

Winning at Pokémon Random Battles Using Reinforcement Learning

by

Jett Wang

B.S. Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2024

© 2024 Jett Wang. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by:	Jett Wang Department of Electrical Engineering and Computer Science January 26, 2024
Certified by:	Joshua Tenenbaum Department of Brain and Cognitive Sciences, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Winning at Pokémon Random Battles Using Reinforcement Learning

by

Jett Wang

Submitted to the Department of Electrical Engineering and Computer Science
on January 26, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Pokémon battling is a challenging domain for reinforcement learning techniques, due to the massive state space, stochasticity, and partial observability. We demonstrate an agent which employs a Monte Carlo Tree Search informed by a actor-critic network trained using Proximal Policy Optimization with experience collected through self-play. The agent peaked at rank 8 (1693 Elo) on the official Pokémon Showdown `gen4randombattles` ladder, which is the best known performance by any non-human agent for this format. This strong showing lays the foundation for superhuman performance in Pokémon and other complex turn-based games of imperfect information, expanding the viability of methods which have historically been used in perfect-information games.

Thesis supervisor: Joshua Tenenbaum

Department: Department of Brain and Cognitive Sciences

Acknowledgments

First and foremost, thanks to Andy Spielberg for facilitating and supervising the undergraduate research opportunity which led to this thesis project, and for providing constant guidance and support.

Pokémon Showdown users `WhatColorIsThis?`, `arolakiv`, and `Star Thunderbolt` provided expert commentary on the bot’s behavior.

Sean Knight, Elizabeth Lee, Annie Liu, and Laena Tieng helped to proofread this thesis.

Jacqueline Wei prototyped the state embedding code.

Kairo Morton made sure the neural network architecture was reasonable and provided ideas on how to deal with symmetries in the state representation.

Samuel Wang was the primary playtester in the earlier stages of the project, other than Andy and myself.

Joseph Berleant, Yen-Lin Chen, Leila Pirhaji, and Jessica Titensky have mentored and developed me as a researcher throughout the past four years.

Finally, my family and friends formed the extraordinary support system that enabled me to push through a year of solitary and challenging work with very nonlinear progress.

Contents

Title page	1
Abstract	3
Acknowledgments	5
List of Figures	9
List of Tables	11
1 Introduction	13
1.1 Background	13
1.1.1 Generation 4 Random Battles	13
1.1.2 Difficulty	14
1.1.3 Game Tree Size	14
1.1.4 Pokémon Showdown	16
1.2 Related Work	16
1.2.1 Previous Approaches to Pokémon	17
2 Preliminaries	18
2.1 Definitions	18
2.2 PPO	19
2.3 Monte Carlo Tree Search	20
3 Methods	23
3.1 Neural Network Trained via PPO	24
3.1.1 Hardware Resources	24
3.1.2 Validation	24
3.1.3 Action Space Masking	25
3.1.4 Learning Rate Schedule	25
3.1.5 Training Parallelization	25
3.2 MCTS Implementation	26
3.2.1 Opponent Modeling	26
3.2.2 Sampling Unknown Information	26

3.2.3	Performance Engineering	27
4	Results	29
4.1	Neural Network Training	29
4.2	Evaluation vs. Reference Bots	30
4.3	Evaluation on Online Ladder	32
4.4	Expert Comments	32
5	Future Work and Discussion	36
5.1	Neural Network Improvements	36
5.1.1	Language Model Embeddings of Attributes	36
5.1.2	Recurrent Policy	36
5.1.3	"Recursive" Learning	37
5.2	MCTS Improvements	38
5.2.1	Mixed Strategy	38
5.2.2	Perfect Information	38
5.2.3	Smarter Opponent Modeling	38
5.3	Extensions	39
5.3.1	Beyond Generation 4	39
5.3.2	Team Curation	39
5.3.3	Non-Random Formats	39
A	Neural Network Details	40
A.0.1	State Representation	40
A.0.2	Architecture	41
A.0.3	Hyperparameters	41
	References	44

List of Figures

1.1	Pokémon Showdown	16
3.1	Learning Rate Annealing	26
4.1	Validation Winrate	29
4.2	Training Loss	31
4.3	Elo Progression	32
4.4	Game Log: Misplay	33
4.5	Game Log: Good Play	34

List of Tables

4.1	Winrates Versus Reference Bots	30
4.2	Winrates Versus Experts	33
A.1	Detailed State Representation	41
A.2	Detailed Pokémon Representation	42
A.3	PPO Hyperparameters	43

Chapter 1

Introduction

1.1 Background

A Pokémon battle takes place between two players, who each have a team of up to 6 Pokémon. Each Pokémon is defined by its species, type(s), health (HP), stats, an ability, an item, and 4 moves, collectively referred to as the Pokémon's *set*. Moves can have a wide range of effects, including directly dealing damage, inflicting status effects, or improving stats; items and abilities also vary greatly. The goal of a Pokémon battle is to cause all of the opponent's Pokémon to faint (defined as their HP reducing to zero). The game is turn-based, and on most turns both players choose their actions simultaneously. On any given turn, each player has one Pokémon "active" and can either choose one of the active Pokémon's four moves or switch to another unfainted Pokémon.

Before battles take place, it is typical to construct one's own team of Pokémon, a difficult and interesting task in its own right. In this work, however, we focus on the battling aspect and randomly generate the Pokémon on each player's team.

1.1.1 Generation 4 Random Battles

Since its inception, Pokémon has had 9 major generations, each introducing new Pokémon, items, moves, etc. For this work, we focused on the `gen4randombattles` format, where each player is given a team of 6 Pokémon, drawn from a pool of 296 Pokémon available in Gen 4 and equipped with a procedurally generated set (moves, item, and ability). We chose this format for a few reasons:

- Gen 4 has most of the intricacies of Pokémon mechanics.
 - Earlier generations have noticeable flaws that make them unsuitable for balanced competitive play. Gen 1 has no real counters to strong Psychic-type Pokémon; Gen 2 lacks abilities, which are considered a core mechanic and strategy point; and Gen 3 designates attacks as physical/special based on type, which centers

the metagame around very few bulky Pokémon with good typing (Tyranitar, Skarmory, and Blissey).

- Notably, Generation 4 excludes mechanics from later generations such as mega-evolving and Terastallization which essentially double the action space of the game until they are used. However, we do not believe this puts later generations out of reach of the methods described in this work.
- Despite being an older format, `gen4randombattles` still has an active player base, with hundreds of battles taking place every day on the Pokémon Showdown server.

Strategy-wise, `gen4randombattles` tends to favor "stalling" tactics, using a combination of entry hazards, status moves, and healing moves/items to outmaneuver and whittle down the opponent's team. By contrast, later generations are more amenable to "sweeping" tactics, i.e. chipping at the opponent's HP and gathering information in preparation to strengthen one particular Pokémon to KO most or all of the opponents at once. Both metagames intuitively require strategizing over a long time horizon to master.

1.1.2 Difficulty

Researchers working on Pokémon bots have thus far fallen short of superhuman performance achieved in other domains such as chess. This is due to a few factors:

- **Massive state space.** The number of reachable states is enormous, on par with other difficult games tackled by RL. Later we give a lower bound on the game tree size, in support of this.
- **Stochastic environment.** Randomization in the environment necessitates more samples to learn all the possible consequences of taking some action from some state.
- **Partial observability.** The fact that information about the opposing team is hidden from the player makes it difficult to apply standard RL methods naively.

1.1.3 Game Tree Size

One rough way to judge whether a game is suitable for algorithmic methods is to analyze how many possible games can be played, i.e. the size of the *game tree*. In this section, we provide an approximate lower bound on the game tree size for `gen4randombattles`, to justify the difficulty of the problem. Because we only want a lower bound, we make a number of conservative simplifications.

