Onix-pected Trainer

Shiva Devarajan, Michael Jurado, Samuel Zhang
Georgia Institute of Technology
North Ave NW, Atlanta, GA 30332
sdevarajan3@gatech.edu; mjurado42@gatech.edu; szhang460@gatech.edu

Abstract

Our team applied deep Reinforcement Learning (DRL) techniques to competitive Pokemon battles. Unlike other approaches to this problem domain, our team applied an online Double Deep Q-Network (DDQN) with soft-update to train our agent against multiple benchmark battle bots. The model showed qualitative and quantitative results of learning and improving over the training iterations. During the evaluation phase, we were able to achieve a 72% and 16% win rate against a Most Damage bot and state-of-the-art Minimax bot respectively.

The source code for this project can be found at: https://github.com/CodeSammich/showdown

1. Introduction

A common theme in machine learning is to benchmark algorithms using video games. Recently, many games, like chess [9] and poker [4], have been conquered by reinforcement learning techniques. The question is, how deep can we go? Can reinforcement learning algorithms conquer games as difficult as Pokemon? Competitive Pokemon is an extremely difficult problem that is non-deterministic, partially observable, and has an extremely large state space. A solution to this problem could be an important benchmark in the capabilities of DRL algorithms. In this paper, we seek to be able to conquer competitive Pokemon through use of Deep Reinforcement Learning.

2. Background

Pokemon is a game franchise that was developed by Game Freak and published by Nintendo. It is an immensely popular game and the most recent game sold the 5th most copies all time on the Nintendo Switch [7]. Pokemon utilizes a turn-based combat system where both players enter their moves simultaneously, and the game resolves the turn order based on certain factors in the game.

For this experiment, we are using game mechanics from Pokemon Sword and Pokemon Shield, also known as Generation 8, released in late 2019. Mechanics from this game have been faithfully reproduced in the Pokemon Showdown battle simulator [12]. We selected the singles OU format, the most popular format on Showdown, for our experiments. This format is a 6v6 battle which Pokemon are sent out one at a time. There are various rules mandated in this format, including certain banned Pokemon, moves, and strategies [10]. This battle format is especially difficult because of the large amount of information encapsulated at the current state due to the magnitude of possible teams being in the order of 10^{218} [3].

3. Related Works

Currently, the state of the art model calculates the reward given all possible action pairs and employs Minimax techniques to select the optimal action. Although this technique takes a long time to run (up to 30 seconds per turn), it is able to defeat many lower ranked players.

The most rudimentary reinforcement learning technique applied to this problem is a vanilla Q-learning bot that maps each state to a discrete state [2]. According to the paper, this approach was able to win in a majority of games against humans (58%). However, they did not benchmark their approach against any hard coded agents, which makes it difficult to compare results empirically.

Another approach from students at Stanford uses Proximal Policy Optimization (PPO) to train an agent to play Pokemon battles [5]. They employ a state space embedding that maps every Pokemon to a coordinate on a 2D-grid before feeding the information into a Neural Network. Although their approach was unable to win against Minimax, their choice of scrambling the agent's and opponent's team every game made their approach more difficult to learn than ours.

In order to get the teams used for this experiment that meet the requirements, we took 20 teams submitted by high ranking players [1]. Each team consisted of all 6 Pokemon and all the information associated with it including moves, stats, and items. The teams were picked to be suffi-

Bot Name	Minimax	Most Damage	Random
Minimax	-	78%	100%
Most Damage	22%	-	98%
Random	0%	2%	-

Table 1. Win Rates of the benchmark bots against each other

ciently different from each other and to hopefully represent all common teams. 6 of these teams were chosen to train against for quicker convergence.

Currently, there have been no bots that have been able to beat humans at a top level of play in this or any other format, but there are many deterministic bots which can be used to compare against. During this experiment, we consider 3 benchmark bots provided by the original source code [8].

