B.Sc. (Hons) in Software Development

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

PoryAI: A Reinforcement Learning Model trained to play competitive Pokémon.

By
Oisin Hearne

April 26, 2025

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

Contents

1 Introduction **2**
- 1.1 Context . . . 2
- 1.2 Definition of Terms . . . 3
- 1.3 Objectives . . . 6
  - 1.3.1 Understanding Reinforcement Learning . . . 6
  - 1.3.2 Developing State . . . 6
  - 1.3.3 Assigning Rewards . . . 6
  - 1.3.4 Defeating the Expert . . . 6

2 Methodology **8**
- 2.1 Interacting with Pokémon Showdown . . . 8
- 2.2 Representing State . . . 9
  - 2.2.1 State Extraction & Normalisation . . . 10
- 2.3 Creating Policy . . . 10
- 2.4 The Agent . . . 11
  - 2.4.1 Design . . . 11
  - 2.4.2 Self-Play . . . 13
  - 2.4.3 The Expert . . . 13
  - 2.4.4 Training Loop . . . 14

3 Technology Review **15**
- 3.1 Reinforcement Learning . . . 15
- 3.2 AlphaZero . . . 15
- 3.3 AlphaStar . . . 16
- 3.4 Long Short-Term Memory . . . 17

4 System Design **18**
- 4.1 Architecture . . . 18
- 4.2 State Representation . . . 20
- 4.3 Policy Creation . . . 22
- 4.4 Agent Design . . . 23
  - 4.4.1 Neural Network . . . 23
  - 4.4.2 Memory . . . 23

5 System Evaluation **24**
- 5.1 Results . . . 24
  - 5.1.1 vs Expert . . . 24
  - 5.1.2 vs Human . . . 26
- 5.2 Developed Habits . . . 26
- 5.3 Limitations . . . 28
  - 5.3.1 Unpredictability . . . 28

5.3.2 Time & Hardware 29

6 Conclusion **30**
6.1 Summary 30
6.2 Future Work 30
6.2.1 Finding a better Rewards Set 30
6.2.2 Addressing Randomness 31
6.2.3 Improving the Training Loop 31
6.2.4 More Efficient Hardware 32

7 Appendix **33**
7.1 Github & Screencast 33
7.2 Reward Scheme Table 33

List of Figures

2.1 A flowchart of the basic training loop. 9
2.2 Comparison between initial and final reward scheme. Original on left, newest on right. 12

4.1 System Architecture. 19
4.2 The state of the average turn in a Pokémon Battle. [1] 21
4.3 A sample of the starter state, generated using JSON Crack [2] . . . 22

5.1 Rewards and win/loss ratio, vs Expert over 900 battles. 24
5.2 Average Rewards, vs Expert over 900 battles. 25
5.3 The agent's performance against Foul Play. 25
5.4 Randomly Selecting performance against Foul Play. 25

List of Tables

1.1 Pokémon Battle Terminology 4

7.1 The reward scheme for PoryAI. 33

Chapter 1

Introduction

1.1 Context

PoryAI is a Reinforcement Learning (RL) Agent trained to play Pokémon Showdown (PS) at a competitive level.

PS is a free, open source and unofficial website for simulating Pokémon battles, with the purpose of creating a space for competitive Pokémon players to compete without requiring a console or game. PS uses an Elo system similar to chess to rank players.

Comparisons with chess will be a common theme throughout this project. Not because Pokémon is anywhere near as sophisticated or established but because it is a much more widely known phenomenon that aids as an explainer regarding concepts like Game Theory, ELO and Reinforcement Learning.

Reinforcement Learning with chess has been done before with AlphaZero [3], making it especially apt as a comparison. It has also been done with other video games, as seen with AlphaStar [4], a Reinforcement Learning agent that could play the game *StarCraft II* at a highly competitive level. There is also a wealth of information on Chess and AI, and far fewer related to competitive Pokémon and AI.

In this context, Chess and Pokémon differ in their state space and possible actions. In the chosen format (discussed further in 2.4.1), the starting state is random, as though the pieces were scattered about the board in a way that still provided even footing for both players.

There is additionally far more state to consider. Pokémon have their own individual attributes and stats - Their health, the attacks they can use, their stats, and so on.

For example: an ailment is added to chess. A piece can now have its movement limited, an effect persisting for a certain number of turns. This would vastly

increase the number of possible states as each state would need to consider whether each piece has that ailment, and how much longer that ailment will persist. This would mean 64 new fields (2 for each piece at the start of the game).

Now consider that Pokémon has six different possible status conditions, ten field effects that have implications for both sides of the board, and dozens more variables that may change for each Pokémon. The state space for Pokémon battles is vast, presenting a somewhat unique challenge.

Additionally, the number of possible actions the agent has each turn is more similar to Chess. There are a maximum of fourteen options - four moves, six switches, and four modified moves. These options may be reduced by effects that disable individual moves, running out of uses of a particular move, a Pokémon fainting (meaning it can no longer be switched to) and so on. Like in chess, as a battle moves into the endgame (where the Agent has perhaps had several Pokémon faint), the number of options reduce. There are fewer options for switching, potentially fewer options for moves, and modified moves may be absent entirely.

1.2 Definition of Terms

Pokémon Showdown

Free, open-source software enabling accurate simulations of Pokémon Battles [5], for testing purposes and for competitive players to measure their skills. Composed of two parts, a client and a server. A public server is hosted for anyone who wants to use it in order to battle other players [1]. The Showdown team make the websockets for interacting with the server public, allowing bots to connect to the server with relative ease, and for users to build their own clients.

This project will be concerned entirely with the server, which was self-hosted for training. Any further references to Pokémon Showdown will be to the server, not the client.

Elo

Elo is a rating system used commonly in chess [6] and similar games, as well as other E-Sports. It can be used to compare how two different players rank in Pokémon (specifically, it is what is used to rank players on Pokémon Showdown [7]), and determine how likely it is that one player will win against another. AlphaZero has a chess elo of 4500, and the expert system PoryAI is trained against (mentioned in section 2.4.3) has a rough elo of 1300 (it should be noted that traditional expert systems in Pokémon are often easy to predict).

Pokémon Battles

Pokémon Battles are turn-based strategy games, consisting of two players with a certain amount of Pokémon (usually six, on Pokémon Showdown). Players battle Pokémon by selecting what moves to use and making switches when necessary.

There are a variety of strategic factors to a Pokémon Battle. For example, certain Pokémon may set up hazards that affect the opposing team later on in the battle, while certain other Pokémon may be able to clear these hazards. Players must carefully decide what to do each turn, as each move may have ramifications extending into the rest of the game.

Being able to accurately predict what the opponent may do on the next turn is a critical skill for Pokémon battles. There are certain scenarios where a player must make a guess as to what their opponent is thinking, with dire consequences for an incorrect guess. This aspect of game theory is where the initial comparisons to chess originated from and is part of why Reinforcement Learning was chosen for this project.

Battle Terminology

Below is a table of terms and their descriptions in the context of a Pokémon Battle, to aid in understanding what is being described in future sections, as well as to provide examples of how strategy can come into play in the game.

Table 1.1: Pokémon Battle Terminology

Term	**Description**
Turns	At the start of battle and in-between turns, players select their option. These options are then executed in a "Turn" after both players have selected. Additional effects occur, status conditions and field effects activate or tick down. Being able to predict what may happen in the next several turns is important to forming strategy in the game.
Attacking	A Pokémon using one of its four moves on another Pokémon. Can activate certain effects, cause status/volatile conditions or just deal damage.
Switching	A Pokémon switching with another Pokémon in the reserve. This can be done for any Pokémon that has not fainted, meaning that a player's options may shrink as the battle continues and more Pokémon faint.