Consider first the number of possible starting states. The number of possible teams can be approximated as follows:

- 296 Pokémon species, of which each team chooses 6. Repeats are not allowed within a

team, but are allowed between teams. This gives

$$\binom{296}{6}^2 \approx 7.88 \cdot 10^{23}$$

- 6 possible sets for each Pokémon species, consisting of its ability, item, and moves (stats are fixed, conditional on the other traits). This figure was obtained empirically by generating 2.6 million teams (15.7 million sets) using Pokémon Showdown’s team generator. The number of sets for a given Pokémon ranges from 1 to 56, distributed close to exponentially with an average around 8, but we conservatively take the median 6, yielding a factor of

$$6^{12} \approx 6.82 \cdot 10^{10}$$

- One Pokémon from each team chosen to be the first sent out, contributing a factor of

$$6 \cdot 6 = 36$$

The number of starting states, then, is the product of these:

$$\binom{296}{6}^2 \cdot 6^{12} \cdot 36 \approx 6.175 \cdot 10^{34}$$

Each game lasts an average of 25 moves, with the absolute minimum being 6. We conservatively assume 15 moves for our lower bound, where on most turns 1 decision is made by each player from 4-9 options, with the number of options decreasing as Pokémon faint on one’s team. Now assume that each player makes 1 decision from 6 options each turn. Once the decisions are locked in, the outcome of the turn is still randomized: each attack has around 10 possibilities for exactly how much damage it does, can typically miss or critically strike, and some have a chance to inflict secondary effects such as poisoning. Once again simplifying greatly, conservatively let the branching factor due to randomization be 100, bringing the total branching factor per turn to $6 \cdot 6 \cdot 100 = 3600$. For comparison, the branching factor is around 35 for chess and 250 for Go [1].

Combining the starting states, branching factor, and game length, we reach an approximate lower bound for the game tree size of `gen4randombattles`:

$$6.175 \cdot 10^{34} \cdot 3600^{15} \approx 1.365 \cdot 10^{88}$$

This is lower than the theoretical game tree size estimates of other well-known games which have been solved to superhuman level by reinforcement learning, with chess around 10^{120} [2] and Go around 10^{360} [3], but still high enough that tabular methods (keeping exhaustive track of each state, as opposed to parameterizing) are out of the question.

Further, the loss in complexity compared to chess and Go is offset by imperfect information and stochasticity. Another example of this effect is found in heads-up no-limit Texas

Hold'em (a well-known variant of poker), which has 10^{161} decision points but was solved to a superhuman level after Go, due to the relative immaturity of methods for imperfect-information games [4].

Finally, note that this estimate is only for the heavily restricted `gen4randombattle` format; an unrestricted form of Pokémon has been estimated to have 10^{358} *starting states* [5], giving a lower bound for the game tree size on the order of 10^{411} .

1.1.4 Pokémon Showdown

Pokémon Showdown is an open-source Pokémon battle simulator with a highly active community of developers and players [6]. Pokémon Showdown also hosts an online server with a "ladder", where players compete and obtain ratings in various battle formats.



Figure 1.1: The web client for Pokémon Showdown, displaying a `gen4randombattle`.

We make extensive use of the Pokémon Showdown simulator in the present work, and evaluate the final agent partly by having it play online, on the official Pokémon Showdown ladder.

1.2 Related Work

Games have long been used as a testbed for search and planning algorithms. After the defeat of world chess champion Garry Kasparov by IBM's Deep Blue supercomputer in 1997 [7], the next great challenge for AI in games was considered to be Go, another board game. When world-class Go player Lee Sedol fell to Deepmind's AlphaGo in 2016 [8], the focus shifted away from perfect-information, turn-based games.

No-limit Texas hold'em poker was toppled in 2019 [9]; DotA 2 in 2018 [10]; and Stratego in 2022 [11]. These games feature immense state spaces, further complicated by imperfect information and stochasticity, and in some cases multi-agent dynamics, bluffing, and continuous timesteps.

The methods used to create AI agents for these games varied depending on the unique challenges of each game. Turn-based games with perfect information have historically been solved by some variant of search with a parameterized state space and heuristic evaluation function. Even today, the best chess engine, Stockfish, uses an advanced alpha-beta search augmented by a neural network state evaluator [12]. In this work, we investigate if a similar approach can be applied to Pokémon, another turn-based game with the added challenge of imperfect information.

1.2.1 Previous Approaches to Pokémon

Many researchers and hobbyists have created AI agents to play Pokémon.

The strongest previous randombattles agent peaked at rank 33 in `gen7randombattles` [13], using two-turn lookahead as well as a handcrafted heuristic function to evaluate the probability of winning from terminal states. The Future Sight AI [5], which went viral on YouTube, used a similar approach to play a variety of formats, reaching the top 6.8% of players but never reaching the top 500 (where Pokémon Showdown starts publishing concrete rankings) in `gen8ou`.

Huang and Lee [14] trained a neural network using PPO with self-play, achieving a Glicko-1 rating of 1677.

Our work can be seen as combining the two approaches: first training the neural network using PPO with self-play, then using the neural network to guide the "lookahead", i.e. Monte Carlo Tree Search.

Chapter 2

Preliminaries

This section is not meant to provide a complete understanding of the methods used. Rather, it is intended as a high-level conceptual overview for those unfamiliar with reinforcement learning, and a refresher for those already familiar.

2.1 Definitions

In a typical reinforcement learning setup, we have an agent interacting with its environment, trying to maximize its *return*, or the sum of rewards gathered during an episode. Each episode starts in the *start state* s_0 and ends upon reaching a *terminal state* s_T . We lay out relatively standard notation for a finite-horizon discounted Markov Decision Process (MDP):

- $s_t \in \mathcal{S}$, the state at time t .
- $a_t \in \mathcal{A}$, the action taken at time t .
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S}' \rightarrow \mathbb{R}$, the transition function, such that

$$P(s, a, s') = \Pr[s_{t+1} = s' \mid s_t = s, a_t = a]$$

is the probability of reaching state s' after taking action a from state s .

- $r_t : \mathcal{S} \rightarrow \mathbb{R}$, the probability of receiving some reward as a function of the state at time t . Note that reward r_t (not r_{t+1}) is a consequence of the previous turn s_t, a_t .
- γ , the discount factor, which can be thought of as how much we care about the future compared to the present.
- π , a policy such that $\pi(a|s)$ is the probability of taking action a from state s .
- $G_t = \sum_{i=t}^T \gamma^{i-t} r_i$, the discounted return from time t , where T is the length of the episode.

We define $0^0 = 1$, so that if $\gamma = 0$ we care only about the reward at the current time; and if $\gamma = 1$, we care equally about all rewards until the end of the episode.

- $Q_\pi(s_t, a_t) = \mathbb{E}_\pi [G_t | s_t, a_t]$, the expected return from taking action a_t at state s_t and then following policy π thereafter.
- $V_\pi(s_t) = \mathbb{E}_\pi [G_t | s_t]$, the expected return from following policy π starting from state s_t
- $A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$ the *advantage* gained by deterministically taking action a_t rather than following policy π at state s_t .

2.2 PPO: A Policy Gradient Method

Policy gradient methods optimize a parameterized policy π_θ directly through gradient descent¹. The policy, once optimized, does not need a separate value function to select actions (though often a value function is learned to help optimize the policy). A classic issue in policy gradient algorithms is that multiple steps of gradient descent involving the same rollout data leads to massive update sizes [15].

Proximal Policy Optimization (PPO) is a family of policy gradient algorithms first described by OpenAI [15], widely adopted in RL problems for its relative simplicity², stability during training, and superior performance. The key innovation of PPO is a simple clipping of the loss function which controls the size of updates, inspired by earlier "trust-region" methods [18].

Generalized Advantage Estimate (GAE)

PPO estimates the advantage A_π using GAE [19], with the following form:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$

$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t).$$

For this estimate, γ is our discount factor as usual, and λ represents a tradeoff between bias ($\lambda = 0$) and high variance ($\lambda = 1$).

Loss Function

As previously stated, PPO addresses the problem of exploding gradients when applying multiple steps of optimization to the same rollout data. This is done by "clipping" the size of gradient updates so that the policy π_θ may not stray too far from its original version, $\pi_{\theta_{\text{old}}}$.

Let $r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)}$ denote the ratio between the probabilities of choosing action a_t at state s_t using the current policy vs. the "old" policy, i.e. the policy before any steps of optimization. The form of PPO we use combines three loss functions with the following purposes:

¹The gradients are estimated gradients of expectations through sampling; they are not analytical.

²Though PPO is conceptually simple, in practice it requires a slew of code-level tricks and optimizations to achieve the performance purported by OpenAI [16][17].

- The *negative* clipped policy loss:

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

where the clip function is defined as $\text{clip}(a, b, c) = \min(c, \max(b, a))$.

