Final Report

Reinforcement Learning for Pokémon Battling

Julia L. Wang, Ryan Chen, Vishnu Akundi

ECE324

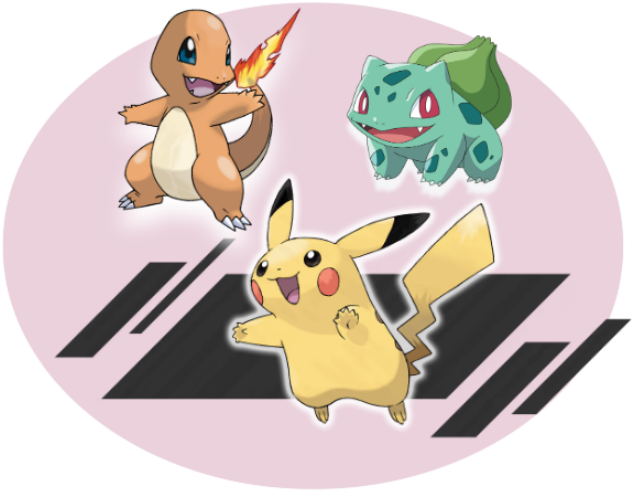

Table of Contents

1 Introduction

This project aims to train agents through reinforcement learning (RL) to maximize performance by providing an optimal strategy during a Pokémon battle on Pokémon Showdown. Pokémon Showdown is an open-source Pokémon battling simulator allowing users to play against each other or AI. The agents are aiming to use their own Pokémon to defeat the opponent by depleting the opposing Pokémon's HP while maximizing one's own (see Appendix 1). These agents will be trained using a variety of hyperparameters and policies in an effort to create or find an optimal policy for the task of Pokémon battles. The goal of this project is to answer the following research question:

> How can the strategies that each agent develops be critically examined? How do/can these decisions emulate rational behaviour?

As described above, the game is turn-based, allowing each agent to make decisions based on the current state of the battle. The game was run on a local server hosting Pokémon Showdown on which the agents were deployed. At each turn, agents have the opportunity to make 1 move from their move sets. These moves can either attack the other agent's Pokémon and gain a potential short-term advantage over their opponent, take the chance to heal, or recharge to improve future attacks. This forces the agent to weigh the long-term and short-term benefits of taking an action, making it a more complex problem that cannot be simply solved with a greedy approach. We seek to output a policy that provides the agent with the optimal series of actions to lead to a victory over the opposing agent. This interim report discusses the progress the team has made. So far, the team has been able to set up an environment that runs on the actual game's servers. The team will be able to control the actions of a pair of agents as they play the game. These actions are currently given by rudimentary learning policies. Eventually, the team plans to use the output of an RL algorithm to predict the most optimal next action for each agent.

2 Prior Work

2.1 Kalose et al. (2018): Optimal Pokémon Battle Strategy

Kalose et al. [3] use Q-Learning in conjunction with a softmax exploration strategy to train an RL agent which maximizes its probability of winning a Pokémon battle against another random agent. As mentioned previously, Kalose et al. use a web-scraper to collect Pokémon stats to train their model. They aimed to find an optimal battle strategy given the current state of the battle. Kalose et al. use a model-free approach which implies that the agent does not need to model the environment; it only needs to respond to input from the environment (opponent's actions and consequence on its own health). They also create their own simulating platform to control the environment and simplify the state space by including only select Pokémon and move sets. The Q-Learning methodology used does not employ a neural network and is driven by a deterministic model that outputs the next action based on the current state of the board and the potential reward of performing a given action. The state vector used in the Q-Learning process encodes important information regarding the status of the agent as well as the opponent such as aggregate health, the Pokémon themselves, and other Pokémon on the team. The agent was given a reward every time a hit was scored on the opponent, while it was penalized for taking damage or losing the game.