Continued on next page

Table 1.1: Pokémon Battle Terminology (Continued)

Damage HP	Pokémon have a certain amount of HP, and may take damage from attacks or status conditions, reducing this HP. Once HP reaches zero, the Pokémon faints.
Fainted	Once a Pokémon faints, it is out of the battle. It can no longer be switched to or use moves, and the player must switch to a healthy Pokémon to continue battling.
Effective Moves	Moves can be either "Super Effective" or "Not Very Effective", depending on the type of the move used on the Pokémon. Effectiveness multiplies the amount of damage dealt. Knowing what types are effective against other types allows the player to deal more damage.
Status Condition	A negative condition applied to a Pokémon. Includes Poisoning, Burning, Paralysis, Sleeping and Frozen. Only one status condition can be applied at a time. Poisoning and Burning reduce the Pokémon's HP, while the others limit actions.
Volatile Condition	Volatile conditions are miscellaneous, usually negative conditions applied by some moves. Multiple volatile conditions can be applied at a time. Some are common, but most are rare enough that they were not considered in this project.
Field Effects	Field effects impact both players and can be overwritten by either player. Only one of each type of field effect can be in play at a time. Weather is one such field effect. "Sunny Day", for example, boosts the power of Fire-type moves and activates certain abilities.
Abilities	Each Pokémon has an ability that can be activated under certain circumstances. The ability "Chlorophyll", for example, boosts speed in sun. Figuring out what ability a Pokémon may have can give players an insight into what that Pokémon may do.
Hazards	Hazards are negative effects that players can set up on their opponents. They discourage switching by applying a negative penalty to it. These penalties include damage and stat reduction. Strategies can often revolve around setting or clearing these hazards.

Continued on next page

Table 1.1: Pokémon Battle Terminology (Continued)

Stat Boosts	Pokémon can boost their own stats, or lower the stats of their opponents, through many different moves. Boosting a Pokémon's stats can make it's power increase greatly, allowing it to faint multiple Pokémon in future turns.

1.3 Objectives

1.3.1 Understanding Reinforcement Learning

One of the main objectives of this project was understanding how Reinforcement Learning functioned, how to build an agent correctly and good practices for configuring policy and rewards. This involved a good deal of research, which is expanded on further in 3.

1.3.2 Developing State

Developing a state template that could reflect all of the relevant fields that apply to the average turn in a Pokémon battle was an early objective for the project, taking data from PS and representing it in a form that could later be flattened and normalised for use in RL.

This was especially challenging given that half of the state had to be extracted from the turn data, while the other half was given by the websocket itself. Data extraction is further elaborated on in 2.2.1.

1.3.3 Assigning Rewards

In researching Reinforcement Learning, it was discovered that a reliable rewards mechanism was required for the agent to learn from and improve itself. This meant examining the new state each turn, and assigning a reward based on what had occurred.

Tweaking these rewards was necessary on several occasions as the project progressed, with the objective being to find a set of rewards that allowed the agent to win more consistently against the expert.

1.3.4 Defeating the Expert

The ultimate goal of the project was to build an agent that could reliably defeat the expert and possibly achieve an above-average elo on the main website for Pokémon Showdown. As the project progressed, it became clear that this was not

an achievable goal within the time given. Instead, the goal shifted into creating an agent that could win against the expert more often than a randomly-selecting agent could.

Chapter 2

Methodology

2.1 Interacting with Pokémon Showdown

Pokémon Showdown (PS) uses websockets for client interactions, sending information about state as well as requests for actions to clients. Clients then send commands to this socket in order to execute certain actions.

How these websockets function is well documented by the PS team [8]. Creating functions to connect to Pokémon Showdown required researching the documentation provided by the PS team, as well as observing how other Pokémon Showdown related projects connect to the websocket. The expert system *Foul Play* features an example of how a connection can be made and maintained with the websocket, and was used as a reference in cases where formatting was unclear.

A set of python functions were built to connect to the websocket, log in, start a battle, and send decisions. This was the first piece of functionality built for this project, and required a good deal of trial and error in order to make sure the right information is parsed, that commands were sent correctly, and that the websocket did not expire (A recurring problem later on, when training was done for hours at a time). This was done in a jupyter notebook that is still available to view on the project Github.

These functions were refined such that once logged in, the agent could run continuously by starting battles over and over. Once a battle has been entered, the application begins to parse each incoming message to determine what to do next - is there new state to record, has the battle ended, or is there another decision to be made? This loop is represented by figure 2.1.

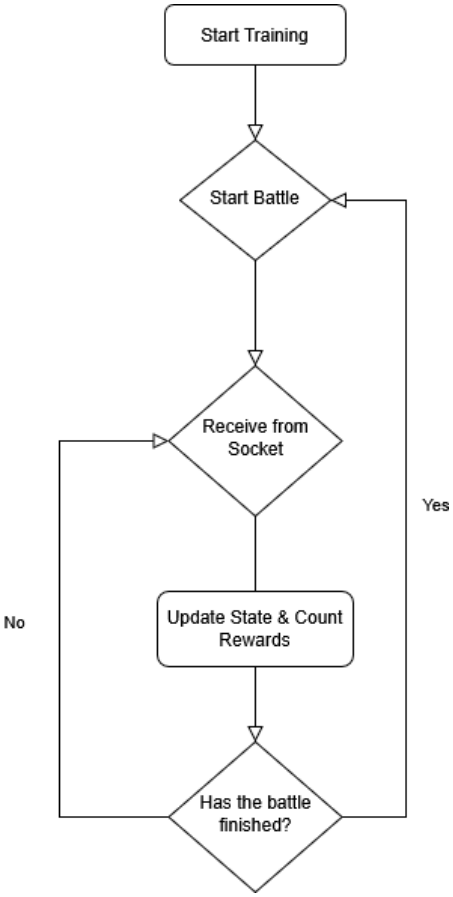

Figure 2.1: A flowchart of the basic training loop.

2.2 Representing State

Representing the state of the average turn in a Pokémon Battle is a difficult task. There are numerous conditions and effects that can apply to a given Pokémon, and there are 12 Pokémon in play during the average battle. In order to reduce the size of the state, certain conditions that do not see regular play (such as Mimic, an obscure condition that is not used in competitive play) were disregarded.

While some states are ephemeral and disappear when a Pokémon leaves the battle, these states still need to be recorded in order to ensure a consistent state size for the agent to interpret later on. This inflated the state size significantly. The final state that was arrived at is described in figure 4.3. This was gathered from the conditions that could apply during the average PS battle, as well as the state that was extracted from PS.

The final state size of 671 is large, but when considering that Reinforcement

Learning is often used with images (and a size of 671 is roughly equivalent to a 15*15 RGB image), it was not too much of a concern.

2.2.1 State Extraction & Normalisation

State Extraction was performed in two different areas - from the data sent by the PS websocket, and from the data gathered each turn. This had to be done as the PS websocket does not send all of the information that a player would normally have at any point of the battle.

It only sends the information required to make a decision, that is, information about the player's side of the field (their active Pokémon and their reserve Pokémon). While this is better than nothing, it leaves a lot of work for manual extraction of the other features under consideration.

The rest of the features are extracted as they appear each turn. When the opponent Pokémon first appears, it is recorded in the state. When it uses a move that hadn't been seen before, this is also recorded, and so on. This is also where information about the general state of the field is recorded - what hazards and field effects (weather, terrain and trick room) are in play.

When a battle begins, the state manager loads a fresh copy of the state (that is, an example of the state with every field being zero). Then, state is filled in depending on the current scenario. Extracting data from the PS websocket involves a 1:1 translation of all of the information received into the state, while extracting from a turn involves checking for every possible string that could mean that a new piece of information has been revealed. All of this information is stored in a python dictionary with entirely numerical values. For example, in place of a Pokémon's name, an ID number is used.

This state is used by the agent, so it must be processed first. This involves flattening the dictionary into a vector, and normalising the numerical values into a range from zero to one, as a fraction of their maximum values.

2.3 Creating Policy

Policy is one of the main considerations of reinforcement learning and it must be thought about carefully. A policy that is missing important features or incorrectly weighted can severely mislead an agent. Shaping a good policy took several attempts and the policy grew continuously as problems arose through the course of the project.