- The value loss, a squared loss term used to learn a value function which forms the baseline for the advantage estimate:

$$L_t^{VF}(\theta) = (V_\theta(s_t) - G_t)^2$$

Sometimes, we elect to also clip the change in the value function:

$$L_t^{VFC}(\theta) = (V_{\theta_{\text{old}}}(s_t) + \text{clip}(V_\theta(s_t) - V_{\theta_{\text{old}}}(s_t), 1 - \epsilon_V, 1 + \epsilon_V) - G_t)^2$$

- A loss based on the *negative* entropy of the policy, to encourage non-deterministic policies and therefore exploration during training:

$$S[\pi_\theta](s) = - \left(- \sum_a \pi_\theta(a|s) \log \pi_\theta(a|s) \right) \quad (2.1)$$

$$= \sum_a \pi_\theta(a|s) \log \pi_\theta(a|s) \quad (2.2)$$

Summing with weights $c_1, c_2 \in [0, 1]$ as hyperparameters, we optimize the following loss:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t [L_t^{CLIP}(\theta) + c_1 L_t^{VFC}(\theta) + c_2 S[\pi_\theta](s_t)]$$

In our implementation, the policy and value functions learned share most of their parameters; we train an actor-critic network, meaning a neural network with one head to estimate the value of the current state (the critic), and another head to output a distribution over actions (the actor).

2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a planning method which searches for the most promising move by building a game tree, sampling state-action values for states within the tree using *rollouts*, or possible futures of the episode.

Vanilla MCTS plays each rollout to the end of the episode (a terminal node), collecting the complete return before backing up to update nodes in the tree. Here we discuss a variant where rollouts may end on either:

- a terminal node, as in vanilla MCTS
- a *leaf node*, i.e. one which is not yet recorded in the tree

Stopping at leaf nodes is made possible because we have V_θ , a trained state value estimator, and provides efficiency gains because rollouts can end earlier.

Search Procedure

Each time the agent needs to make a decision, we launch a MCTS with root node at the current state, s_0 . The search consists of R rollouts, which start from the current state and continue until a leaf or terminal state. R is not a fixed number, but depends on our computational budget (see [subsection 3.2.3](#)). Throughout the game, we keep track of the following statistics as dictionaries:

- $Q[s, a]$, the estimated value of taking action a from state s .
- $N[s, a]$, the number of times we've taken action a from state s .
- $M[s]$, the number of times we've visited state s .
- $P[s, a] = \pi_\theta(a | s)$, the chance that the neural network policy chooses action a from state s .

Tree Policy

At timestep $t < T$ during a rollout, actions are selected according to the following *tree policy*:

$$a_t = \arg \max_a (Q[s_t, a] + \alpha \cdot U(s_t, a))$$

where $U(s, a)$ is an "upper confidence bound"-like term [20] to encourage exploration while using the neural network policy π_θ as a prior:

$$U(s, a) = P[s, a]^\beta \cdot \frac{\sqrt{M[s]}}{N[s, a] + 1}$$

and hyperparameters $\alpha, \beta \in [0, 1]$ dictate how much to value exploration and how much to trust the neural network policy, respectively³.

Backup Update

A rollout ends when we encounter a terminal node or a leaf node at timestep T . At this point we obtain a value v for the final state s_T . If s_T is terminal, v is +1/-1/0 for a win/loss/tie respectively. Otherwise, we use $v = V_\theta(s_T)$, the output of the neural network's critic head, to estimate the value of s_T .

If the rollout ends in a leaf node, we add the node to the tree. Then, for each (state, action) pair encountered during the rollout, we apply following update rules:

- $Q[s_t, a_t] = \frac{N[s_t, a_t] \cdot Q[s_t, a_t] + v}{N[s_t, a_t] + 1}$
- $N[s_t, a_t] = N[s_t, a_t] + 1$

³The tree policy used is similar to that used in AlphaZero [21], except that α is a constant instead of a function of $M[s]$ and β is introduced.

- $M[s_t] = M[s_t] + 1$

After all rollouts are complete, we choose the action with the greatest visit count from the root node:

$$a^* = \max_{a \in \mathcal{A}} N(s_0, a)$$

It is intuitive to choose the action with the largest Q value instead, but less-visited actions may have higher variance in their Q estimates.

Chapter 3

Methods

Inspired by AlphaZero [21], we employ a Monte Carlo Tree Search which is guided by an actor-critic neural network trained through self-play. MCTS deals naturally with the stochasticity of Pokémon by taking an expectation over rollouts. Self-play ensures that no hand-crafted Pokémon-specific heuristics are used to produce our neural network’s value function and policy.

However, our approach diverges from that of AlphaZero in that MCTS is not used to train the neural network. Instead, the neural network is trained via PPO, then MCTS is used purely at inference time as a policy improvement operator. This was done because simulating the environment is very slow, compared to a game like chess; generating gameplay using MCTS would not likely lead to enough samples for a neural network to converge, given the computational constraints of the present work.

We model Pokémon `gen4randombattles` as a POMDP (Partially Observable Markov Decision Process), with the following definitions:

- $s \in \mathcal{S} \subset [0, 1]^{3725}$: Details on the state representation may be found in [Appendix A](#).
- $a \in \mathcal{A} = \{0, 1, \dots, 494\}$: The first 199 actions correspond to moves, while the latter 295 actions correspond to switching to another Pokémon. On any given turn, up to 9 actions are valid (up to 4 moves and up to 5 possible switches); the rest get masked out (see [subsection 3.1.3](#)).
- $r \in \{-1, 0, 1\}$: The reward is 1 on a turn where the agent wins (causes all opposing Pokémon to faint), -1 on a loss, and 0 on any other turn (including ties).

Naively, Pokémon battles break the Markov assumption, meaning that the expected outcome of a turn does not depend strictly on the state and action of that turn. For example, some moves like Light Screen and Rain Dance create conditions in the battle which last multiple turns. We account for this by encoding the durations of multi-turn effects into the state. For example, since Light Screen lasts a maximum of 8 turns, we associate with it a one-hot vector of size 10 in the state (each dimension represents a number of turns between

0 and 8 inclusive, plus an extra dimension for No Light Screen). With this modification, we restore the Markov assumption.

Time Control

In `gen4randombattles`, a timer is kept for each player starting at 150 seconds, which gets replenished by 10 seconds for every decision made. If a player’s timer reaches 0, they instantly lose the game. Thus, the agent was allowed 10 seconds of thinking time per move, which constrained the number of possible futures that could be explored during MCTS.

3.1 Neural Network Trained via PPO

The neural network was trained for 150M steps (approximately 3 million battles, since the average battle lasted 25 steps and experience was collected from the perspective of both players) over 4 days via PPO, with rollout trajectories collected via self-play.

In each game played during training, both players used the most recent iteration of the policy, and both players recorded the trajectory for learning. In other words, each game played produced two games for the algorithm to learn from.

More details on state representation, architecture, and hyperparameters are provided in [Appendix A](#). In the rest of this section, we describe key design choices for the algorithm.

3.1.1 Hardware Resources

The neural network was trained using a single NVIDIA A6000 48G GPU and 80 CPU workers with at most 1G of RAM used by each.

By contrast, AlphaZero uses 4 TPUs to play, and used over 5000 TPUs during training [21]; the system used by OpenAI to defeat human champions at DotA 2 used thousands of GPUs over the course of months [10].

Our hardware setup is comparable to previous work in Pokémon AI, the most resource-intensive of which used "4 AWS c5a.24xlarge instances connected to each other using 10 Gbps Ethernet, each featuring a single 48-core AMD EPYC 7R32 processor and 192 GB of main memory" [13].

3.1.2 Validation

During training, the neural network was validated every 20,000 steps. The sole validation metric was the agent’s winrate over 200 games against `SimpleHeuristicsPlayer` [22], an open-source bot which takes into account hazards and setup moves, while also switching out of bad matchups, equivalent in skill to a beginner Pokémon player. This proved to be a useful metric, especially early on in experimentation, since `SimpleHeuristicsPlayer` is weak enough that any amount of useful learning is reflected in winrate against it, but strong

enough not to get completely dominated throughout the training process (the winrate never surpassed 90%).

3.1.3 Action Space Masking

Drawing from previous work [16][23], we mask out all invalid actions by setting their corresponding output logits to `-float("inf")` before the final softmax layer.

We use the action masks while collecting trajectories, then save them to be applied once again during gradient updates, in particular when calculating the probability that the updated policy would have chosen the saved actions.

3.1.4 Learning Rate Schedule

The learning rate was annealed (lowered) smoothly, according to the formula

$$\ell(x) = \frac{10^{-4.23}}{(8x + 1)^{1.5}}$$

where x denotes the training progress as a floating point number from 0 to 1.

Learning rate annealing had a massive impact on performance of the neural network. With a constant learning rate, the validation winrate was stuck at around 55%, compared to 80% after annealing the learning rate.

The learning rate decay constants 8 and 1.5 were chosen after a few manual runs, rather than tuned with other hyperparameters; future work might more properly tune them, given the empirical importance of learning rate annealing to the neural network’s strength.

