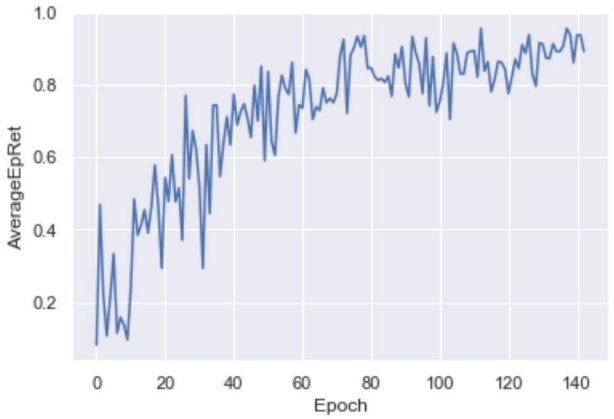

Figure 2. Reward increasing over training epochs against a random agent.

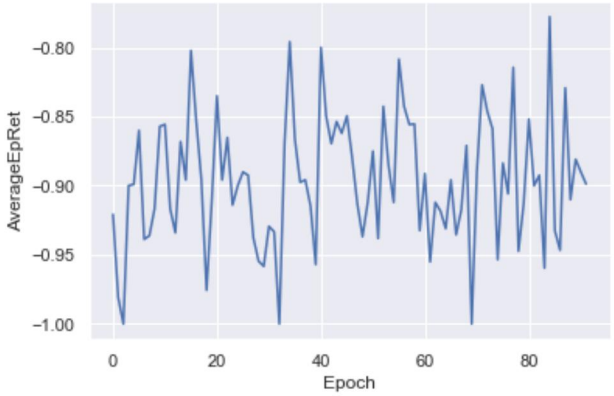

Figure 3. Reward stagnating against a minimax opponent. The agent is unable to make significant progress in this scenario.

6.3. Learned Policy

We had the agent play 100 games against each agent after training, and recorded the moves that it chose, creating histograms such as Figure 4. The first four indices of the move index are using one of the four moves that a Pokémon has, the next five are switching to one of the other Pokémon in reserve, and the last two are passing and no-op (which are the only allowed moves in specific battle situations).

We find that the agent learns to use moves rather than switching Pokémon, but oddly enough it preferentially chooses the fourth move. The agent learned essentially the same policy against each agent, which is why it was able to beat the random and default agents after poorly training against minimax. We are unfortunately unable to determine why the agent learns to only choose the fourth move, as the Pokémon teams and moves are generated randomly, so no clear pattern emerges.

Typically, one would expect the agent to choose moves that would be "super effective" against the opponent (i.e.

the type of the move has an advantage over the type of the opponent's active Pokémon, causing the move to deal twice the amount of damage). It could also choose moves that restore health to its own Pokémon, or choose moves that boost its Pokémon's stats to deal more damage or take less damage in later turns. However, given that Pokémon and moves are random, this is not the case, since only choosing the fourth move no matter the situation indicates that the agent appears to not differentiate between different moves in that slot.

Upon closer inspection of the games played, the agent appears to learn to switch Pokémon when the active Pokémon reaches low health. There are still some peculiarities though, as it almost always chooses to switch to the Pokémon in the last slot (move index 8). The usual strategies players have is to choose to switch when the current active Pokémon is at a disadvantage (e.g. low in health or is at a type disadvantage), and then switch to another Pokémon that would have an advantage (generally one that has a "super effective" move against the opponent's Pokémon). Since the agent only chooses to switch to its last Pokémon, however, it does not appear to know of this strategy, instead opting to switch just so the current active Pokémon would have full or high health.

We are still uncertain as to the reason why the RL agent learns to preferentially choose the fourth move. As explained, battles are initialized at random with different Pokémon each time, each with different moves, and the agent is initialized with random weights. There does not appear to be an issue with the implementation of the learning algorithm, as the moves chosen by an untrained agent are generally uniformly distributed among legal moves, and it does appear to learn to win against the random and default agents. It is possible that this policy is a local optimum, but this is doubtful because we see the agent repeatedly learn the same policy against different agents, rather than a more even distribution of policies among preferentially choosing a specific move. More experiments need to be attempted in order to fully understand why the agent believes the fourth move is almost always the best move it can make.

7. Conclusion

Our results show that an RL agent is able to make progress in playing against simple agents; however, we were unable to achieve performance on-par with human play. Our PPO-trained agent is able to make significant progress against the random and default agents; however, it fails to perform well against the minimax agent.

We believe that although our agent was not entirely successful, the environment and training framework we have developed here will be useful for others attempting to perform similar experiments.

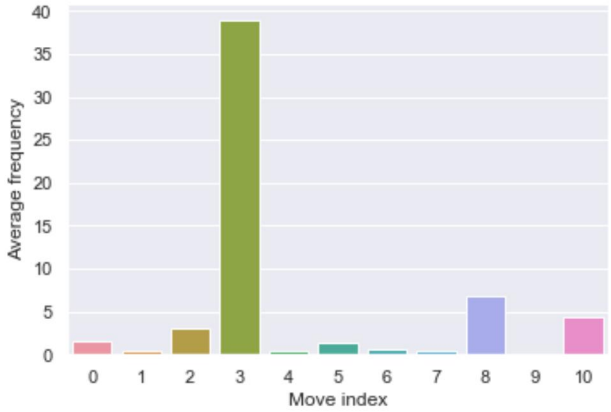

Figure 4. Learned policy against a random opponent.

8. Future Work

Our final agent was ultimately disappointingly flawed, and given more time we would like to find ways to mitigate the agent's strange tendency to choose the fourth move. We would need to develop a method to better visualize what occurs during training, such as displaying a histogram that changes over time as the agent learns new policies. We may also wish to train on certain preset teams of Pokémon, and observe if the agent is able to discover more effective strategies given some predetermined starting point.

Self-play is one final area we wished to explore [4]. We believe that training an agent against itself would prevent the "learned helplessness" problem we had against minimax and allow the agent to learn a better policy.

9. Acknowledgements

We leverage Tensorflow [1] and the OpenAI PPO implementation [2] as part of our code. We would also like to thank the CS230 staff for their hard work this quarter.

10. Contributions

- Kevin Chen: Proxy server, environment, and RL agent implementations
- Elbert Lin: Embeddings and featurization

11. Code References

Our code in this project is broken down into four repositories:

- Clientside OpenAI gym environment: `https://github.com/kvchen/gym-showdown`
- Reinforcement learning agent and opponent implementations: `https://github.com/kvchen/showdown-rl-server`
- Proxy server implementation: `https://github.com/kvchen/showdown-rl`
- Data exploration on final results: `https://github.com/kvchen/showdown-rl-notebooks`

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] J. Achiam. Spinning up in deep rl, 2018.

[3] alopez247. Pokémon for data mining and machine learning, 2016.

[4] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[6] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, New York, NY, USA, 2016. ACM.

[7] S. Lee and J. Togelius. Showdown ai competition. In *2017 IEEE Conference on Computational Intelligence and Games, CIG 2017*, pages 191–198, United States, 10 2017. Institute of Electrical and Electronics Engineers Inc.

[8] G. Luo, B. Halberkamp, C. Monsanto, and M. Dias-Martins. Pokémon showdown, 2018.

[9] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[12] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.