The initial naive approach to the policy involved a set of flat values given/taken away for performing certain actions (dealing damage, boosting stats, setting effects), as well as a large bonus for knocking out Pokémon, and a larger one for

winning.

This led to an overarching issue that plagued the policy structure for quite a while - the agent would figure out that switching was generally a safe move, and it would continue switching for as long as it could. This led to no battle progress, while the opponent would continue to chip away at the agent's Pokémon until eventually winning.

This issue required a harsh negative reward in order to discourage the behaviour. Further negative rewards would be given for consecutive switches, and a battle would be "tainted" slightly if too many consecutive switches occurred. This later had to be done for moves, in order to prevent the agent from picking a particular slot and never using any other move, regardless of outcome.

Further tweaks were made to the rewards to encourage strategic play. Setting field effects and statuses were given further rewards. Additionally, rewards were seperated into turn-by-turn rewards and overall battle rewards. Turn-by-turn rewards are given based on the outcome of a move that turn, but overall battle rewards were given by measuring the performance of the battle itself. Whether the agent won or lost, how long the battle went on, and whether the agent used any behaviours that should be discouraged (overusing switching or a specific move slot, for example).

In order to impress how the rewards policy has grown over the course of this project, see figure 2.2 where the initial rewards are compared to the final ones.

2.4 The Agent

2.4.1 Design

Once all of the above features (Websocket connection, State Extraction, and Policy) were functioning correctly, work began on the agent. Designing an RL model required a good deal of research into PyTorch and other libraries. While quite a lot of resources exist for building RL models, most assume an image as the state (the stick balancing example was a common one, for instance) which required some adaptation into the format being used.

Battle Format

PS has many different battle formats offering different play styles. The two most popular by far are the latest generation's (currently Generation 9) *Random Battles* and *OverUsed*.

OverUsed as a format is far more competitive, but involves a whole other factor to consider in *teambuilding*. The player must pick and configure the Pokémon that

Figure 2.2: Comparison between initial and final reward scheme. Original on left, newest on right.

make up their team, considering their roles, the general team composition, and so on. Given that this is heavily influenced by the current state of the game, which changes often (determined by The Pokémon Company's release schedule), the other popular format was considered.

Random Battles are a lot simpler in that both players are given a set of six random Pokémon from a list of many (there are 507 Pokémon, which can be found in the format's JSON on the Pokémon Showdown GitHub [5]). Player's may get good or bad Pokémon, but battles themselves can often be determined by random factor and Random Battles are still considered a relatively competitive format. Additionally, a constantly changing team forces the agent to consider the type match-ups, rather than just the pokémon themselves, allowing it to be flexible with several different teams if switching to an OverUsed format was ever desired.

Once trained to a sufficient degree in Random Battles, it in theory would not be difficult to switch it to an OverUsed format - the general battle and training

loop would stay the same, with the additional consideration of what team the agent should use.

Neural Network

The Neural Network was initially much simpler, being four layers deep with alternating linear and ReLU layers, and without dropout. This was not sufficient in capturing the complexity of Pokémon Showdown. The current version of the network instead features LSTM (Long Short Term Memory) at the input layer, in order to better form long-term dependencies. The number of nodes in each of the hidden layers was doubled, and dropout (of 0.2) was added.

2.4.2 Self-Play

Self-Play is a strategy wherein a Reinforcement Learning agent battles itself, with the goal being that as the agent improves, so too does its opponent. This is used often for similar goals, such as in training a chess RL agent [3].

In PoryAI, a separate training loop is used for Self-Play. Two agents are created, but only one is actively trained and adjusted. The second agent (the opponent) instead copies the weights of the first at certain intervals throughout the training process (every 100 battles, at time of writing).

This means that the two agents will always be similarly skilled. If successful, an even win rate with a growing average reward should be seen. In order to keep the win rate even, the epsilon of the first agent is increased if the win rate falls out of a certain range (30%-70%).

Through Self-Play, the agent should gain some understanding of certain game mechanics - such as type match-ups, field effects, statuses and so on.

2.4.3 The Expert

While Self-Play allows the agent to improve up to a certain point, better training is needed in order for the agent to gain some more strategic knowledge. The play style that may develop between two agents in Self-Play is one that may not perform well against human players, so a style more similar to actual strategic play is necessary.

Foul Play (referred to as the expert), developed by pmariglia [9], is an AI that uses the ExpectMinimax algorithm (though it can also use a Monte-Carlo Tree Search) to select a move on each turn of a battle. It runs it's own internal simulation of the battle to look ahead at the possibilities for the next few turns, and uses this information to decide what to do.

Foul Play runs fast, doesn't use much computational power, and scores relatively well against players with an ELO of 1300. It utilises strategies and game mechanics well and required little modification in order to function as an expert 'coach' for PoryAI.

2.4.4 Training Loop

The training loop was given different functionality based on how the agent needed to be trained - a Self-Play mode, a vs Expert mode, and a random mode for achieving a baseline to compare results to.

The training loop needed to be calibrated to allow for consistent growth and exploration, with a renewed epsilon every 1000 iterations based on the performance of the model. In Self-Play, the best model is saved every 500 iterations and renewed if the current model is performing worse than a previously saved one. This occurs in vs Expert mode too every 1000 battles.

Additionally, data is gathered often in order to generate plots and measure how the model is performing - how many times it performs ideal or unideal behaviours, like using super effective moves or switching too many times in a row.

Initially, the training loop only involved PoryAI battling the expert. However, the expert was too sophisticated for PoryAI to ever initially overcome when it was still developing a strategy and selecting mostly randomly. Self-Play was introduced as an initial step, allowed to run for a certain amount of battles, and then the expert was introduced to attempt to get it to refine battles.

Future versions of this training loop may adaptively switch between the two, running Self-Play for a few battles, then a few expert battles and switching back. This would allow Self-Play to adapt to the strategies learned from the expert, while still allowing room to grow rather than getting consistently defeated by the expert. As of time of writing, this has been done manually by allowing the program to run overnight in Self-Play mode, then switching it to vs expert mode for the day.

Chapter 3

Technology Review

3.1 Reinforcement Learning

Reinforcement Learning (RL) is a form of machine learning. Agents learn how to make decisions based on the rewards they receive for their actions [10]. The goal of the agent is to learn a policy that maximises its reward. This is done through trial and error, the agent takes different actions in different states and sees what the best action to take is.

Reinforcement learning is a good fit for strategic turn-based games as the agent can learn from its mistakes. From playing against itself, it can learn what actions are good and what actions are bad. This is Self-Play, a common strategy utilised with Reinforcement Learning [11]. This is particularly useful in games where a competitive element is present. The agent can learn from its own losses and wins in order to improve its performance. This makes it a good fit for Pokémon Showdown, given it is a turn-based strategic game. Agents can be pit against each other in order to develop their strategies and form the best possible policy. PoryAI itself uses Deep Q-Network (DQN), a type of RL algorithm where a Deep Neural Network is used to approximate the Q-Values (the expected rewards for a given action in a given state) [12]. This is a good fit for PoryAI as the state space is discrete (the possible states are distinct and finite, even if the number of states is very large). DQN allows the agent to learn from its own experiences, without requiring initial training data.

3.2 AlphaZero

AlphaZero is a reinforcement learning algorithm that uses a Deep Neural Network in order to learn how to play strategic, turn-based games. It was based on AlphaGo, which was specialised to play Go, while AlphaZero can be adapted to a

variety of different games [3]. AlphaZero learns via Self-Play, meaning that there is no initial training data requirement. AlphaZero instead plays against itself, and learns from the results of those games. This Self-Play method of training is particularly suitable to Pokémon Showdown. While Pokémon Showdown makes battle replays public, they are not stored for very long. It would be difficult to scrape the amount of training data required to train a different kind of model. Previous Pokémon Showdown projects have had to petition the site's developers for this data, but even then it was limited. Additionally, the state size and random battle format would make scraped data less useful. The amount of data required to train a model would be enormous. Given the limited hardware being used, it would be difficult to train a model on this data.