The learning rate is depicted as a function of training steps in [Figure 3.1](#).

3.1.5 Training Parallelization

Training proceeded with 39 games (78 workers) being played in parallel. A separate rollout buffer was kept for each worker, and environments were not required to act in lockstep with each other. Once at least half of the rollout buffers were full, returns and advantages were computed, followed by 7 epochs of gradient descent over the batched data.

Environments were not run in lockstep with each other for two reasons:

- First, there can be significant variance in the speed at which environments step. Requiring all environments to return a state before sending the next action would make each step only as fast as the slowest environment. As we scaled up, we found this to significantly dampen the speedup provided by parallelism.
- Second, collecting rollout data from both players in a battle is much more difficult if running all workers in lockstep. This is because turns are not necessarily taken in alternating fashion: sometimes they are simultaneous, or one player makes two

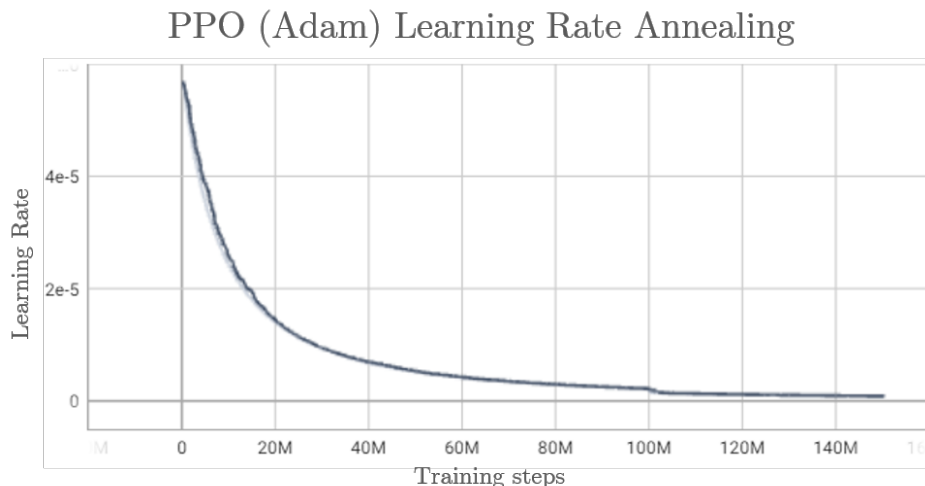


Figure 3.1: A mistake was made while resuming training at 100M steps which caused the learning rate to suddenly dip, then continue slowly decreasing. We do not believe this had any noticeable impact on the final agent’s strength.

decisions in a row while the other waits. Thus, treating two opposing players as independent environments in lockstep inevitably led to a race condition.

3.2 MCTS Implementation

This section describes important deviations from vanilla MCTS and implementation details which allowed us to apply this algorithm to Pokémon battles.

3.2.1 Opponent Modeling

During MCTS, we model the opponent’s decisions using the trained neural network policy. This has the benefit of simplicity, but weakens the agent’s performance against players who play differently from the neural network. In [subsection 5.2.3](#), we discuss a potential alternate approach which may address this weakness.

3.2.2 Sampling Unknown Information

At most points in the game, information about the opponent’s team is largely hidden from the player, making it difficult to naively simulate rollout games. We address this by sampling one possibility for all unknown opponent information at the start of each MCTS trajectory. This is possible because we have access to the exact procedure by which Pokémon Showdown generates `randombattles` teams.

For unknown Pokémon, the server generates a new Pokémon and its set, and adds it to the opponent team. For known Pokémon, the server attempts to generate a valid set

consistent with the known traits of the Pokémon. It does this through *rejection sampling*: generating random sets until one satisfies the constraints formed by known traits.

It can be difficult to sample a valid set, because some sets are modified based on traits of the Pokémon's team, and so could not ordinarily be generated without knowing the opponent's full team. If after 10 attempts we do not generate any valid sets, we "force" the known information to be in the set, randomly filling in unknown information with no regard for compatibility.

3.2.3 Performance Engineering

Speed is of great importance in the search process because of the 10 second time constraint in Random Battle formats. The faster we can search, the more rollouts we can attempt, leading to a more informed decision.

By parallelizing search and controlling tree size, we usually achieve a number of rollouts R between 1000 and 2000 within 10 seconds, depending on the length of each rollout and size of the game tree being tracked.

MCTS Parallelization

MCTS is performed with 20 workers, which periodically share information with each other via an aggregator process. After finishing 10 rollouts, a worker sends the "tree" (in practice a tuple of dictionaries) resulting from its search to the master process, which aggregates the results into a master copy. Finally, the master copy is sent back to the worker and MCTS is run for another 10 rollouts, with this process repeating until a predetermined time limit.

The dictionary of neural network prior probabilities P is not sent back and forth between processes. This is because it is computationally very cheap to recompute the neural network probabilities; the speed of each rollout is bottlenecked by the environment stepping, not GPU inference. In addition, it can be expensive to send the probabilities back and forth between processes. All the other dictionaries store simple scalars rather than a length-496 distribution over actions, and so are much cheaper to send than P .

It is possible that given similar initial trees, each worker traverses the tree in a similar way, inflating the visit count of certain actions. This effect is mitigated, however, by the stochasticity of the environment: even if workers select the same action, it is likely that they will receive different outcomes from the environment, improving the state-action estimate and justifying the extra visits. Empirically, we find this not to be a major issue: the most-visited action nearly always has the highest estimated value.

Controlling Tree Size

In addition to the dictionaries Q, N, M, P mentioned in [section 2.3](#), we also track $F[s]$, the total number of fainted Pokémon in state s . As the game progresses, the dictionaries being passed back and forth between processes can grow to tens of thousands of states, slowing down the search significantly. To address this, $F[s]$ is used to control the size of the

dictionaries. During a `gen4randombattle` game, the total number of fainted Pokémon never decreases. This means once the game reaches f_1 fainted Pokémon, we can remove all states s' from the tree where $F[s'] < f_1$. Other than this, no states are removed from the tree until the end of the game, because states could potentially be re-visited in later rollouts.

With this optimization, the number of nodes stored at any given time varies between 2,000 and 15,000.

Chapter 4

Results

4.1 Neural Network Training

Validation

By our validation metric, the neural network made most of its progress within the first 40M steps (1 day) of training, quickly reaching 80% winrate against `SimpleHeuristicsPlayer`. After 150M total steps (4 days) of training, we reach slightly more than that, roughly 85%.

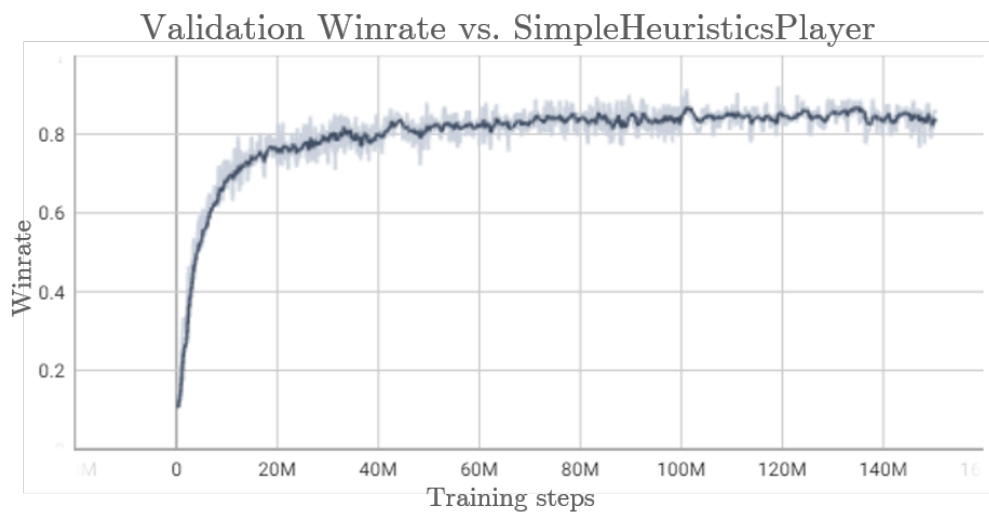


Figure 4.1: Winrate vs `SimpleHeuristicsPlayer` as a function of training steps.