- Minimax: the state of the art bot is a standard Minimax bot which calculates all possible options of moves made by both players and picks the Minimax result from this game matrix. By default, the benchmark matrix calculates up to two moves deep.
- Most Damage: This bot always picks the maximum damage move in a current situation. It does not consider the opponent or switching.
- Random Bot: This bot picks a random move from all options each time.

As shown in Table 1, Minimax has the best performance by far and Random the worst. Random has significant difficulty winning against Most Damage and Minimax, and can only win against Most Damage occasionally. Surprisingly, Most Damage occasionally wins against Minimax. A possible explanation to Most Damage's high win rate against Minimax is that Minimax assumes that the opponent is capable of long term planning. Therefore, Most Damage will sometimes thwart Minimax by acting unexpectedly.

4. RL Terminology

Episode: An episode is defined as playing a set number of games against itself. In this experiment, 20 games are played in an episode.

Epsilon ϵ: An exploration constant ϵ governs the percentage of times the agent takes a random move during training. We did not tune ϵ and kept it at a constant value of 0.2.

Evaluation: Every 5 episodes the neural network is evaluated against a Most Damage opponent and a Minimax opponent for 10 games each. During evaluation, ϵ is set to 0, as well as dropout and other regularization parameters.

Tau τ: Instead of updating the target to match the local network after a certain number of steps like in traditional DDQN [2], we use a parameter τ to adjust how much to update the target network with the local network [6]. The equation is shown below:

$$\theta'_t = \tau * \theta_t$$

Every 4 steps, the target network is updated by τ to the local greedy epsilon network.

Reward: In order for the network to learn, we defined a final and an intermediate reward function. The final rewards consisted of a 1 or -1 at the end of the game to incentivize long-term planning and game completion.

On the other hand, the intermediate reward is given per turn to help the bot learn short-term move tactics and offset their variance caused by the randomness inherent to Pokemon's game mechanics. Our intermediate reward function is defined as:

$$R_t = \frac{1}{2000}\left(75(n_{user} - n_{opp}) + \sum_p (h_{user} - h_{opp}) + 30\sum_p (c_{user} - c_{opp}) + 15\sum_p (b_{user} - b_{opp}) + 10(s_{user} - s_{opp})\right) \quad (1)$$

1. n: Number of living Pokemon per side
2. h: Hit points of a Pokemon p
3. c: Conditions of a Pokemon p (e.g. status)
4. b: Boosts of a Pokemon p (e.g. attack)
5. s: Side Conditions per side (e.g. Stealth Rock, Aurora Veil)

The magnitudes of these intermediate rewards typically average between 0 and 0.1 per turn. This reward gives a decent indication of the value of the current state and change in this reward per turn gives a good indication of the quality of the move done by the agent.

5. Approach

We contributed to an open-source interface that runs battle bots on Pokemon Showdown by modifying a Deep-Q-Network reinforcement learning agent for the Showdown environment [8]. In order to augment the original open-source code, we started by converting the total knowable

state information of any particular turn into a long one-hot encoded vector, including all observable Pokemon and environmental information.

We chose to use Deep Reinforcement Learning because all deterministic bots, such as Minimax, are only able to defeat low-level players as their move choices are quite obvious and easy to defeat by higher level players [8]. Deep Learning could help make the bot difficult to predict and therefore able to get wins against smart opponents.

We obtained our data using live simulation data with the bot playing against the Most Damage and Minimax agent. As turns were made, the information was stored in a replay buffer which was used for the agent to learn. Each element of the replay buffer for each turn consists of the previous state, current state, action taken, and reward function. All elements of the replay buffer were generated from games between our bot and the three benchmark bots, each of which were using one of the six sample teams.