PoryAI differs from AlphaZero in that it doesn't use a Monte Carlo Tree Search (MCTS) for decision making, as the options in a Pokémon Battle are limited to a known number of actions. PoryAI instead uses its DNN to predict the best action to take in a given state. AlphaZero's MCTS is sophisticated, simulating the game many times using Self-Play to find the best action to take. This is not something that was used in PoryAI, though it is possible as simulators for searching through Pokémon Battles do exist (poke-engine being one such engine [13]) and is a potential area of future work, which would possibly improve the model's decision making. This type of in-depth search may be too much for the present hardware. It is worth noting that AlphaZero was trained on about 5000 TPUs [3], hardware specially designed for machine learning. PoryAI was trained on a single desktop computer, so not all of the features of AlphaZero could be replicated for a Pokémon Showdown environment.

3.3 AlphaStar

AlphaStar by contrast with AlphaZero uses a combination of both supervised learning and reinforcement learning. In a game with as many states as *StarCraft II*, it would be next to impossible for a Self-Play entirely reinforcement learning model to stumble into a good strategy. It learns from humans first, then begins Self-Play. Another difference between AlphaStar and AlphaZero is that AlphaStar features a rewards system. AlphaZero is trained to win and rewarded only for winning, while AlphaStar receives a reward for implementing certain strategies [4]. PoryAI features a similar reward system, rewarding the model for carrying out certain strategic moves, though longer-term strategies are currently not accounted for. AlphaStar also uses league training, where the model is trained against several varieties of itself. This was considered as a possible strategy for PoryAI, but it was quickly found that the hardware was not sufficient to train multiple models at once. Finally, AlphaStar uses a recurrent neural network (RNN) rather than a DNN. This

fits well with StarCraft II as a real time strategy game, and a combination RNN and DNN was used for PoryAI due to the sequential nature of Pokémon battles. Strategies are executed over several turns, so a memory of the previous few turns is required to make the best decision.

It should be noted that both AlphaZero and AlphaStar were built by teams of people, and were trained on far better hardware than was available for PoryAI. That being said, research into both of these models offered understanding of how to build a model such as PoryAI, and the way to go about training it.

3.4 Long Short-Term Memory

PoryAI utilises a Long Short-Term Memory (LSTM) network as part of its DNN. LSTM networks are a type of RNN able to form long-term dependencies. The general idea is that LSTM networks are able to remember representations of the past in their hidden state, and use this information to make decisions about the future [14]. This is important for Pokémon battles, as what happened in the last few turns can have a large impact on the current turn. For example, if a Pokémon has just used a set-up move, it should remember this and take advantage of that set-up, instead of simply setting up again. LSTM also learns from sequences of data, ideal for Pokémon battles where moves are made in sequence. This in theory allows the model to learn that a certain sequence resulted in a win, and to repeat that sequence in the future.

Chapter 4

System Design

4.1 Architecture

The PoryAI architecture consists of several different python classes, each taking on a different role.

Showdown.py

Responsible for logging into Pokémon Showdown and managing the websocket connection to the server. From here, the state is passed onto interpreter.py, and decisions are received from agent.py and passed on to the server.

Features a *manage_ battle* loop, wherein everything received from the websocket is checked - is it some new state, is there a decision to be made, or is the battle over?

A showdown object is initialised with a URI (where to send log-in data to), a username and password, a websocket address, the format to battle in, whether the agent starts challenges or receives them, whether or not to print logs to console, and the user to challenge/receive challenges from.

Interpreter.py

The interpreter manages state and rewards, interpreting the state from showdown.py into a manageable format (resembling a JSON) and applying rewards for a given turn data.

Rewards are applied by iterating through everything that occurred within a turn (turn data is stored in an array, for each line of the turn).

Keeps track of certain battle stats, such as how many times the agent has used a super effective move, how many times it's switched concurrently, and so on.

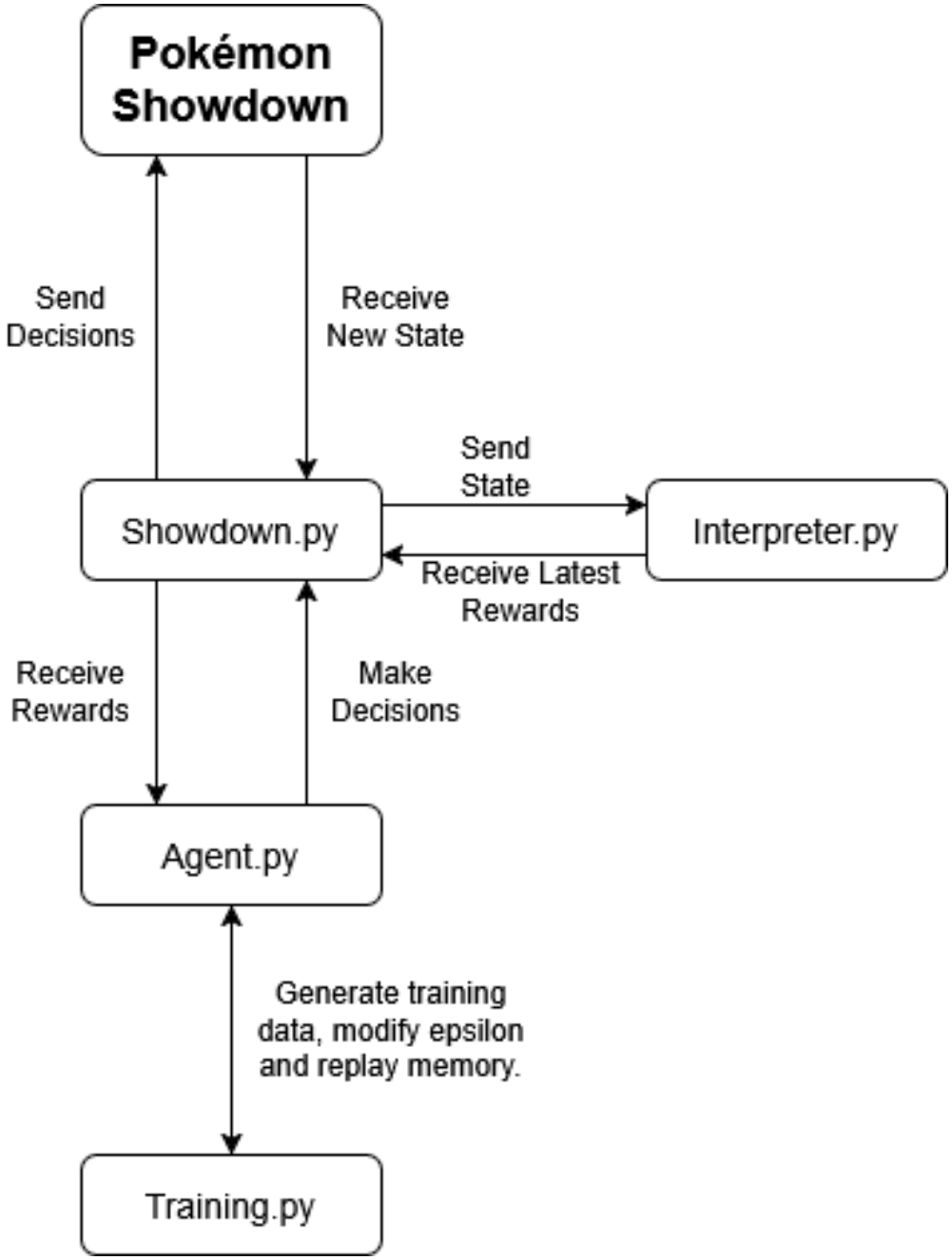

Figure 4.1: System Architecture.

Agent.py

The class containing all of the logic for the agent. *PyTorch* is the library used for the model, allowing a Neural Network to be built, state to be fed into it and

outputs to be received from said network.

The state is flattened and normalised from a dictionary into a 1-dimensional vector, with every value between 0 and 1. This is a necessary step to make sure that initially, no particular field has any more importance than the rest in the agent's view.