PPO Losses

Recall from [section 2.2](#) that the PPO objective is the sum of a policy loss, value loss, and entropy loss:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t [L_t^{CLIP}(\theta) + c_1 \cdot L_t^{VF}(\theta) + c_2 \cdot S[\pi_\theta](s_t)]$$

where the weights $c_1 = 0.4375, c_2 = 0.0588$ are tuned hyperparameters. Counterintuitively, the overall loss increased throughout training, as depicted in [Figure 4.2](#). This is actually not surprising. The behavior is due to two components of the PPO objective:

- The *entropy loss* increases as training progresses. Recall that the entropy loss is the *negative* entropy of the policy, such that gradient descent will promote a high entropy for the sake of exploration. Despite this, the policy trends towards determinism during training as it becomes more sure about its choices; this corresponds to lowering the entropy, and *increasing the entropy-based loss*.
- The policy loss, which is roughly the *negative* of the clipped estimated advantage, also increases during training. Intuitively, the advantage to be gained over the estimated value function should decrease in magnitude as training progresses. More formally, a perfect state-value function V_{π^*} for an optimal policy π^* should permit no advantage from a deterministic policy: $\max_a \mathbb{E}[G_t(s_t|a, \pi^*)] - V_{\pi^*} \leq 0$. So as the actor-critic network improves, the *magnitude* of the advantage to be gained should decrease. But then the loss should increase, because it has opposite sign from the advantage estimate.

4.2 Evaluation vs. Reference Bots

We evaluated the full agent (MCTS + NN) against:

- NN: The neural network policy playing on its own, with no planning.
- Heuristic: The `SimpleHeuristicsPlayer`, described in [subsection 3.1.2](#).
- Random: A player which chooses valid actions uniformly at random.

The results are in [Table 4.1](#).

Table 4.1: Winrates Versus Reference Bots

	MCTS + NN	NN	Heuristic	Random
MCTS + NN	—	.809	.908	.996
NN	.191	—	.786	1.
Heuristic	.088	.206	—	.992
Random	.004	0.	.007	—

Some head-to-head winrates don't add up to 1, because of ties.

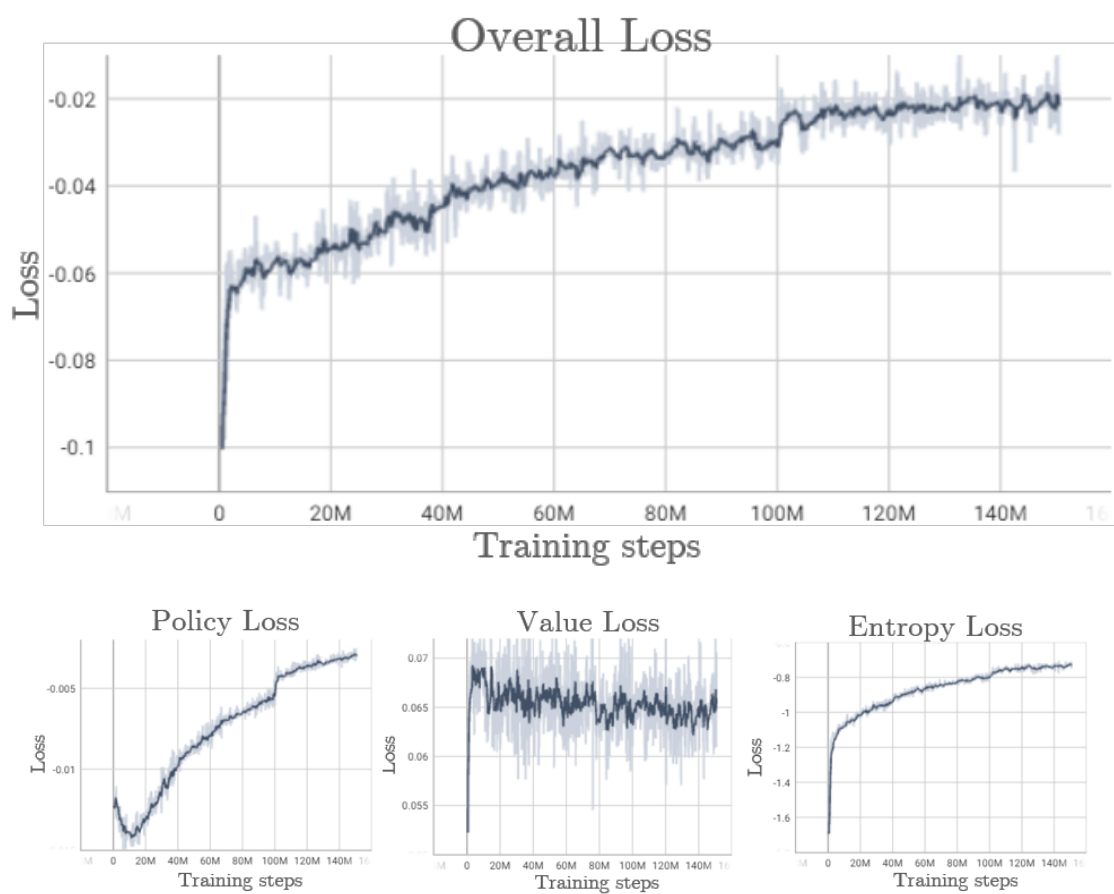


Figure 4.2: Top: overall loss. Left: policy loss. Middle: value loss. Right: entropy loss.

Note that we believe the winrate of the full agent with MCTS vs NN to be somewhat "inflated". Recall that during MCTS, we assume our opponent plays according to the NN policy, and search for the best response. Then, because in essence the MCTS always knows exactly what the NN will do, its winrate when playing against NN is higher than when playing against humans of equivalent strength to NN.

4.3 Evaluation on Online Ladder

Pokémon Showdown displays 3 ratings for a given player in a given battle format: Elo, GXE, and Glicko-1. Elo is used to officially rank players on the ladder, but has high variance; Glicko-1 is considered a more stable estimate of a player's true skill since it takes into account how many games the player has recently played to determine error bars as well as gains/losses. GXE, devised by a Pokémon community member, interprets Glicko-1 to estimate a player's winrate against a randomly chosen player from the ladder [24].

After 200 games played on the Pokémon Showdown ladder with the username `ihftp_abra`, the agent displays an average performance around 1615 Elo¹, peaking at rank 8 (1693 Elo, 1756 ± 28 Glicko-1, 79.5% GXE). This is the best known rank achieved by any non-human agent in `gen4randombattles`.

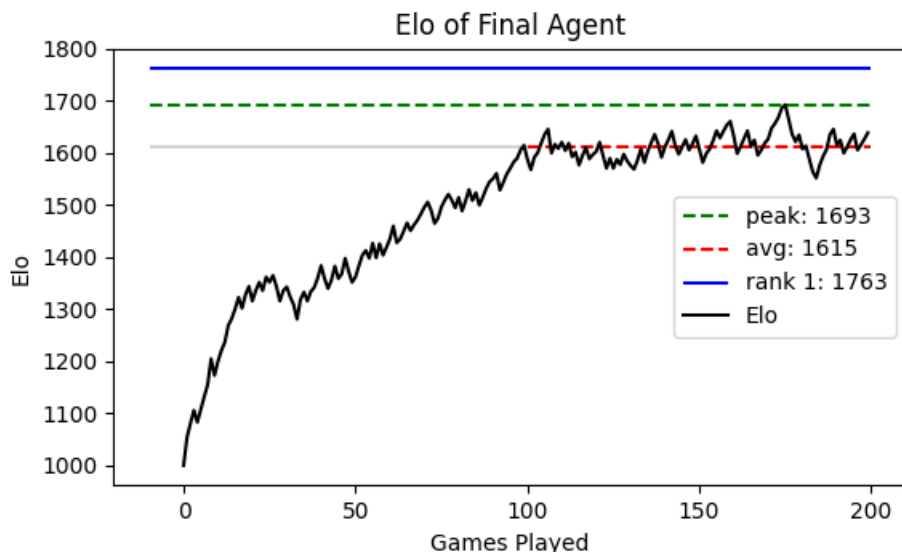


Figure 4.3: The agent's Elo rating progression over 200 games.

4.4 Expert Comments

Other than relative measures like ratings and winrates, it is difficult to quantify the strength of a Pokémon player. To provide a more complete picture, we asked three Pokémon experts to

¹average taken after game 100, to exclude the period where it was climbing

play against the bot and provide insights into its behavior. Table 4.2 displays their win-loss records and credentials.