It is important to note here that we do not use Q-Learning as our primary method in updating models. In our testing thus far, it is shown that using alternate distribution-based methods such as the Epsilon Greedy Q generally fare better against more skilled opponents such as the max-damage player. This is likely because the distribution-based model is able to learn at a faster rate because it takes more risks, thus being more likely to discover a wider breadth of strategies in the limited time it has to train. A Deep Q-Learning approach also allows for the model to account for an ever-increasing number of states. Kalose et al. had to limit the scope of their project by scaling down the number of allowable Pokémon and select move-sets. Kalose et al. also dropped the use of items (check Appendix for elaboration) as it would introduce too many states. As such, they greatly simplified the environment in which the agent will be training. However, the team has been able to use an existing Pokémon Environment[2] which does not compromise on the intricacies of the game, allowing for a greater number of states. This is complemented by the DQN which allows for decisions to be made over much larger state spaces. Since this DQN will be using state variables from the official Pokémon Showdown website, the results will be better trained when deployed against humans on the same platform.

2.2 Chen & Lin (2018): RL for Pokémon Showdown

Chen & Lin's paper [4] covers the process of setting up an environment and evaluating agents participating in the aforementioned Pokémon battles using a Proximal Policy Optimization (PPO) algorithm to evaluate and iterate/choose a new policy until the agents act as desired. The algorithm takes the current state of the battle on the agent's turn and then outputs the probabilities of each move's effectiveness in achieving the desired result of winning.

Chen & Lin trained their model against 3 different agents: a random agent, an agent which only picks attacking moves (default agent), and a minimax agent which uses a heuristic that prioritizes damaging the opponent. The PPO agent was evaluated on a single discrete metric where it received 1 point reward for winning and -1 for losing. Averaged per epoch, this became the evaluation metric; the faster the model improved, then the higher the average epoch reward as the number of epochs approaches infinity.

Chen & Lin found that their PPO agent was able to defeat the other two opposing agents with more ease than the one it was trained on, which suggested that the agent was learning which moves were strong against which Pokémon, but never learned any specific strategy to implement in battle. This may also be a result of the minimax agent being significantly stronger than the other two opposing agents in training, such that the PPO agent was defeated too quickly and was not able to learn anything from battles.

Contrary to the results of Chen & Lin, our DQN agents performed with relative success when trained against a greedy-choice max agent. This may be the result of methodology and the choice of algorithm or RL architecture (likely the latter). Our DQN model was able to learn and optimize during each battle on a turn-by-turn basis, based on the state of the field, its own Pokémon, and its opponent's Pokémon. This likely gave it an edge in training, as learning on a turn-by-turn basis would allow it to catch mistakes more often and correct them more often, as opposed to a simple win-loss record to adjust its strategy. Despite playing in a more complex environment (the Pokémon Showdown server, as opposed to a basic terminal game), access to more data and more frequent optimization improved performance considerably.

3 Our Work

3.1 Data gathering and server setup

All experiments were conducted on a local server setup hosting Pokémon Showdown, an open-source Pokémon battling library [1] using Node.js. The Python environment poke-env [2] was used for our agents to interact with the Pokémon Showdown interface. This environment automatically retrieves state information of the battle including possible moves with corresponding damage, items, abilities, and type, how many Pokémon have fainted, and data about opponents' Pokémon. The database consisting of Pokémon information was directly taken from Pokémon Showdown [1].

3.2 Training models

All agents were trained with a Deep Q Network (DQN) utilizing the Keras-RL library. The RL agent class implemented consists of a 2-part structure with a state embedder and reward calculator. The embedder takes in a special poke-env 'battle' class, which contains all information necessary to determine the game state (weather, terrain, traps, Pokémon stats, health, abilities and types, their items, the agent's and opponent's active Pokémon), and creates an embedding vector containing the necessary values for computing the reward, using the function below (see Appendix 7.1 for more details regarding multipliers). The reward function is displayed below.

$$\sum_{p \epsilon T}\left[HP_p \cdot w_{HP} + F_p \cdot w_F + S_p \cdot w_S + W \cdot w_W\right] - \sum_{q \epsilon O}\left[HP_q \cdot w_{HP} + F_q \cdot w_F + S_q \cdot w_S + W \cdot w_W\right]$$

p: Player Pokémon

q: Opponent Pokémon

T: Player's team

O: Opponent's team

HP: Percentage Health

F: Fainted Pokemon (Boolean)

S: Status Condition

w: weights

W: win

State considerations include comparing the agents' remaining Pokémon alive and their HP statuses with the opponents. All battles were Generation 8 Pokémon battles with the agent playing single-player mode against the Random Player opponent, as provided by poke-env which only handles up to Pokemon generation 8. The DQN agents were all trained over 1000 steps with associated policies. See Appendix 2 for the default training parameters.