It was necessary to introduce some biases to the agent via the *act* method. The agent would initially prefer options with a higher index, resulting in the agent continuously switching without attempting any other option. The lower-index options were therefore given a slightly higher weight to make up for this.

This class additionally contains methods for the agent to append new data to memory and replay it in order to learn from it, as well as helper methods for retrieving batches of memory at a time and saving/loading the model/memory.

Training.py

Training contains several different training loops used to train the agent in a variety of ways, allowing for Self-Play, vs Expert and Random modes. These training loops keep track of the current number of iterations (battles) and tweaks the Epsilon of the agent based on certain metrics every 500 iterations. In Self Play, the loop keeps track of which 500 battles had the best average reward, takes a checkpoint, and loads the model from said checkpoint if it stops improving later on.

This was originally done with win count in vs Expert mode as well, but this was changed to use the same reward-based function after it was observed that the agent "accidentally" winning earlier on frequently meant that the agent would return to the first checkpoint over and over again.

This class also contains certain helper methods for outputting data in order to measure how well the agent is performing. Scatter charts are produced every 50 iterations, showing the current win rate and the current average reward per battle. Statistics about how the model is performing - such as how many times it has used a super-effective move, are also stored along with the current epsilon and win count.

4.2 State Representation

While directly using the state sent down by the PS websocket was considered, it quickly became clear that a unique design was required in this case. As previously discussed, Reinforcement Learning requires that the input be of a consistent size. The output size from the websocket was inconsistent and therefore could not have been used, so a new solution was required. The figure 4.2 shows some of the state present on the average turn of a Pokémon battle, but there is even more that

couldn't be shown - the moves of each of the player's reserve Pokémon, their stats, field effects and so on.

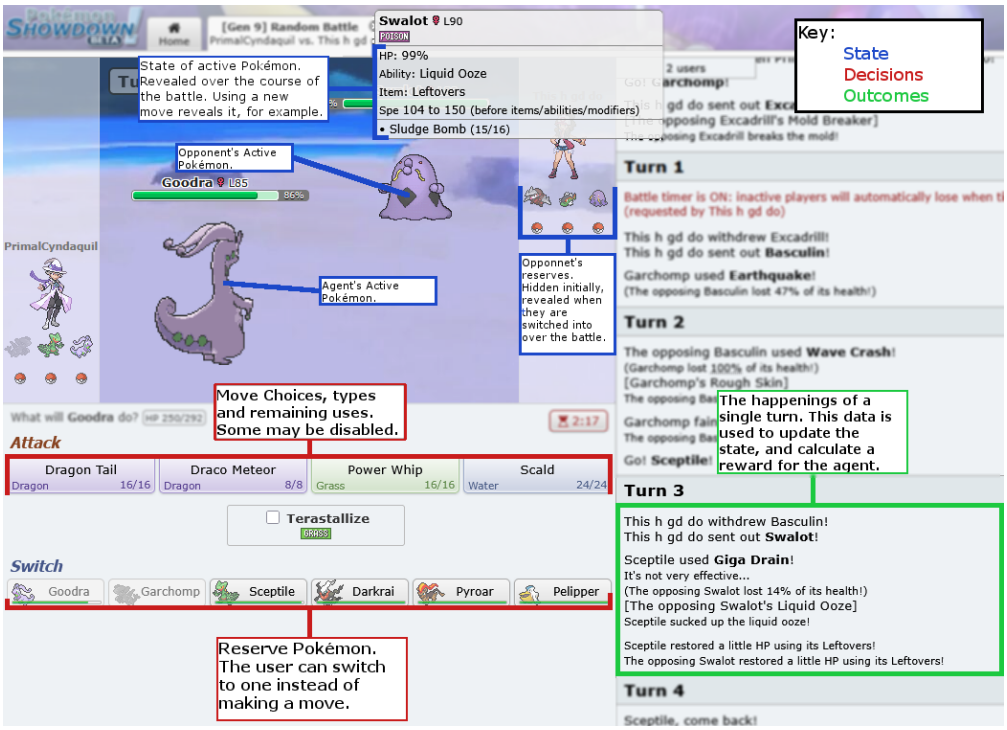

Figure 4.2: The state of the average turn in a Pokémon Battle. [1]

The state was split into three sections: the player side, the opposing side, and universal. The player side encompasses the agent's active Pokémon, its reserve Pokémon, and any effects on its side of the field. The opposing side covers the same details for the opponent (filled out as those details are discovered), and the universal section contains any universal effects, such as weather and terrain.

This sample state also contains a request section, changing based on what the PS websocket is requesting. These requests are usually either *active* or *forceSwitch*, corresponding to the user being able to do whatever they'd like or the user being forced to make a switch.

The player side of the state is updated as requests come in, while the opposing side and universal sections are updated from turn data as the battle continues.

A visualisation of part of the initial state JSON has been included below. Note that fields are zero until they are filled out through the course of the battle (to keep state size consistent). This is only a small selection of the state - there are also ten more Pokémon to consider with just as many fields as the active Pokémon, as well as four moves per Pokémon, with each move having five fields. These details were left out to ensure that the graph could fit within the figure and still be legible.

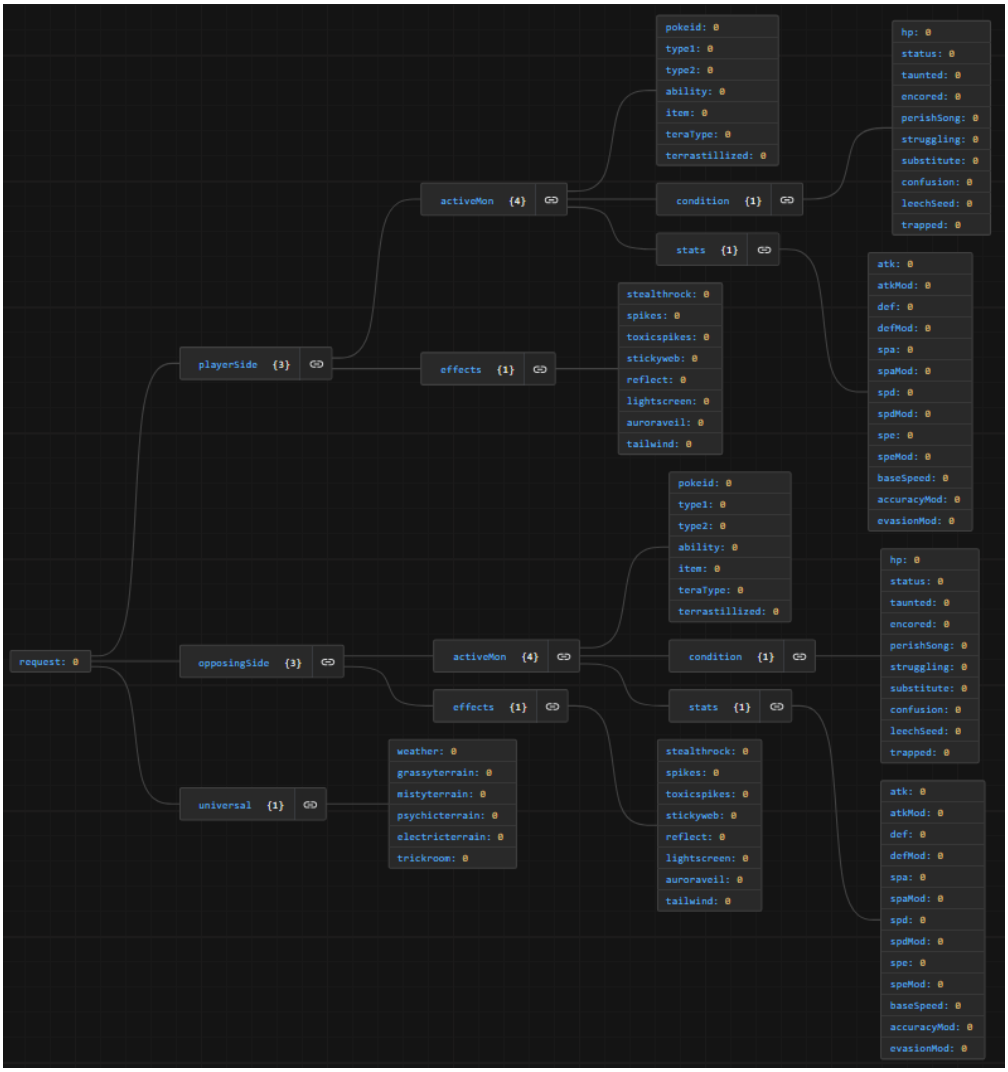

Figure 4.3: A sample of the starter state, generated using JSON Crack [2]

4.3 Policy Creation

The rewards policy for the agent was designed with the goal of encouraging strategic, long-term play and discouraging bad habits that the agent had been observed to form over longer training sessions (such as repeatedly using the same move or switching constantly).