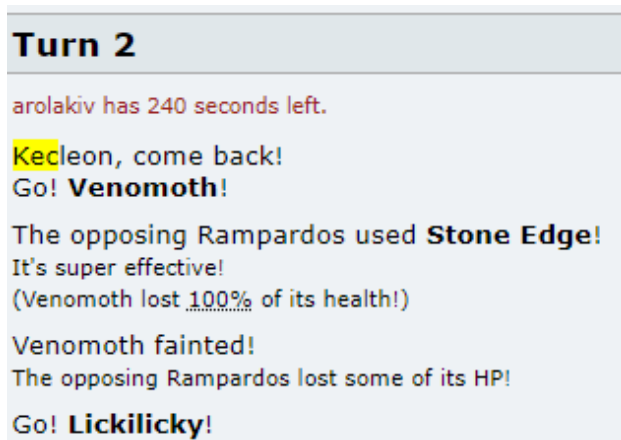
Table 4.2: Winrates Versus Experts

Username	Credentials	W-L vs. bot
WhatColorIsThis?	rank 6, gen4randombattles	2-1
arolakiv	top 50, gen7randombattles	3-1
Star Thunderbolt	rank 1, gen8randombattles	4-11

WhatColorIsThis? noted that the bot was "good in preserving what it needed to win; but when its in a bad position it gets pretty predictable". Star Thunderbolt agreed, saying "one thing that it did really well (which is not surprising for a bot but humans often fail it) was having stuff saved in the back". In other words, the agent seems capable of identifying Pokémon that have significant bearing on its win probability from early on, and playing to preserve those Pokémon throughout the game.

Gameplay Examples

arolakiv noted a few unconventional decisions made by the bot, which seemed to make overly strong assumptions about its opponent’s gameplay. For example, take turn 2 of [their second game](#), where the bot has a Kecleon matched up against the opponent’s Rampardos. Statistically, around 37.4% of Rampardos have the move Superpower, which is supereffective against the Normal-type Kecleon. The bot, fearing this, switched out to Venomoth, which was a poor choice because *all* Rampardos know some powerful Rock-type move, supereffective against the Bug-type Venomoth.



```

Turn 2
arolakiv has 240 seconds left.
Kecleon, come back!
Go! Venomoth!
The opposing Rampardos used Stone Edge!
It's super effective!
(Venomoth lost 100% of its health!)
Venomoth fainted!
The opposing Rampardos lost some of its HP!
Go! Lickilicky!

```

Figure 4.4: Game log of a poor switch made by the bot vs arolakiv.

This suboptimal behavior could be explained by the fact that during MCTS, the bot strongly assumes that the opponent plays according to π_θ , the policy of the trained neural network. It is conceivable that the neural net never chooses a Rock-type attack on that turn

against Kecleon, so the search was not aware of the potential for the switched-in Venomoth to get instantly knocked out. In [subsection 5.2.3](#) we discuss a potential method to better model opponent behavior.

Star Thunderbolt was pleasantly surprised at a couple of the strategies employed by the bot. For example, consider the following strategy from turns 12-13 of [their fourteenth game](#), which we call "bait-and-switch for marginal gain".

At the end of turn 12, the bot's Drifblim faints, so it must send out a Pokémon to replace it in turn 13. Its strongest switch would be to Grumpig, which is particularly strong against the opponent's Hitmontop, but instead the bot opts to switch to Hitmonlee, then *double switch* to Grumpig. This baits **Star Thunderbolt** in turn 13 to use Close Combat, a move which is powerful but lowers the user's defensive stats. Because Close Combat is not very effective against Grumpig, the bot sustained minimal damage in exchange for a significant stat advantage.

```

Turn 12

The opposing Hitmontop used Sucker Punch!
It's super effective!
(Drifblim lost 17.9% of its health!)
Drifblim fainted!
Go! Hitmonlee!

Turn 13

Hitmonlee, come back!
Go! Grumpig!

The opposing Hitmontop used Close Combat!
It's not very effective...
(Grumpig lost 25.8% of its health!)
The opposing Hitmontop's Defense fell!
The opposing Hitmontop's Sp. Def fell!
Grumpig restored a little HP using its Leftovers!

```

Figure 4.5: Game log of the bot's "bait-and-switch" strategy vs **Star Thunderbolt**.

This strategy is not commonly employed by human players because it relies on the opponent "taking the bait". We suspect optimal play uses this tactic some fraction of the time, as part of a mixed strategy.

Overall Opinions

The final question asked to all 3 experts was to estimate their long-term winrate against the bot, percentage-wise, if they each had time to adjust to the `gen4randombattle` format (two of them do not specialize in this format). `WhatColorIsThis?` said "maybe 50 or 60"; `arolakiv` said 50; and **Star Thunderbolt** said "in theory slightly over 50".

Overall, all 3 players thought the bot was competitive but slightly worse than the top human players in its format. **Star Thunderbolt** expressed this viewpoint when they

described the bot as "very good but not elite". This is consistent with its performance on the ladder, where it hovers within the top 100 players worldwide.

Saved replays from most games played against the expert players can be viewed at <https://github.com/quadraticmuffin/pkmmn-thesis-replays> by downloading the HTML files and opening using any browser.

Chapter 5

Future Work and Discussion

5.1 Neural Network Improvements

5.1.1 Language Model Embeddings of Attributes

The current architecture for the neural network makes use of learned embeddings of moves, items, abilities, implemented as PyTorch `nn.Embedding` layers. These embeddings are initialized randomly and updated throughout training. We would be interested to see the impact of fixed embeddings which instead come from, e.g., a large language model (LLM) after passing in a description of the move, item, or ability. We conjecture that if given robust, low-dimensional embeddings rather than having to learn them through experience, the model would become proficient after fewer training iterations. This more closely matches the way humans learn about Pokémon: not just through experience, but also in "zero-shot" fashion, through reading!

This approach, if successful, has the added benefit of flexibility: if a completely new move was made up, one would not need to re-train or fine-tune the model to adapt. Rather, one would need only to describe the move in plain English to an LLM, then add the resulting embedding to the model's Embedding layer.

5.1.2 Recurrent Policy

The neural network is not currently recurrent, meaning it cannot take into account previous states in the game. This is partly accounted for as we provide many relevant details in the state representation, such as the last move used, number of turns spent asleep, and number of turns weather has been active. However, we still believe that a recurrent architecture could benefit the agent, allowing it in particular to exploit patterns in previously-seen behavior of the current opponent.

The literature also suggests empirically that a recurrent policy should provide some benefit for certain classes of problems, in particular where the Markov assumption is broken

as in Pokémon battles [16]. Thus, using a recurrent policy might allow us to reduce the effort and domain knowledge that went into encoding durations of multi-turn effects in the state representation.

5.1.3 "Recursive" Learning

In some sense, Pokémon can be viewed as a recursive game. Imagine that you have trained 6 actor-critic networks $(\pi_1, v_1), (\pi_2, v_2), \dots, (\pi_6, v_6)$, where (π_k, v_k) specializes in Pokémon battles with teams containing k Pokémon. Now suppose that in a 2v2 battle, one Pokémon has fainted from each team, leaving one left on each side. Should you continue using policy π_2 to decide actions, or should you switch to π_1 , which presumably has more in-depth knowledge of 1v1 matchups?

A further question would be whether (π_1, v_1) can actually be used to help *train* (π_2, v_2) in a more sample-efficient way than from scratch. π_1 has already learned to deal with the 1v1 case; why should π_2 learn the same thing again? Perhaps π_2 should instead focus solely on the subtask which lies *outside* the 1v1 domain, consisting of 2v2, 1v2, and 2v1 matchups. Whenever a game devolves into the 1v1 case, say at time t , one can simply use $v_1(t)$ as an estimate of the return from that point on, saving time by truncating each episode and training (π_2, v_2) only on timesteps where some team still has 2 Pokémon. Similarly, π_2 could be used to train π_3 and so on, with the reasoning that more specialized agents will perform better at their individual subtasks.

We attempted such a setup, but saw no significant improvements over simply training a 6v6 from scratch, in either training efficiency or strength of the trained agent. We see two potential issues which might be holding back the idea:

- States which arise after one Pokémon has fainted on each team in a 2v2 may be *out-of-distribution* compared to the states which π_1 trained on. For example, it may be the case that our Pikachu is poisoned even though the opponent's Slaking doesn't know any moves which cause poisoning. This state could arise if the opponent's already-fainted Muk poisoned our Pikachu 4 turns earlier, but π_1 would never have encountered the current state, where a Pikachu is poisoned while facing off against a Slaking. This illustrates why after reducing to the 1v1 case, π_1 might not be the best-performing option.
- It may be that similar reasoning is required at most stages of the game. For example, π_4 might not actually be all that different from π_3 except for the size of their state spaces; strategies involving more than 3 Pokémon are seldom employed. Thus it might be more efficient to, say, use π_1 directly to train π_6 .