We chose to use Double Deep Q-learning (DDQN) over traditional Deep Q-Learning for a smoother and more reliable convergence [2]. Typically, vanilla DQN's have an unstable convergence since they are using a single network θ_t to produce Q-values for the current state and the next state as shown below:

$$Y_t^Q = R_{t+1} + \gamma * Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t)$$

DDQN adds a separate target network with weights θ' whose sole purpose it to evaluate the value of the policy:

$$Y_t^Q = R_{t+1} + \gamma * Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta'_t)$$

Periodically, we update θ'_t to be equal to θ_t so the target network can improve its policy value estimation over time.

Similar to the Stanford paper employing RL for a similar problem, our model used a three layer fully connected network [5]. However, our model used Leaky RELU instead of RELU and a smaller network width of 128 instead of 512. The motivation for using a smaller network with Leaky RELU was to allow the network to converge faster and to avoid dead neurons that could result from our sparse input space.

Although a recurrent neural network structure could potentially help the agent learn, our team decided against adding additional complexity. For example, in order for the network to properly learn we would have had to modify our experience replay buffer to sample chronologically ordered experiences. Also, our state representation contains adequate information for the agent to learn a decent policy without the need of memory.

6. Preprocessing and Postprocessing

For a given turn, the state is represented as a vectorized dictionary of information including the Pokemon on the field, state that each Pokemon is in, environmental factors, and side conditions. For the sake of simplicity, we choose to ignore all previous turns and only consider the current turn as the state. The initial approach was to make each piece of information in the dictionary into a one-hot vector encoding. However, even with the Markov assumption, a complete one-hot encoding for the state space is prohibitively large and unmanageable to work with and learn.

Using domain knowledge, we reduced the state space including only the most necessary information for the network to learn. A notable piece of information removed is our Pokemon. Since we keep a fixed team, there was no need to store our Pokemon in the state space. Instead, we replaced it with a simple one-hot vector denoting what current Pokemon is on the field. Given the large Pokedex available in Generation 8, our one-hot encoding of the Pokemon names massively outnumbered our other state information. In order to remedy this, we shrunk the state space by removing the one-hot naming encoding entirely, as a Pokemon's name is not directly relevant to the battle's outcome when common stats, types, and move sets are able to be learned.

This preprocessing allows it to learn with much less information and allows the model to generalize to Pokemon it hasn't seen before as it will be similar in stats and typing to a Pokemon it has seen before.

We also employed post-processing techniques to allow the network to learn faster and generalize better. Originally, we had the network map to only of the 9 changing actions that one could do at each state. But, we changed it to return a 25 dimensional array which maps to the following information:

- 1-6: Switch to the Pokemon in the corresponding position on the team roster
- 7: Status Move
- 8-25: Select move of a specific elemental type (e.g. Fire)

This way, instead of choosing one particular move, our network would choose the type of move it wanted. If that move was not available in the current available move-set, it would continue choosing the next most preferred result until it found a valid choice. However, this required us to make the assumption that a pokemon would not have 2 moves of the same type.

This preprocessing and postprocessing allows the network to converge must faster as the network can learn smaller pieces of information (such as type matchups) rather than every single scenario. This approach theoretically allows it to be more generalizable to different opponent teams.

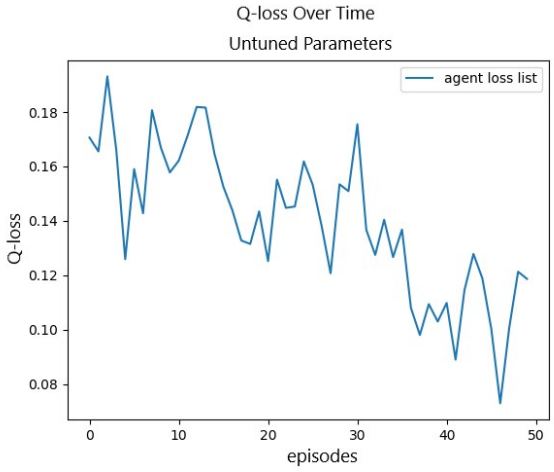

Figure 1. The agent loss decreases over time with the initial set of parameters but the loss curve fluctuates. (LR: 5e-4, γ = 0.99, τ = 0.001, Batch Size = 64)