The table 7.1 in the Appendix section features the complete rewards policy for PoryAI, as well as a description of why each value was set as it was. This can be used as reference to understand more about Pokémon battles and how the agent has been trained.

4.4 Agent Design

4.4.1 Neural Network

The neural network used is a combination Recurrent Neural Network and Deep Neural Network. This allows the agent to learn from sequences of data, so that the agent has some memory of what it has just done. This is important for Pokémon battles - has the agent just set up? If so, it should possibly attack now instead. It consists of a Long Short-Term Memory (LSTM) layer, followed by a fully connected layer. The LSTM layer is used to learn from the sequence of data, while the fully connected layer is used to make the final decision. The output of the network is a vector of Q-values, one for each possible action. The action with the highest Q-value is chosen as the action to take. Pytorch's nn module was used to build the neural network, allowing for easy creation of the aforementioned layers. The weights of this network are stored every 50 iterations, creating a checkpoint of the model for later use.

4.4.2 Memory

The memory of the agent is a list of tuples. Each of these contains a state, action and then the result of that action: the reward, the result (the next state) and whether or not the battle was concluded. Memory from winning battles is saved multiple times, such that the agent will remember how it won a battle, and be able to replicate that behaviour later on. This is done in training.py. Checkpoints of the memory and model are saved every 50 iterations. This was done more often than perhaps necessary, in the interest of not losing too much progress in the case of a crash (see section 5.3.2 for details on limitations with the hardware the agent was running on). These memory checkpoints are used such that the agent can switch from Self-Play to vs Expert mode, and vice versa. This is done manually at the moment, but a training loop featuring a switch between the two modes every certain number of iterations could be done in the future. Memory is replayed in batches of 512, with half of this batch being sampled from the agent's memory at random, and the other half being whole-battle samples. This allows the agent to learn long-term strategy by remembering the course of a whole battle, as well as how individual moves may have affected the outcome of a battle. These battles are weighted by the Whole-Battle Performance rewards discussed in the previous section, meaning that the agent will learn more from battles where it performed well according to those metrics (whether or not it won, how long the battle lasted, etc).

Chapter 5

System Evaluation

5.1 Results

5.1.1 vs Expert

The agent has a winrate of 1-3% against Foul Play. The variance comes from the inherent randomness of the game, as well as the agent becoming traumatized in certain situations by the steep learning curve between Self-Play and expert battles. If the agent wins early on, it is more likely to win again soon after, but if it goes on a losing streak, it will continue to degrade until it falls into bad repeated habits in an effort to minimise the negative rewards it is receiving. This pattern is demonstrated in the below figures 5.1 and 5.2. The former shows that wins are clustered together (represented by green dots and large spikes in the rewards), while the latter shows a clear upwards curve as the average reward per battle increases.

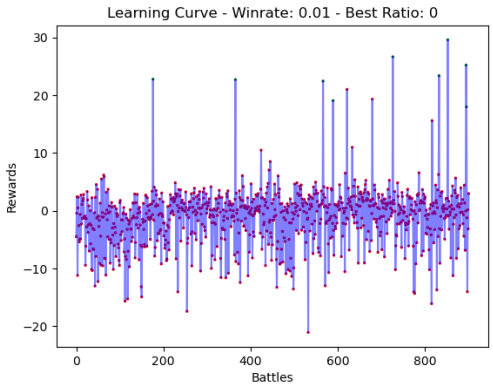

Figure 5.1: Rewards and win/loss ratio, vs Expert over 900 battles.

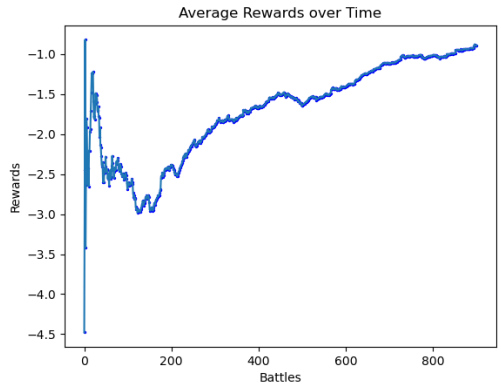

Figure 5.2: Average Rewards, vs Expert over 900 battles.

Foul Play has an Elo of 1300 (as of 2019, likely improved since [15]), roughly equivalent to a below-average but still competent human player. After several rounds alternating between Self-Play and expert battles, the agent won against the expert 1% of the time, used super effective moves 0.99 times in battle, and fainted 1.16 Pokémon per battle. By contrast, a randomly selecting agent battling against the expert won 0.13% of the time, used super effective moves 0.7 times in battle, and fainted 0.66 Pokémon per battle. These statistics are from figures 5.3 and 5.4, which come from the output of PoryAI's training loop. These figures demonstrate clear improvement in the agent. It has been able to learn to use super effective moves more often, and is performing better overall than a randomly selecting agent, which serves as a baseline to compare against.

```
Wins This Cycle: 7
Battles: 700
Epsilon: 0.26291276629926075
Stats: {"wins": 7, "rewards": -705.5303458872057, "winsThisCycle": 0, "epsilon":
0.26291276629926075, "stats": {"superEffective": 99, "resisted": 159, "fainted":
116, "boosted": 516, "damaged": 1166, "setHazards": 26, "clearedHazards": 0,
"switched": 15, "repeatMoves": 620})
```

Figure 5.3: The agent's performance against Foul Play.

```
Wins This Cycle: 2
Battles: 1500,
Stats: {'superEffective': 1059, 'resisted': 3649, 'fainted': 996, 'boosted':
5552, 'damaged': 14108, 'setHazards': 358, 'clearedHazards': 23, 'switched':
5467, 'repeatMoves': 1464}
```

Figure 5.4: Randomly Selecting performance against Foul Play.

5.1.2 vs Human

Data against human players was gathered by having the agent join the Random Battle ladder on the main Pokémon Showdown server. In 12 hours of vs Human play, the agent achieved a winrate of approx. 1.7% and an Elo ranging from 1050-1090 in the Gen 9 Random Battle format. This is barely above the Elo floor of 1000, but demonstrates that the agent was still winning occasionally. Over the course of 170 battles, the agent used super effective moves 0.98 times per battle, and fainted 1.5 Pokémon per battle. This is a much smaller sample of battles compared to the other methods, because battles against human players take far longer (as they have to take time to think of their next move as well as manually input it). A better win ratio could be achieved by allowing the model to run on the main Pokémon Showdown server for several days, but as Showdown restarts every now and then, in it's current state PoryAI would require some manual intervention to get it to reconnect. A further possible inaccuracy in this win ratio comes from the fact that other bots may be in the Random Battles queue too, skewing the results. Additionally, forfeits and timeouts from other players were counted neutrally. This was to prevent the agent from getting incorrectly rewarded for winning a battle it did not win.

5.2 Developed Habits

The following are all observed behaviours that the agent has developed over the course of training and are not necessarily optimal or representive of the model as a whole.