We believe that this approach may still be promising, and has some theoretical justification in the literature of so-called "hierarchical" reinforcement learning [25][26].

5.2 MCTS Improvements

5.2.1 Mixed Strategy

Currently, our MCTS procedure ends by picking the action with greatest visit count from the root node s_0 in the tree:

$$a^* = \max_{a \in \mathcal{A}} N(s_0, a)$$

However, in some situations the optimal strategy is to randomize, e.g., choose randomly between two or more actions. The results of our MCTS often imply this, with one action just barely edging out the other. Therefore a better approach might be to randomly choose between actions, weighted by their visit counts (of course pruning actions with very low visit counts or Q values).

5.2.2 Perfect Information

After sampling opponent hidden information, we actually have moved into a regime with perfect information. Currently, the sampled information is only made available to the opponent so that it can play moves, and to the server so that it can properly simulate the rollout. If all info were made available to both players, higher-quality moves could be chosen for each rollout, improving the result of MCTS.

To complete this approach, we would also train a new neural network on a perfect-information version of the game.

5.2.3 Smarter Opponent Modeling

As described in [section 3.2](#), we model the opponent’s decisions using the neural network policy during search, but this leads to problems when facing players who are significantly stronger than (or just play differently from) the neural network. A possible future direction would be to try to also run MCTS from the opponent’s perspective. This is difficult because we don’t know everything about the opponent’s team; the MCTS would somehow need to have many root nodes.

One potential implementation is to alternate between perspectives, running a full MCTS from one player’s perspective, then using the obtained tree policy as an opponent model for the other player. Procedurally, if playing as Player A:

1. Run MCTS from Player A’s perspective, assuming that Player B plays according to the neural net policy π_θ . Obtain tree policy π_A^1 for Player A from MCTS.
2. Run MCTS from Player B’s perspective, assuming that Player A plays according to π_A^1 for states within its tree search and π_θ otherwise. Obtain tree policy π_B^1 for Player B from MCTS.
3. Run MCTS from Player A’s perspective, assuming that Player B plays according to π_B^1 for states within its tree search and π_θ otherwise. Obtain tree policy π_A^2 for Player

A from MCTS.

4. Repeat steps 2 and 3 for desired number of rounds.

We conjecture that such a procedure might eventually arrive at an equilibrium strategy for the current turn, if one exists.

5.3 Extensions

We have confidence that the exhibited methods form a foundation which can be used to extend beyond `gen4randombattles`.

5.3.1 Beyond Generation 4

One exciting avenue would be taking the agent to the currently most-played random format, `gen9randombattles`. There are new mechanics such as Terastallization, which expands the action space, and new Pokémon such as Zoroark, which complicates the state space by disguising itself as another Pokémon on its own team.

5.3.2 Team Curation

Once a strong battling agent is created (in particular, one with a strong state evaluation function), a natural next step is turning towards team curation, e.g. choosing the species, item, ability, and moves for one's Pokémon rather than being given these traits. Previous works have equipped a genetic algorithm with their value function, employing various heuristics to create balanced teams without many exploitable weaknesses [13].

5.3.3 Non-Random Formats

Equipped with strong battling and team creation, we envision superhuman performance in all Pokémon formats, including those where players create their own teams. A robust strategy here would have to involve some team randomization, rather than always playing with the same team (which would be exploitable).

In addition, some changes might have to be made to our search procedure during battle. In `randombattles`, we know with precision the possible sets an opponent Pokémon might have, as well as their relative frequencies. Once players can make their own teams, this guarantee is gone, once again exposing our agent to potential exploitation.

Appendix A

Neural Network Details

A.0.1 State Representation

The input to the neural network is a single vector of length 3725. The features captured in the state are described in [Table A.1](#), with Pokémon-specific features expanded upon in [Table A.2](#).

Multi-turn effects with maximum duration k are generally encoded using a $k + 1$ -length onehot vector, with the extra dimension for when the effect is not present. This encoding preserves the Markov property as discussed in [Methods](#).

Weather

Weather, when summoned by moves, can last 5 or 8 turns (or until overridden by a different weather), depending on the item held by the move’s user. On the other hand, weather caused by abilities (as opposed to moves) is permanent until a different weather is imposed. This permanence only occurs in Generations 3-5. Thus we add an extra onehot bin for each weather condition which signifies "permanent" imposition of that type of weather.

State Binning

Many states in Pokémon are very similar, so we group together certain states, treating them as identical. Firstly, we divide a Pokémon’s HP into 1 special state for 0 HP and 6 equally-sized bins¹, regardless of its total HP, such that having, e.g., 10% and 12% HP are treated as the same state. We also bin a move’s PP, or how many times it can be used, using the formula $\lfloor x^{1/3} \rfloor$ (chosen because resolution matters more when a move has fewer uses left; inspired by Deepmind’s AlphaStar, which plays Starcraft II [\[27\]](#)), which gives a total of four bins since 64 is the highest possible PP.

¹Arbitrarily chosen to significantly cut down on state space. Future work would explore more fine-grained bins, perhaps 16 bins since that is the smallest increment of both healing from the popular item Leftovers and damage from being poisoned by Toxic.

Table A.1: Detailed State Representation

feature	length	domain	notes
sun	9	$\{0, 1\}$	onehot: # turns, or Permanent
rain	9	$\{0, 1\}$	onehot: # turns, or Permanent
hail	9	$\{0, 1\}$	onehot: # turns, or Permanent
sandstorm	9	$\{0, 1\}$	onehot: # turns, or Permanent
no weather	1	$\{0, 1\}$	
trick room	7	$\{0, 1\}$	onehot: # turns, or No Trick Room
force switch	2	$\{0, 1\}$	e.g. used the move U-turn, fainted
# unknown	7	$\{0, 1\}$	
stealth rock	$2 \cdot 2$	$\{0, 1\}$	onehot for each side
spikes	$2 \cdot 4$	$\{0, 1\}$	onehot for each side: # layers, or none
toxic spikes	$2 \cdot 3$	$\{0, 1\}$	onehot for each side: # layers, or none
reflect	$2 \cdot 10$	$\{0, 1\}$	onehot for each side: # turns, or none
light screen	$2 \cdot 10$	$\{0, 1\}$	onehot for each side: # turns, or none
safeguard	$2 \cdot 7$	$\{0, 1\}$	onehot for each side: # turns, or none
Pokémon	$12 \cdot 300$	mixed	see Table A.2

A.0.2 Architecture

The neural network is comprised of the following components:

- A feature extractor, which converts indices (for species, abilities, items, moves) into tensors via learned `nn.Embedding` layers, and projects to a length 896-vector for each Pokémon as well as for the overall battle, for a total output length of $13 \cdot 896$.
- A 3-layer MLP with hidden dimension 256 and ReLU activations.
- An *actor* head, composed of a 2-layer MLP followed by projection to a distribution over the action space.
- A *critic* head, composed of a 2-layer MLP followed by projection to a scalar estimate of the state value.

A.0.3 Hyperparameters

Hyperparameters were tuned via Bayesian optimization on a surrogate task: 3v3 battles where each team is given only 3 Pokémon instead of 6. Because training just one model takes so long for full 6v6 games, we tuned hyperparameters in the 3v3 case, where battles are half as long. This subgame retained some complexity, but vastly shortened the amount of time to see significant differences in hyperparameter performance.

The tuned hyperparameters are displayed in [Table A.3](#).