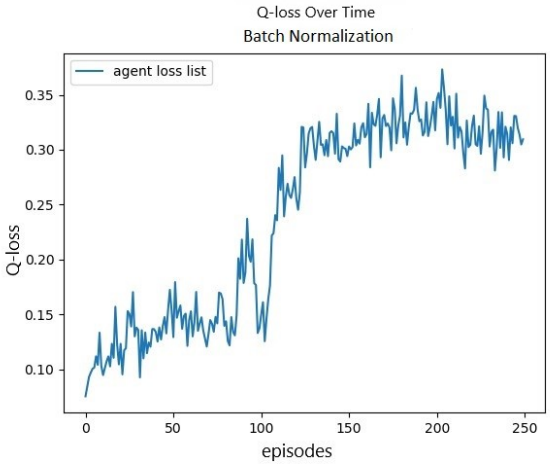

Figure 2. The Q-loss function graph for a batch normalized run increases over time and has a large discontinuity around episode 100. This sharply increasing loss indicates the model is not learning.

7. Hyperparameter Tuning

In order to achieve our results, we had to experiment with various combinations of hyperparameters for the reinforcement learning agent, including:

1. Learning Rate α
2. Discount Rate γ
3. Soft Update Rate τ
4. Batch Normalization (BN)

The state and action spaces are quite large, even after our reductions. Thus, the agent must traverse a significant search space in order to begin winning and obtain a substantial reward. In addition, the Pokemon Showdown environment caused significant delays and often crashed, as it was not originally developed for rapid bot-to-bot matches. As a result, runs that lasted for 200 episodes often took over 10-15 hours to play 4,000 games.

Since we were only able to run a limited number of games, we decided to use the adaptive momentum (Adam) optimizer to adjust for our network's sparse gradient. Furthermore, due to these time constraints, we were only able to iterate through a few dozen hyperparameter configurations. Our initial parameters came from an implementation of DDQN found online [11]. The performance with these default parameters can be seen in Figure 1.

In order to compensate for this limitation, we had to consider the nature of a Pokemon battle and select the hyperparameters most likely to be relevant. For example, we believed γ to be important since it influences how much the agent prioritizes long term versus short term rewards. For example, a gamma set to 0.99 encourages the agent to formulate long term strategies, while a lower gamma values prioritize shorter term strategies. But, if the gamma is too low, then it would not seek a long term victory. Another hyperparameter we decided to tune was the learning rate. A lower learning rate usually makes learning curves smoother but also increase training time.

Since the Q-loss for the original parameters (Figure 1) was fluctuating wildly, we decided to vary both of these parameters. Firstly, we lowered gamma from 0.99 to 0.8 since our reward function is non-sparse and accurately reflects agent performance through time. Our motivation was to make the agent discover a policy faster by making it more greedy. Additionally we lowered the learning rate from 5e-4 to 1e-4 to facilitate a smoother convergence. Figure 4 shows the resultant Q-loss learning curve. Clearly the curve is much smoother and matches the expectation of loss decreasing exponentially as training progresses. Therefore, we decided to adopt these hyperparameters into our final model (See Column 3 of Figure 5).

Finally, we experimented with applying batch normalization. Somewhat amusingly, when batch normalization was applied, the agent loss dramatically increased over time, as shown in Figure 2. Furthermore, Figure 3 shows that the agent was not increasing its accumulation of rewards over time, indicating that it was not learning. In contrast, our final non-normalized model was able to achieve higher accumulated rewards and positive trends against both benchmarks 7. Therefore, we decided not to use batch normalization and cannot recommend its use for this problem domain.