Healing Addiction

The agent has developed a habit of overusing healing moves. It gets rewarded for restoring HP, and naturally lasts longer during the battle when using them, only giving it the opportunity to use these moves more. This results in the agent using healing moves even when it is almost at max HP. This is not a good strategy, but does demonstrate that it has begun to associate move IDs with their effects. In Self-Play, it will typically use these moves until they run out of uses. This is one area where it has clearly not learned higher level strategy, as stall tactics require the agent to only use these moves when absolutely necessary.

Boosting Moves

Certain moves boost the agent's stats multiple times, such as Shell Smash. The agent has learned that it gets a massive reward for using these moves, and will use

them repeatedly. This is especially a problem with Shell Smash as it is usually used only once in competitive play, as using it more times is risky and unnecessary. In a similar vein, Multi-Hit moves trigger the *damage* reward multiple times, meaning that the agent will use them even when they are not optimal. Accomodating for this would require some heavy changes to the reward system, which is due for a refactor in the future.

Hazard Setting

The agent has learned that setting hazards is good and that it will get rewarded for doing so (with possible additional future rewards as the hazards take effect). It has identified the sample of moves that set hazards, and will overuse them. This is poor strategy, as it will use hazards even if the opponent is in a position to clear them immediately, or punish the agent for using them.

Type Effectiveness

The agent has gained a rudimentary understanding of type effectiveness. It has learned to use super effective moves and avoid using ineffective ones, demonstrating a burgeoning understanding of the game. However, it will often go for a 'safer' move even in cases where the super effective move would result in a faint. This is likely due to the aforementioned issues with other moves, like set-up moves, healing moves and hazards. Certain additional effects may also influence these decisions. If a move additionally causes a status effect 10% of the time, the agent may choose to use that move more often due to a previous randomly-given reward for using it. PoryAI will also often make a switch that puts it into a better position type-wise, even if it goes on to not utilise that type advantage. This is another habit showcasing that PoryAI has picked up positive behaviours, but often in negative ways.

The above habits demonstrate that, while the agent is certainly learning, there are certain flaws in the rewards schema as well as in how the agent is trained. Poor habits are often formed when the agent goes from Self-Play to Expert battles, as the steep learning curve is difficult to overcome.

5.3 Limitations

5.3.1 Unpredictability

Randomness in Pokémon

In order to keep battles interesting to players, Pokémon battles feature a good deal of randomness. Moves may sometimes miss, or may not do as much damage as expected (these are *Damage Rolls*). This means that the agent may make a move that was optimal, but the outcome of the move may still be negative. This would discourage it from using that optimal move in the future. The project as it currently stands does not take this into account, and so the agent may avoid moves that are actually optimal as it may have been punished incorrectly for using them. Foul Play addresses this as it uses a ExpectiMiniMax algorithm to determine what move to use, something ideal for dealing with randomness in games. The agent does not have this ability.

Randomness vs. Expert

Expert systems like Foul Play are often limited in how they can respond to random behaviour from opponents. They are designed to play against human opponents with some semblance of strategy, and may be completely thrown by how a random opponent plays. Randomness may occasionally be ideal. Foul Play itself is built to occasionally select moves randomly, as to not be too predictable for human players. This becomes a problem in that the agent is random when it begins training. The higher the epsilon of the agent, the more it explores, but the more it may win accidentally by making moves that are not optimal. This means it may learn the wrong lessons from how it battles early on. This problem has been observed during training: as training begins, the agent will win more often as it learns some good behaviours, but still occasionally makes completely random choices that confuse the expert. It cannot meaningfully learn from these battles, and ends up winning less often as it gets less random later on.

Random Battles

The chosen format of Random Battles is a highly unpredictable one. In most formats, players are able to build their own teams but mostly stick to a set of Pokémon that are considered viable in the format [16]. In that case, the agent could learn to play against the common Pokémon in the format. Recognising a certain Pokémon and what that Pokémon is likely to do, for example. In Random Battles, both players are given teams of random Pokémon. Developing familiarity with the Pokémon that appear more often is far more difficult, as the agent may

not see a Pokémon again for a long time. There are 507 random Pokémon in the chosen format, while in OverUsed there are only a certain amount of Pokémon that are considered viable. An agent in this case may develop more of a familiarity with, for example, Gliscor (a Pokémon that appears in approximately 30% of battles in OverUsed [16]) than it would with a Pokémon that appears in only 1% of battles.

5.3.2 Time & Hardware

Training an agent is a time-consuming process, especially given the large amount of state to consider and the hardware restrictions present. The PC being used has a single GPU, and a limited amount of RAM and CPU. This meant that each battle took longer, and therefore less battles could be run in the time alloted. Every small change to the agent's reward scheme required retraining, taking even longer. As mentioned in 3, other similar projects have used large amounts of TPUs in order to train their agents. Additionally, the PC has lately been suffering from hardware issues, with several crashes during training, making training for long periods of time even more difficult. A virtual machine was provided, but could not be used effectively as Azure restarts the VM every few hours when the user is detected as idle. This meant that training could only go on autonomously for a few hours at a time.

Given more time and more powerful hardware, the agent could be trained for longer periods of time to better understand how each iteration of the agent is performing, possibly allowing for multiple different agents to be trained at once. As it is, the training process takes far too long to properly examine multiple different reward schemes and functionality.

Chapter 6

Conclusion

6.1 Summary

The goal of PoryAI was to create a Pokémon Showdown agent that could play the game at a human level through the use of Reinforcement Learning. This required creating a representation of state, a reward system, and a training loop that could allow the agent to learn as much as possible. While some growth was clearly demonstrated 2.2, the agent was not able to reach anywhere near the level of a competitive player, and struggled even against regular players. This is likely due to a number of factors, such as the limited hardware and time available for training purposes. These limitations are discussed more in 5.3. It became clear that the agent was not going to be able to learn to a sufficient degree in the time available, and so the goal was shifted to instead demonstrating clear improvement in certain specific areas - it learned to use super effective moves more, it fainted more Pokémon and it won more often than a randomly-selecting agent.

6.2 Future Work

Given more time to develop PoryAI, there are several areas that could be improved on with the goal of improving how the agent is learning and what strategies it is developing.

6.2.1 Finding a better Rewards Set

The rewards as currently outlined are likely imperfect. There are certainly more ways to reward the agent for more strategic behaviour, and the current rewards policy is relatively simple. It is rewarded based on the move it uses, but the current state context is not taken into account. For example, it is generally a bad idea to

use set-up moves (such as hazards or boosts) when the opponent Pokémon has the ability to KO the agent's Pokémon. This is not currently taken into account. The agent will be punished for fainting, of course, but it may not be punished for its move choice. This means that the agent may learn that these moves are good, but it will not use them in the appropriate situations. Developing a more robust set of rewards would be a good way to improve how the agent is performing, but would require careful consideration and time for training, as well as perhaps consulting better Pokémon players. The code for assigning rewards needs a general refactor. It is currently rather monolithic, difficult to read and inefficient. A refactor could clean up the code and fix some of the aforementioned issues, such as the multi-hit moves and correctly implement damage thresholds.

6.2.2 Addressing Randomness

To make the training process smoother, randomness can be reduced in certain controlled environments. For example, starting training with a set scenario of Pokémon. This comes with the risk of overfitting to that scenario, but allows the agent to learn the basics of battle by how it can win most consistently in that scenario. Additionally, working within an OverUsed format brings with it more complexity, but less randomness. The agent may be able to learn better in that environment, though this would require far more work (setting up teambuilding functionality, for example). As addressed in the above section, a better reward set that takes into account the current state of the battle may help with overcoming randomness in battle, allowing the agent to learn better in the long run. All of these solutions would require time and effort to implement, but may pay dividends in the long run.

6.2.3 Improving the Training Loop

The training loop as it currently stands requires manual intervention in order to switch between Self-Play and expert. An improvement would be an automated version, running tests to determine when to switch the agent between the two. Currently, the agent is allowed to run for several hours before being switched back and forth, but a better solution may be running an expert battle every 10 or so battles. This would possibly allow the agent to adapt the expert's behaviour into Self-Play. Additionally, there is a gulf between the difficulty of an expert battle and a Self-Play battle. A better training loop would have a gentler slope into the difficulty of an expert battle, requiring expert systems that are not as intelligent as Foul Play. Currently, the agent is thrown into the deep end whenever it is switched to expert battles, 'traumatizing' it in a way.