After our presentation, we shifted our focus to developing a simplified reward function. Further discussion with our supervisor led to the proposal that the previously defined reward function's emphasis on health differences between players was creating a bias towards specific strategies which maximized the health percentage of the agent's own Pokémon relative to their opponent's Pokémon. This can be achieved with the reward function presented above by setting all weights to 0 except for the weights pertaining to the number of wins. This should let the agent develop a more organic strategy, with a lesser bias towards percentage health. Because of the turn-based nature of Pokémon battles, a Pokémon's value in battle is only as good as how many turns it can remain viable in damaging opposing Pokémon or impeding the opponent's strategy, a continuous percentage evaluation of the metric is suboptimal, as a stochastic and discrete evaluation is more accurate.

The function for determining this metric is as follows (when not considering switching):

Viable Turns Remaining for Pokémon: $T_{viable} = \lceil \frac{HP}{dmg_{opt}} \rceil$

HP: Own Pokémon's Current Percentage Health
dmg_{opt}: Optimal damage dealt by opponent's Pokémon's per turn

The current reward also does not take into account each individual Pokémon's capabilities at differing amounts of health. Not all Pokémon are equally effective at the same percentage health, because varying stats, matchups, move sets, held items, and abilities can affect the optimal damage capabilities. The concept of only rewarding based on wins and loses was explored further under Prior Works by Chen & Lin (2018).

3.3 Specific Objective

Through our project, we want to create an RL agent capable of emulating rational behaviour. Since this is a very generic research question, we scoped down our problem to the following:

> Train an RL agent that is capable of winning against the Max-Damage Player 80% of the time.

4 Experimental Results

For the experiment, the RL agents were evaluated against Random Player, Max Player, and Minimax Player agents for over 300 battles each. The Random Player chooses a random move from the available moves, and the Max Player chooses immediate moves which deal the most damage. Meanwhile, the Minimax algorithm was implemented with a depth of 1. This means that the agent will be able to look one step ahead. By training against these agents, the team hopes to see certain patterns of behaviour emerging as a result of training against certain types of agents. This will be further explored along with the results of our experiments.

4.1 Experiment 1: Choice of policy

3 agents were trained against the Random Player with different policies and the same hyperparameters. Policies were imported from Keras-RL, where the Linear Annealing Policy was utilized with varying inner policies for Epsilon Greedy Q Policy and Boltzmann Q Policy. The 3 selected policies for the experiment were Epsilon Greedy Q Policy, Greedy Q Policy, and Boltzmann Q Policy. Each policy was trained over 10000 steps and utilizes sequential memory provided by Keras-RL.

The *Greedy Q Policy* configures an immediate Q table consisting of Q values assigned to the possible moves which represent the relative success of each immediate move according to the rewards set by the team. It then proceeds to pick the move which will offer the best reward at a given time. This generally yields models which are short-sighted and limited by the size of the Q-table and how many steps it can see into the future.

The *Epsilon Greedy Q Policy* is similar to the Greedy Q Policy, however, it is given an epsilon (ϵ) value, where $(1 - \epsilon)$ becomes the probability that the algorithm makes the greedy choice. Otherwise, it is free to randomly pick an immediate suboptimal outcome, however, this may benefit the model, as an immediate loss may result in a larger reward (based on the balancing of the game: Pokémon as a turn-based combat system).

The *Boltzmann Q Policy* takes the softmax of the Q values to extract probabilities for each choice, which is then further controlled by a parameter tau (τ). This τ serves as a scalar denominator to each Q value in the softmax function, which controls the spread of the probability distribution yielded by the Boltzmann Q Policy. What sets this policy apart from the Epsilon Greedy Q Policy is that the probability of suboptimal choices is still kept relative to each other. When the Epsilon Greedy Q Policy chooses suboptimally, it chooses with a uniform probability distribution for all suboptimal choices, whilst the Boltzmann Q Policy does not, as it maintains the relative weights of each choice; some suboptimal choices are still more optimal than others.

The results of changing RL agent policies are summarised in Table 1 below.