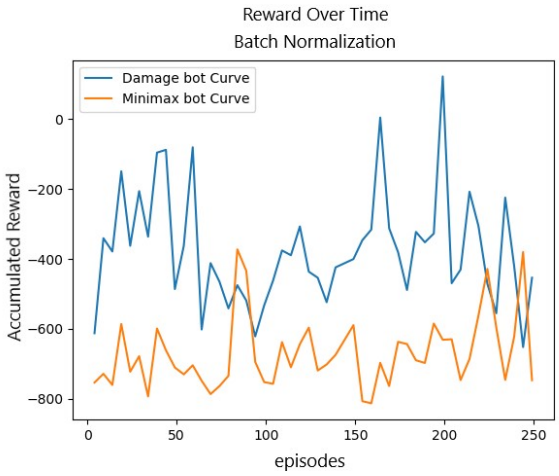

Figure 3. The reward graph for a batch normalized run is centered in a negative region during the entire episode. It is failing to accumulate rewards and win games.

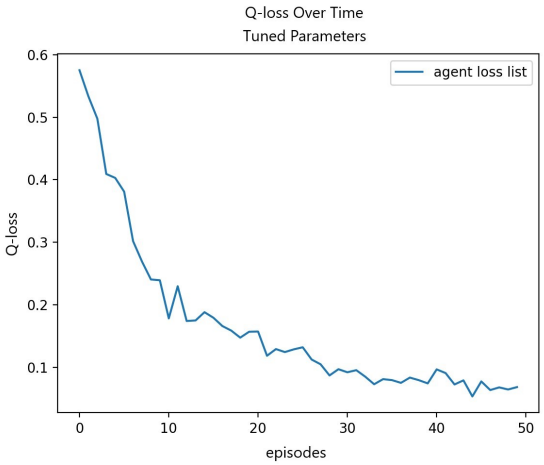

Figure 4. Q-loss is much smoother with tuned parameters. (LR: 1e-4, $\gamma = 0.8$, $\tau = 0.01$, Batch Size = 16)

8. Results

As this is a competitive game, the only metric that truly matters at the end of a game is whether the agent won or lost. We sought to build a bot that could defeat the state-of-the-art Minimax bot, which is able to defeat most low ranked players and is estimated to be better than approximately 70% of players [8]. Since we realized this was unlikely with the time allotted, our preliminary goal was to defeat the Most Damage bot with high probability.

For our final evaluation run, we trained on a randomly initialized model against 400 episodes, which is equivalent to a total of 8000 games alternating between facing Minimax and Most Damage. For each game, we randomly chose the opponent to be one of six teams, but used a constant team for our agent. For this run, we used the final parameters in Figure 5 as they were the most successful in our test runs. As mentioned earlier, we did not exhaustively search the parameter space due to training time constraints.

Parameters	Original Value	Tried Values	Final Value
Buffer Size	10000	10000, 1000	1000
Batch Size	64	8, 16, 32, 64	16
Gamma	0.99	.5, .7, .8, .9, .95, .99	0.8
Tau	0.001	.1, .01, .001	0.01
Learning Rate	5.00E-04	5e-5, 1e-5, 5e-4, 1e-4	1.00E-04
Batch Norm	0	0, 1	0
Dropout	0	0, .1, .3	0

Figure 5. Hyperparameters tuned and final selected values. Batch Norm and Dropout were not used in the final evaluation due to adverse results

In doing so, we observed that a γ of 0.8 allowed the agent to learn significantly faster when compared with higher values. As mentioned earlier, we believe this was likely due to the lower value allowing the agent to prioritize shorter term tactics, which it had learned faster due to our intermediate reward function. Similarly, a lower α of 10^{-4} allowed the agent to learn well against weaker opponents (i.e. Most Damage). However, tuning these parameters did little to improve our performance against stronger opponents like Minimax. This likely means that our agent still has difficulty learning positive behavior when it is consistently and overwhelmingly losing. One possible remedy to this issue is to further tune the intermediate reward function to better define positive behavior.