Table A.2: Detailed Pokémon Representation

feature	length	domain	notes
species	1	$\{0, 1, \dots, 295\}$	
ability	1	$\{0, 1, \dots, 100\}$	
item	1	$\{0, 1, \dots, 39\}$	
move	4	$\{0, 1, \dots, 198\}$	
$\lfloor \sqrt[3]{\text{move PP}} \rfloor / 4$	4	$\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$	see State Binning
last used move	1	$\{0, 1, \dots, 198\}$	
type(s)	18	$\{0, 1\}$	can have 1 or 2 types
current hp fraction	7	$\{0, 1\}$	onehot; see State Binning
accuracy boost	13	$\{0, 1\}$	onehot from -6 to +6
atk boost	13	$\{0, 1\}$	
\dots boost	$5 \cdot 13$	$\{0, 1\}$	def, evasion, spa, spd, spe
volatile effects	$2 \cdot 38$	$\{0, 1\}$	length-2 onehots for OFF or ON
encore	9	$\{0, 1\}$	onehot: # turns or none
taunt	6	$\{0, 1\}$	onehot: # turns or none
magnet rise	7	$\{0, 1\}$	onehot: # turns or none
slow start	6	$\{0, 1\}$	onehot: # turns or none
gender	3	$\{0, 1\}$	onehot: male, female, neutral
status	7	$\{0, 1\}$	onehot: all non-volatile statuses + FNT
toxic counter	21	$\{0, 1\}$	onehot: # turns
sleep counter	11	$\{0, 1\}$	onehot: # turns
$\log_{10}(\text{weight})$	5	$\{0, 1\}$	onehot after rounding
$\log_{10}(\text{height})$	4	$\{0, 1\}$	onehot after rounding
first turn	2	$\{0, 1\}$	
protect counter	6	$\{0, 1\}$	how many Protects in a row, max 5
must recharge	2	$\{0, 1\}$	due to Giga Impact
preparing	2	$\{0, 1\}$	due to moves like Bounce
active	2	$\{0, 1\}$	
is opponent	2	$\{0, 1\}$	
unknown	1	$\{0, 1\}$	If this is 1, all other values are 0.

Table A.3: PPO Hyperparameters

parameter	value	description
learning_rate	$10^{-4.23}$	annealing rate not tuned; see subsection 3.1.4
n_epochs	7	# gradient descent passes over a given batch of data
gamma	0.9999	discount factor
gae_lambda	0.754	trades off between bias ($\lambda = 0$) and variance ($\lambda = 1$)[19].
clip_range	0.0829	smaller value will clip the prob. ratio more aggressively
clip_range_vf	0.0184	
entropy_coef	0.0588	weight of entropy term in loss
value_coef	0.4375	weight of value function loss
max_grad_norm	0.5430	limit to norm of gradient update
n_steps	$78 \cdot 512$	# steps experience to collect to train; 78 is # workers
batch_size	1024	
features_dim	896	
hidden_dim	256	

References

- [1] J. Burmeister and J. Wiles, “The challenge of go as a domain for ai research: A comparison between go and chess,” in *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, 1995, pp. 181–186. DOI: [10.1109/ANZIIS.1995.705737](https://doi.org/10.1109/ANZIIS.1995.705737).
- [2] C. E. Shannon, “Programming a computer playing chess,” *Philosophical Magazine*, vol. Ser.7, 41, no. 312, 1959.
- [3] L. Allis, “Searching for solutions in games and artificial intelligence,” English, Ph.D. dissertation, Maastricht University, Jan. 1994, ISBN: 9090074880. DOI: [10.26481/dis.19940923la](https://doi.org/10.26481/dis.19940923la).
- [4] N. Brown and T. Sandholm, “Superhuman ai for heads-up no-limit poker: Libratus beats top professionals,” *Science*, vol. 359, no. 6374, pp. 418–424, 2018. DOI: [10.1126/science.aao1733](https://doi.org/10.1126/science.aao1733). eprint: <https://www.science.org/doi/pdf/10.1126/science.aao1733>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aao1733>.
- [5] A. D. III, *Future sight ai*, 2021. [Online]. Available: <https://www.pokemonbattlepredictor.com/FSAI>.
- [6] G. Luo, A. Werner, A. L., C. Monsanto, K. Johnson, L. C. III, M. Dias-Martins, and M. A, *Pokemon showdown*, <https://github.com/smogon/pokemon-showdown>, 2023.
- [7] M. Campbell, A. Hoane, and F.-h. Hsu, “Deep blue,” *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002, ISSN: 0004-3702. DOI: [https://doi.org/10.1016/S0004-3702\(01\)00129-1](https://doi.org/10.1016/S0004-3702(01)00129-1). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0004370201001291>.
- [8] D. Silver, A. Huang, C. J. Maddison, *et al.*, “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, 2016, ISSN: 0028-0836. DOI: [10.1038/nature16961](https://doi.org/10.1038/nature16961).
- [9] N. Brown and T. Sandholm, “Superhuman ai for multiplayer poker,” *Science*, vol. 365, no. 6456, pp. 885–890, 2019. DOI: [10.1126/science.aay2400](https://doi.org/10.1126/science.aay2400). eprint: <https://www.science.org/doi/pdf/10.1126/science.aay2400>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aay2400>.
- [10] OpenAI, : C. Berner, *et al.*, *Dota 2 with large scale deep reinforcement learning*, 2019. arXiv: [1912.06680](https://arxiv.org/abs/1912.06680) [[cs.LG](#)].

- [11] J. Perolat, B. De Vylder, D. Hennes, *et al.*, “Mastering the game of stratego with model-free multiagent reinforcement learning,” *Science*, vol. 378, no. 6623, pp. 990–996, Dec. 2022, ISSN: 1095-9203. DOI: [10.1126/science.add4679](https://doi.org/10.1126/science.add4679). [Online]. Available: <http://dx.doi.org/10.1126/science.add4679>.
- [12] T. S. developers, *Stockfish*, 2023. [Online]. Available: <https://github.com/official-stockfish/Stockfish>.
- [13] N. R. Sarantinos, *Teamwork under extreme uncertainty: Ai for pokemon ranks 33rd in the world*, 2023. arXiv: [2212.13338](https://arxiv.org/abs/2212.13338) [cs.AI].
- [14] D. Huang and S. Lee, “A self-play policy optimization approach to battling pokémon,” in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–4. DOI: [10.1109/CIG.2019.8848014](https://doi.org/10.1109/CIG.2019.8848014).
- [15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: [1707.06347](https://arxiv.org/abs/1707.06347) [cs.LG].
- [16] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, “The 37 implementation details of proximal policy optimization,” in *ICLR Blog Track*, <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>, 2022. [Online]. Available: <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>.
- [17] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, *Implementation matters in deep policy gradients: A case study on ppo and trpo*, 2020. arXiv: [2005.12729](https://arxiv.org/abs/2005.12729) [cs.LG].
- [18] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, 2017. arXiv: [1502.05477](https://arxiv.org/abs/1502.05477) [cs.LG].
- [19] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, *High-dimensional continuous control using generalized advantage estimation*, 2018. arXiv: [1506.02438](https://arxiv.org/abs/1506.02438) [cs.LG].
- [20] C. D. Rosin, “Multi-armed bandits with episode context,” *Ann. Math. Artif. Intell.*, vol. 61, no. 3, pp. 203–230, 2011. [Online]. Available: <http://dblp.uni-trier.de/db/journals/amai/amai61.html#Rosin11>.
- [21] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: [10.1126/science.aar6404](https://doi.org/10.1126/science.aar6404). eprint: <https://www.science.org/doi/pdf/10.1126/science.aar6404>. [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aar6404>.
- [22] H. Sahovic, *Poke-env: Pokemon ai in python*, 2023. [Online]. Available: <https://github.com/hsahovic/poke-env>.
- [23] S. Huang and S. Ontañón, “A closer look at invalid action masking in policy gradient algorithms,” *The International FLAIRS Conference Proceedings*, vol. 35, May 2022, ISSN: 2334-0762. DOI: [10.32473/flairs.v35i.130584](https://doi.org/10.32473/flairs.v35i.130584). [Online]. Available: <http://dx.doi.org/10.32473/flairs.v35i.130584>.

- [24] X-Act, *Gxe (glixare): A much better way of estimating a player's overall rating than shoddy's cre*, 2009. [Online]. Available: <https://www.smogon.com/forums/threads/gxe-glixare-a-much-better-way-of-estimating-a-players-overall-rating-than-shoddys-cre.51169/>.
- [25] R. Makar, S. Mahadevan, and M. Ghavamzadeh, "Hierarchical multi-agent reinforcement learning," in *Proceedings of the Fifth International Conference on Autonomous Agents*, ser. AGENTS '01, Montreal, Quebec, Canada: Association for Computing Machinery, 2001, pp. 246–253, ISBN: 158113326X. DOI: [10.1145/375735.376302](https://doi.org/10.1145/375735.376302). [Online]. Available: <https://doi.org/10.1145/375735.376302>.
- [26] E. M. Hahn, M. Perez, S. Schewe, F. Somenzi, A. Trivedi, and D. Wojtczak, "Recursive reinforcement learning," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 35 519–35 532. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/e6f8759254d86ea9c197d30b92b313ca-Paper-Conference.pdf.
- [27] M. Mathieu, S. Ozair, S. Srinivasan, *et al.*, *Alphastar unplugged: Large-scale offline reinforcement learning*, 2023. arXiv: [2308.03526](https://arxiv.org/abs/2308.03526) [cs.LG].