Agent Policy	Victories against Random Player	Victories Against Max Player
Epsilon Greedy Q Policy	97/100	82/100
Greedy Q Policy	95/100	70/100
Boltzmann Q Policy	98/100	75/100

Table 1: Effects of Policy on Agent Success

It is important to note that the Boltzmann Q Policy was trained with a tau value of 1, meaning it was significantly more likely to take risks than the Epsilon Greedy Q Policy with an epsilon of 0.1. This meant that the Boltzmann Q Policy was spread normally and therefore the optimal choice per turn was not selected as often. Contrary to this, the Epsilon Greedy Q Policy selected the most optimal choice 9/10 times on average. Given these results, it is certainly possible to train the Boltzmann Q Policy to match the Greedy Q Policy in terms of risk-taking and choosing greedily based on hyperparameters tau and epsilon in the following trials. Perhaps a better choice for tau which is less exploratory could result in an improvement in the Boltzmann Q Policy's performance.

4.2 Experiment 2: Policy Hyperparameter Tuning

Experiment 2 aimed to investigate the optimal hyperparameters and policy choice for the agent. As such, the agent was trained against a MiniMaxPlayer using 10k in-game turns per epoch. The results are displayed in Table 2 below, where the wins against the MiniMaxPlayer are shown out of 100 games. All other parameters are held constant to the defaults outlined in Appendix 2.

Hyperparameter		Greedy Q	Epsilon Greedy Q	Boltzmann Q
Epochs	1 (default)	54	54	48
	5	71	59	47
	10	76	70	49
Reward Function	default	54	54	48
	w_F (2->5)	48	55	41
	w_S (0->1.5)	45	48	42
Gamma	0.5 (default)	54	54	48

	0.75	52	50	48
	0.99	36	35	42

Table 2. Effects of hyperparameter tuning and policy choice on agent success

As seen in this table, the win rate consistently increases as the number of epochs is increased. This general trend implies that the model was underfitted for those training cycles with a lower number of epochs. We also note that the win rate does not plateau for Greedy Q and Epsilon Greedy Q, showing that the models may still be underfitted by the end of 10 epochs.

The gamma function during hyperparameter tuning represents a coefficient that determines how heavily to weigh/value future rewards against a potential suboptimal choice in the present. The higher the gamma value, the more future rewards are considered over present ones. The results from this experiment show that the model is not efficient at predicting potential future rewards, thus, impeding its performance.

Overall, this experiment yielded that the EpsGreedyQ policy is most resilient to reward function weight changes.

4.3 Experiment 3: Round Robin

Experiment 3 investigated the optimal agent based on the player it was trained against. This was conducted by having the agents play against each other after training. The results displayed in Table 3 are the win rate over 100 games averaged over 10 trials, each trained using 10k in-game turns with varying epochs. The numbers displayed above the diagonal are the number of times Player 1 won against Player 2. Similarly, the results below the diagonal illustrate the number of losses against Player 2 (the number of wins from Player 2s perspective). Games in which the players won over 50% of the time were highlighted in green for emphasis.

Player 1 → Player 2 ↓	Random	Max	RLAgent1	RLAgent2	RLAgent3	RLAgent4
Trained vs			Random	MaxPlayer	RLAgent1	RLAgent2
Epochs			1	2	5	5
Random			95.3	98.7	50.2	52.3
MaxPlayer			79.4	73.2	23.3	13.5
RLAgent1	4.7	20.6		45.5	56.8	51.4
RLAgent2	1.3	26.8	54.5		51.1	52.7
RLAgent3	49.8	76.7	43.2	48.9		51.9
RLAgent4	47.7	86.5	48.6	47.3	48.1	

Table 3: Effects of training on agent success through round robin

As seen in the table above, RLAgent 4 performs the best out of all the agents tested here. We note that RLAgent 2 (The Agent RLAgent4 is trained against) performs better against MaxPlayer. This is

because RLAgent2 is trained directly against MaxPlayer and has learnt specifically how to deal with this agent. RLAgent4, however, trained against another RLAgent. This explains why RLAgent4 performs the best against all other agents in the system. This shows that RLAgent4 generalizes the best out of all the models in this experiment.