Unfortunately, we do not have clear explanations for why a batch size of 16 or a τ of 0.01 were particularly successful. One potential explanation could be that a relatively high τ (i.e. 0.1) could cause the target network to continually overestimate our Q-value function. Conversely, a relatively low τ (i.e. 0.001) could cause the target network to underfit and update the value of the policy too slowly.

As shown in Figure 6, we were able to take our win rate versus the Most Damage bot from 20% to above 70% in just 400 episodes or 8000 games. We were also able to learn to beat the Minimax bot nearly 20% of the time, despite it being an extremely hard opponent to learn as we started at a 0% win rate.

Although wins was the true measure of success, since win rate can be inaccurate due to the luck involved in Pokemon, we also incorporated reward function defined above as an additional metric of success. As shown in Figure 7, the reward function showed a clear increase versus the Most Damage bot, and a small, but positive slope versus Minimax.

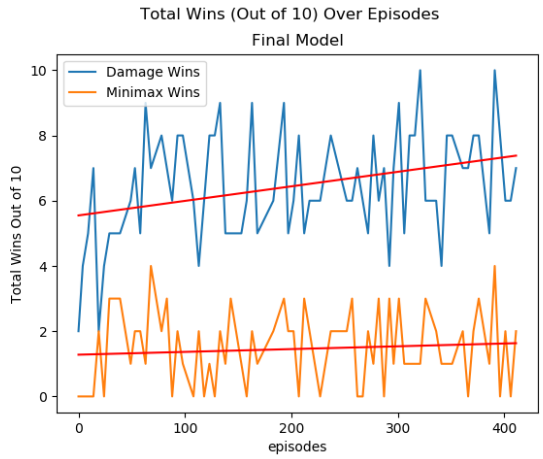

Figure 6. For the final model, the win rate oscillates around 70% as it trains. Furthermore, the rate of total wins over time increases over time as shown by the linear fit.

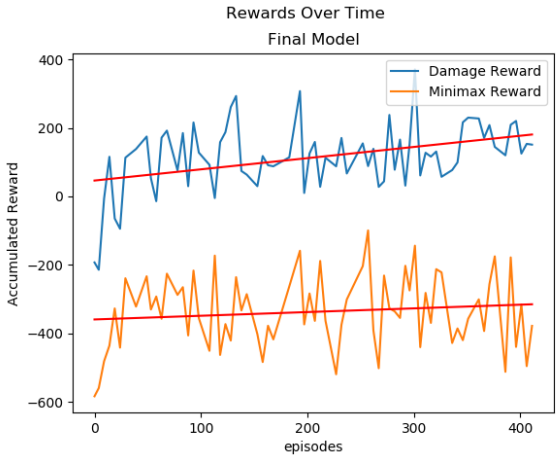

Figure 7. Our final model's positive reward slope against Most Damage and Minimax shows that the agent is learning over time. However, the small Minimax slope indicates that the agent will likely never beat Minimax consistently.

9. Limitations to our Approach

Although the agent obtained a high win rate against Most Damage, our approach was not robust to playing against out of sample Pokemon teams. When we constructed 10 new teams, our agent only won only around 30% of the games playing against the Most Damage bot. In other words, the agent lost twice as often against out of sample teams. In order to remedy this situation, we would need to randomly sample teams for the agent to play against or provide the agent with a much broader set of teams.

Another limitation our agent faces is the limited state and action space we are feeding into it. The agent, for example, is not able to see what move was just taken against it. Moreover, since the agent is not recurrent, it is unable to store information about what moves were recently used against it. In our personal experience as Pokemon trainers, remembering opponents moves is essential to adjusting one's strategy on the fly.

Figure 8. Although Excadrill knew many effective moves, like Earthquake, Iron Head, and Stealth Rocks, it overused Rapid Spin in battle. This was due to our reward function assigning too much weight to boosted abilities.