6.2.4 More Efficient Hardware

PoryAI is currently designed to run on regular hardware. A single GPU, a single CPU and a limited amount of RAM. This set-up is not ideal for training an agent, especially given the complexity of the state in Pokémon battles. In the future, the project may be adapted to run on cloud TPUs or other hardware more suited to training agents. Battles could then be run in parallel, making the training process much faster and by extension allowing problems with the rewards scheme and other functionality to be identified quicker. More complexity could also be implemented, such as an ExpectMiniMax search tree to better deal with randomness, utilising the aforementioned internal emulator.

Chapter 7

Appendix

7.1 Github & Screencast

Github Link: `https://github.com/Oisin-Hearne/PoryAI/`
Screencast Link: `https://youtu.be/c534bu7IDqg`

7.2 Reward Scheme Table

Table 7.1: The reward scheme for PoryAI.

Category	Reward	Value	Reasoning
Dealing Receiving Damage	damage	4	Reward the agent for dealing damage, or punish it for taking damage.
	heal	1	Reward the agent for healing its Pokémon, or punish it for allowing the opponent to heal their Pokémon.
	damageThreshold	0.15	A threshold to determine whether or not to reward the agent for damage dealt. If the damage as a % of the Pokémon's HP is below this threshold, no reward is gained. This discourages the agent from repeatedly using ineffective moves and encourages the agent when it is able to effectively "tank" - that is, reducing the damage its Pokémon take.
	healThreshold	0.25	A similar threshold for healing, implemented due to certain effects that heal the user very slightly which may encourage bad habits if allowed to give the full reward.
	progress	4	A bonus reward to damage dealt, awarded when the agent lowers the opponent Pokémon's HP below 50%. This means it is making some progress in the battle.
	ko	12	Fainting the opposing Pokémon is the best measure of how well the agent is performing. The more faints it gets per battle, the better it is doing. Fainting Pokémon is heavily encouraged because of this. The converse of this is halved, as to not traumatise the agent and because occasionally sacrificing a Pokémon to get into a better position is the optimal move.

Continued on next page

Table 7.1: The reward scheme for PoryAI. (Continued)

Strategic Play	status	3	Encourage inflicting a status effect and discourage getting inflicted with one.
	statusBonus	2	Certain status effects are far more valuable, namely Sleep and Toxic. When inflicted, a slight bonus reward is given to reflect their importance and encourage their use. Spreading the Toxic status over multiple of the opponent's Pokémon is a common strategy in competitive play.
	boost	2	Stat boosts make Pokémon more powerful. The agent is encouraged to boost its own stats, and discouraged when said stats are lowered. The reverse is true of the opponent - encourage lowering the opponent's stats, discourage allowing the opponent them.
	field	5	Setting a field effect (positive or negative) gives the agent an advantage in the long term, so it is heavily encouraged.
	weather	2	Similar to field effects, but more easily set and frequent. Can additionally have positive effects for the opponent.
	hazardSet	2	Setting hazards is hugely important to long-term strategy and is rewarded because of it. Hazards can be stacked, which is why the initial reward may seem low, but it can add up quickly.
	hazardClear	3	Clearing hazards that the opponent has set is often more important than the agent setting its own hazards, which is why the reward is made higher.
	effective	5	Using a super effective move nets a large positive reward, and using a move that is resisted by the opponent nets a negative one. The reverse is true for if the opponent uses a super effective move or a resisted one. This is to encourage switching in to a Pokémon that may resist an attack the opponent is about to use, which puts the agent in a better position.
	fail	3	A move can fail under certain circumstances - for example, the agent trying to inflict a status effect on a Pokémon that already has one. This is punished in order to discourage the agent from repeatedly using certain strategic moves due to the initial reward it gets from them. It should make the connection that if a Pokémon already has a status effect, it shouldn't attempt to apply it again.
	attackIncentive	5	An incentive given for using moves, in order to discourage switching. Due to there typically being more options for switching than using attacks, the agent is predisposed to wanting to switch. This discourages repeated switching behaviour.
	switchDecentive	4	A decentive for switching, to the same effect as the above.
	ratioPunishment	4	Switching is further punished if the ratio of switches to moves exceeds 0.5. In other words, the agent should not switch more than it moves.
	movePunishment	4	A punishment for using the move in the same slot repeatedly. This is to discourage certain effects that result in the user being locked into a move, and to discourage the agent from associating the positive effects of a move with the slot it is in.
	moveLeeway	1.8	Leeway given before move punishment starts kicking in. In certain circumstances, using the same move 10 times in a row may be desirable. Once a move slots usage exceeds the average amount of times a move is used multiplied by this leweay, the movePunishment is applied.

Continued on next page

Table 7.1: The reward scheme for PoryAI. (Continued)

Normalisation	usualMax	15	Used for normalising the rewards tallied up for the events of a turn. It is impossible for every reward to be applied in a single turn, so a usual maximum reward was chosen.
Whole-Battle Performance	shortBattle	2	Shorter battles mean that the agent is finding the correct path to victory quicker. This may be disabled in the future, as high-level strategic play can often go on for far longer. This reward is only given if the user wins.
	switchyBattle	5	Punishment is applied to a whole battle in retrospect if the user switched too many times over the course of the battle.
	repeatBattle	5	Similar to above, but for using the same move repeatedly.
	lengthThreshold	100	The amount of turns a battle can go on for before it's no longer considered a short battle.
	actionThreshold	0.5	The action ratio limit for when switchyBattle can be applied. If the agent switched more than it attacked, switchyBattle is applied.
	repeatThreshold	10	A similar threshold to actionThreshold, but for how many times the agent repeated a move beyond the leeway (how many times it was punished for it).
	win	10	Winning a battle is the ultimate goal here, so a battle that resutled in a win is valued over one that resulted in a loss.

Bibliography

[1] Smogon, The Pokémon Showdown Team. Pokémon showdown. `https://play.pokemonshowdown.com/`. Web Application.

[2] jsoncrack. Json crack. `https://jsoncrack.com/`. Web Application.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

[4] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782), 2019.

[5] Smogon, The Pokémon Showdown Team. Pokémon showdown github. `https://github.com/smogon/pokemon-showdown`. GitHub Repository.

[6] Mark E Glickman. A comprehensive guide to chess ratings. *American Chess Journal*, 1995.

[7] Antar. Everything you ever wanted to know about ratings. Smogon Forums, Mar 2013. [Online]. Available at: https://www.smogon.com/forums/threads/everything-you-ever-wanted-to-know-about-ratings.3487422/ (accessed Apr. 23, 2025).

[8] Smogon, The Pokémon Showdown Team. Pokémon showdown github documentation. https://github.com/smogon/pokemon-showdown/blob/master/sim/SIM-PROTOCOL.md. GitHub Repository Documentation, Protocol Section.

[9] pmariglia. Foul play. https://github.com/pmariglia/foul-play. GitHub Repository.

[10] Majid Ghasemi and Dariush Ebrahimi. Introduction to reinforcement learning.

[11] Ruize Zhang, Zelai Xu, Chengdong Ma, Chao Yu, Wei-Wei Tu, Wenhao Tang, Shiyu Huang, Deheng Ye, Wenbo Ding, Yaodong Yang, and Yu Wang. A survey on self-play methods in reinforcement learning.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. 518(7540):529–533.

[13] pmariglia. poke-engine. https://github.com/pmariglia/poke-engine. GitHub Repository.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[15] pmariglia. Yet another pokemon showdown battle-bot. Smogon Forums, Apr 2019. [Online]. Available at: https://www.smogon.com/forums/threads/yet-another-pokemon-showdown-battle-bot.3648979/.

[16] Pikalytics. Gen 9 ou usage rates - pikalytics. https://pikalytics.com/pokedex/gen9ou. Website.