4.4 Experiment 4: Round Robin with Minimax Algorithm

Similar to Experiment 3, Experiment 4 investigated the optimal agent based on training, with the addition of an RL agent trained against a minimax player. Based on the results from Chen & Lin (2018) where the minimax player outperformed their RL agent, a minimax player was not initially included in our earlier experiment. It was described by Chen & Lin as too costly to train against, as for many epochs, the minimax player would quickly defeat the RL agent, without letting it learn the game. Thus a minimax agent is likely to be overly dominant in its performance against the other models. However, we decided to add it for Experiment 4 to confirm the results from Chen & Lin. Table 4 below displays the results of the round robin averaged over 10 trials and also trained using 10k in-game turns with varying epochs.

Player 1→ Player 2 ↓	Random	Max	Minimax	RL 1	RL 2	RL 3	RL 4	RL 5	RL 6
Trained vs				Random	Max	Minimax	RL1	RL2	RL3
Epochs				1	2	5	5	5	5
Random				95.3	98.7	97.1	50.2	52.3	51.5
Max				79.4	73.2	86.6	23.3	13.5	47.2
Minimax				17.2	24.6	55.1	9.5	11.7	32.5
RL 1	4.7	20.6	82.8		45.5	55.9	56.8	51.4	54.2
RL 2	1.3	26.8	75.4	54.5		60.0	51.1	52.7	54.2
RL 3	2.9	13.4	44.9	44.1	40.0		50.0	43.5	48.0
RL 4	49.8	76.7	90.5	43.2	48.9	50.0		51.9	56.4
RL 5	47.7	86.5	88.3	48.6	47.3	56.5	48.1		49.6
RL 6	48.5	52.8	67.5	45.8	45.8	52.0	43.6	50.4	

Table 4: Effect of training on agent performance through round robin with minimax

Contrary to Chen & Lin's results, the RL agent that was trained against the minimax player outperformed it. This may be due to the reward function evaluating more than just the simple wins and loses, able to evaluate more states of the game (percentage health, and Pokémon), where rewards were given on a per-turn basis instead of a per-game basis. This likely means that our 3rd RL agent was able to learn the game at a faster rate and perform rationally, as opposed to Chen & Lin's PPO model.

Despite Chen & Lin's results, it does make sense that the RL agent which trained against the strongest rational algorithm would have the best overall performance. It understood the choices of the

strongest opponent (minimax player) and was able to play around them, and learned a weak rational strategy to outperform against the other algorithms and RL agents in the round-robin.

It is notable that the rational strategy learned was weak, as although RL agent 3 generally made better decisions than its opponents, there were still instances of suboptimal behaviour in spectated battles. Occasionally, the agent would use moves with no effect, effectively wasting a turn, and letting its opponent make free moves. This is likely the result of the flaws previously mentioned in the reward function, and underfitting, as the agent is not punished by its opponent for letting it move for free.

Another interesting phenomenon here are the results from the fourth, fifth, and sixth RL agents. They performed poorly against the basic rational algorithms, which is likely a result of underfitting and training against somewhat irrational agents themselves. Similar to the ideas discussed in Experiment 3, the RLAgents trained against other RLAgents will perform worse since the original impact of the rational agent is diluted as a result of two RL training processes. Thus, when these irrational agents which were secondarily trained on previous algorithms through their RL agents, perform worse against their rational algorithms.

As seen from the table above, RLAgent3 is able to defeat the Max-Damage Agent 86.6% of the time on average. This meets our initial goal of training an agent capable of defeating the Max-Damage agent more than 80% of the time. As such, we update our specific goal to be the following:

> Train an RL agent that is capable of winning against the Minimax Player 80% of the time.

The current RLAgent3 wins against Minimax Player 55.1% of the time. This shows room for improvement and gives the team plenty of room to explore.

5 Broader Impact

Since Pokémon battling is a niche game, there are not many applications in the real world. Reinforcement learning is useful for analysis and prediction, where the discrete turn-based actions of Pokémon along with the unpredictable nature of which player moves first makes the application niche. Outside of Pokémon AI competitions and entertainment, the usage of these RL agents is not widely applicable. As such, there are limited ethical implications of this RL instance thus far.

However, ethical implications could arise if the agent is deployed on Pokémon Showdown or other Pokémon battling games. Human awareness that they are playing against an agent rather than another human could degrade the competitive spirit and demotivate them towards participating. Furthermore, playing against them would require utilizing and collecting their data, which could lead to potential data privacy issues.