Our preprocessing strategies also caused many problems that made it difficult or impossible for the DDQN to learn. For example, our agent's state representation does not explicitly discriminate between different Pokemon. We did not include this information as a one-hot vector because it would have added 13632 variables to our state space, which may have ballooned the training time. This information is crucial because different Pokemon, even though they may have the same type and similar stats, have different abilities and move sets. Notably, we noticed one of our Pokemon was repeatedly using a water move on a Seismitoad which has the ability Water Absorb, which makes it immune to all water moves. Although the agent can theoretically learn this information over time, the lack of a concise preprocessing hinders its ability to learn quickly.

Another area that could be improved with our approach was the intermediate reward function. When playing against a human benchmark, our team noticed that the agent's Excadrill, shown in Figure 8, kept spamming the Rapid Spin move even though there were more advantageous moves available. Upon further examination, we discovered that boosting moves were given a disproportionately high reward compared to others. This oversight coupled with a low γ caused the agent to continuously choose Rapid Spin because the move increases Excadrill's speed.

Student Name	Contributed Aspects	Details
Shiva Devarajan	Implementation and Experimentation	Created state encoding and conducted numerous tests
Michael Jurado	Implementation and Research	Allowed parallel runs and conducted DQN research
Samuel Zhang	Implementation and Experimentation	Created action space encoding and neural network

Table 2. Contributions of team members.

10. Conclusions and Future Work

In conclusion, we were able to defeat the state-of-the-art Minimax algorithm and Most Damage approximately 16% and 72% respectively on the 6 sample training teams. Despite not being perfectly generalizable, it does significantly better than random out of sample teams (Table 1).

Although this is was only a decent preliminary result, we still believe that a Deep Reinforcement Learning algorithm has the potential to perform even better against Minimax and even real players a high percentage of the time. We recommend many future experiments that could increase the model's accuracy, speed of convergence and ability to generalize including:

- Introducing a LR Decay parameter
- Scaling up training and running for more episodes
- Regularization Techniques (like Dropout)
- Deeper/Wider Model
- Autoencoder/Dimensionality Reduction
- More robust state space
- More robust action space when dealing with switches
- Recurrent Neural Networks/LSTM
- Replacement of DDQN with policy gradient architectures like PPO and A2C

We also recommend other methods of training:

- Training versus itself
- Training on the live ladder
- Training versus more or randomly generated teams

References

[1] Various Authors. Smogon SS OU Rate My Team. 1

[2] David Silver Hado van Hasselt, Arthur Guez. Deep reinforcement learning with double q-learning. https://arxiv.org/pdf/1509.06461.pdf, 2015. Paper on DDQN. 1, 2, 3

[3] Delibird Heart. How many possible teams are there in pokemon? https://www.smogon.com/forums/threads/how-many-possible-teams-are-there-in-pokemon-a-comprehensive-and-detailed-answer.3648997/. 1

[4] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. 1

[5] Elbert Lin Kevin Chen. Gotta train 'em all. http://cs230.stanford.edu/projects$_{f}$all$_{2}$018/reports/12447633.pdf. 1, 3

[6] Timothy P. Lillicrap et al. Continuous control with deep reinforcement learning. https://arxiv.org/pdf/1509.02971.pdf, 2016. Original paper introducing soft update with DDQN. 2

[7] Nintendo. Top selling title sales units. https://www.nintendo.co.jp/ir/en/finance/software/index.html. 1

[8] pmariglia. Showdown. https://github.com/pmariglia/showdown. Original source code that was the base for this project that had minimax and most damage bots. 2, 3, 5

[9] David Silver et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 1

[10] Smogon. Smogon OU Rules. https://www.smogon.com/dex/ss/formats/ou/. Rules for OU Metagame. 1

[11] Unnat Singh. Deep Q Network with Pytorch. https://medium.com/@unnatsingh/deep-q-network-with-pytorch-d1ca6f40bfda. 4

[12] Zarel. Pokemon battle simulator. https://github.com/smogon/pokemon-showdown/, 2020. Source Code for Battle Simulator. 1