6 Conclusion & Next Steps

Moving forward, various changes to the RL agent could be implemented. Currently, the reward methodology for RL assesses the battle state using the number of Pokémon that have not fainted and their health levels compared to the opponents' Pokémon. This could be further improved by assigning weights to different aspects, for instance, learning to value losing less HP over doing more damage. The reward implementation could also include more or fewer considerations to evaluate different biases and strategies which the model develops as a result of the changes made to how rewards are calculated.

Another implementation is considering further features in the state embedder of the RL agent. The current embedding encodes information about move damage, multipliers, and Pokémon fainting status, totalling to 10 features. This embedding could be altered by adding more features such as held items, the battlefield conditions, weather conditions, trick-room (reverses turn order), entry traps, guard status (defence multipliers), stat alterations, Pokémon abilities, and status effects. Allowing the agent to see more of the battlefield will hopefully allow it to make more intelligent decisions when updating its weights. Furthermore, we could try to save battle states and compare the previous state to the current to see if its decision resulted in an advantage or a disadvantage in helping calculate the reward.

Further optimization of policy hyperparameters (choice of epsilon, tau, increasing training epochs, etc.) will likely be the next immediate step in this project, given the results of the previous experiment regarding the Boltzmann Q Policy, the Epsilon Greedy Q Policy, and their alternatives. The following step in our experiment will be to implement adversarial training with two models and to train a model against itself. The potential benefit of training the adversarial network against itself is mostly for computational reasons. Since the team only needs to train one DQN in this scenario, the training procedure is greatly simplified. Regardless, conducting this experiment in the near future will allow the team to form stronger conclusions.

Lastly, the policies utilized in the RL training were predefined through the Keras-RL library. The team will work towards developing their own policy (taking inspiration from Chen and Lin's PPO algorithm) and adjusting the policy's hyperparameters to improve performance during Pokémon battling.

We have taken this as an opportunity to delve deeper into not only the theory behind reinforcement learning (by researching different types of policies for RL agents) but also how to apply them through the Keras-RL library. We have also successfully familiarized ourselves with the environment set up by the poke-env Python library, ran multiple experiments to train numerous RL agents, and tuned hyperparameters. Ultimately, the team has been successful in its venture to create an effective Pokémon Battling RL Agent that acts as a rational agent. We look forward to working on this project over the summer to eventually deploy it on Pokémon Showdown against real players.

7 Appendices

A.1 Pokémon battling description

Pokémon battles are played as a multiplayer turn-based combat game where players take turns giving orders to their Pokémon in an effort to reduce the opposing Pokémon's Health Points (HP) to zero, where they are knocked out and unable to continue battling. A player wins when their opponent has no more Pokémon left to battle with. Each player assembles a team of up to 6 Pokémon from a pool of 807. Each Pokémon has a unique set of 1-2 types, 4 moves, 6 stats, an ability, and a held item which can affect the state of the battle and how they're most optimally played. There are a total of 18 types of Pokémon (water, fire, grass, etc.), each with its own strengths and weaknesses relative to other types, where some are immune to others. Using an attack with a Pokémon type that is stronger than the opposing Pokémon can yield double the damage and specific moves which are weaker against the opposing Pokémon halves the damage. The multipliers are multiplicative for dual-type Pokémon. For the purposes of this project,

abilities and held items have been neglected as they do not have a clearly defined method of being effectively simulated as a result of having too diverse a range of effects.

A.2 Default parameters

Optimizer: Adam Learning Rate: 2.5e-4 Warm-up steps: 1000	Gamma: 0.5 Delta Clip: 0.01 Target Model Update: 1	Double DQN: True Epsilon (Epsilon- Greedy Q Policy Only): 0.1 Tau (Boltzmann Q Policy Only): 1

A.3 Project Github Repository

https://github.com/JuliaLWang8/Pokemon-Battling-RL

A.4 References

[1] Smogon (2021) *pokemon-showdown,* Available from: https://github.com/smogon/pokemon-showdown

[2] Hsahovic (2021) *poke-env,* Available from: https://github.com/hsahovic/poke-env

[3] A. Kalose, K. Kaya, A. Kim (2018) *Optimal Battle Strategy in Pokemon using Reinforcement Learning*, Stanford University

[4] K. Chen, E. Lin (2018) *Learning to play Pokemon Showdown with Reinforcement Learning*, Stanford University

[5] https://www.youtube.com/watch?v=EE-xtCF3T94&ab_channel=Shaunpants