# A Performant Game Engine and Monte-Carlo Tree Search Agent for Pokemon Battles

Derek Dong, James Glenn (advisor)

April 29, 2025

**Abstract**

With recent work developing superhuman competitors in games like Starcraft II and Quake III Capture the Flag (by DeepMind), and Dota 2 (by OpenAI), the development of agents to play video games is clearly an exciting frontier of machine learning research. Meanwhile, Pokemon is one of the most popular games that surprisingly has yet to truly be explored by the machine learning community. The many game mechanics like types, abilities, moves, and items interact in complex ways, but the overall strongly causal design of the game makes computational solutions especially appealing. So what would it take to develop an agent to win Pokemon battles? An important barrier is the limited quantity of game history data, linked to the relatively slow speed of game simulators. We build an engine using C++ that simulates Pokemon battles two to four orders of magnitude faster than Pokemon Showdown, the existing state-of-the-art. We demonstrate the engine's performance by comparing the time it takes to run turn loops against the time that Showdown takes, and we provide an example of a Monte Carlo Tree Search agent that leverages the engine to quickly play through rollouts and search for the best move at a given state of battle. Our results are from standardized examples from Pokemon Run and Bun, chosen for its recent surge in popularity as well as its shockingly difficult battles. Our agent battles against teams handcrafted by competitive players and one of the smartest rule-based Pokemon opponents in Pokemon—and wins. Our engine compiles to a shared object file that can be imported into a Python environment, and our source code is available on GitHub.

## 1 Introduction

Pokemon is one of, if not the, most popular and highest-grossing media franchises in the world [pkm]. Its foundation lies in role-playing games centered around a complex "turn-based" battle system where players pit the team of Pokemon they've caught against opposing teams. Players can battle against each other or against rule-based opponents built into the games. The largest Pokemon tournament each year has a prize pool of over $2,000,000, and YouTube videos detailing playthroughs of the base games or their "ROM hacks" can each garner millions of views, which goes to show how much time, energy, and attention is invested into Pokemon.

Developing agents to play video games is also of particular interest in the ML community. Agents have been successfully trained to excel and even surpass human performance in games ranging from reasonably complex combinatorial games like chess to higher-branching dynamic, stochastic, and/or multi-agent games like Dota, Starcraft, and Quake [RP20]. Games have controlled, well-defined environments and, simultaneously, sufficient complexity where techniques can be demonstrated or refined and then transferred to more practical applications of high-agency AI, like robotics or large language models.

Combining the two, the task of developing an AI skilled at Pokemon battling is a research problem of interest to both Pokemon's vast playerbase and academic researchers. Many modern approaches to developing video game agents involve state representations and vast repositories of game histories to indirectly encode mechanics. That is, agents tend to be trained to learn how the game works by being shown lots of examples instead of being inherently coded with the rules. There are limitations to recording match histories between human players such as: battle animation runtime, competitive format restrictions/regulations, and lack of infrastructure in all but the biggest tournaments. Thus, generating the data via simulated battles is crucial.

The goal of the project is to provide *efficient* Pokemon battle simulation in order to more cheaply generate ample game history data and enable training powerful battling agents. In this paper, we use the terms "performance" and "efficiency" with respect to the objective, i.e. speed. We aim to provide ML researchers with a well-understood environment containing a high stochastic branching factor, and we aim to democratize access to high-quality Pokemon battles (e.g. between two agents or between a player and an agent) and bring more serious Pokemon battling to the public eye.

## 2 Background

### 2.1 Pokemon battles

The basic process of a Pokemon battle consists of two players ("trainers") with "teams" of up to 6 Pokemon each. Each Pokemon has its own stats (offensive and defensive), moves, and health point (HP) values. At any point in the match, each player has one Pokemon active (in a "singles battle"), and they simultaneously can choose in a turn to attack with a moves or to switch to one of up to 5 other Pokemon, for a branching factor of about 9. Once a Pokemon's HP reaches 0, they "faint," and when a trainer's Pokemon all faint, they lose.

The computational difficulty of battling comes from the complex interactions and massive state space. There are over 1,000 distinct Pokemon, not including forme changes. Each Pokemon has up to two of 18 types, has one of over 200 Abilities, might be holding one of over 300 Held Items (given to it before the battle by its trainer), and has access to up to 4 of over 700 moves (taught to it before the battle by its trainer). There are further restrictions on which Ability or moves a species of Pokemon can have. Stat-changes, multi-hit moves, move priority, and Statuses (e.g. Sleep or Paralysis) expand the state space, as do combinations of environmental conditions like "weather," "terrain," and "hazards." An agent must understand all of these conditions/effects and how they interact – both with the rest of their team and the opposing team's – to win the game.

In the core Pokemon games, a player can battle against either a rule-based opponent (hard-coded into the game) or against another player (online). So there are three types of participants: human players, artificial agents (the development of which is the goal of the field), and the rule-based opponent. We distinguish the latter two as "AI" and "opponent," respectively, to avoid confusion.

### 2.2 Related work

Pokemon battles, in their original form, are in a video game and meant to be played by a human. This, combined with the fact that the games have strict anti-piracy measures, means that they are inseparable from the GUI and animations. This is rarely a problem for human players, but it's a severe performance and interface bottleneck for agent development. An AI can only interface with the core games by simulating clicks in the original console and parsing/processing entire still frames to understand the game state. Most machine learning techniques require significant amounts of data, which requires the speed for a large number of playthroughs. Thus, battle simulators are a crucial tool for research in the field.

One popular existing framework to simulate Pokemon battles is Pokemon Showdown [sho, Sah,LT17]. The implementation allows players to easily create their own teams (instead of catching/training them from scratch in the base games) and battle each other in their browsers. It also provides a protocol for programs to send input, partially circumventing the GUI/animation requirements. However, it's built for generality and security. Playing against an opponent that chooses moves that are strong but impossible for their Pokemon to have (recall that such pairs might be restricted, especially in certain generations of the core games or in certain competitive formats) wouldn't be fun to play against, and so Pokemon Showdown validates each input. Showdown also supports features like a timer (to prevent players from refusing to choose a move) and move changes (so long as the other player hasn't committed their choice yet) for UX that present overhead in both the underlying algorithm and the design choices of the system itself—Showdown is implemented in TypeScript for its dynamism.

Additionally, interesting work has been done in training agents themselves. The two most common approaches are MCTS [Lin19] and reinforcement learning [HL19, SRLR20], and the most impressive work (from 2024) uses a neural network together with MCTS similarly to AlphaGo to achieve a peak rank of top-8 in the Pokemon Showdown ladder for its format [Wan24]. These works tend to overcome performance issues by aggressively parallelizing training/simulation, but a higher-performance engine should allow more work to be concentrated on developing the approach itself.

### 2.3 Contribution

The primary contribution of this project is to provide a performant environment/engine for further development of artificial agents, and in particular deep/reinforcement learning agents, for Pokemon battling. We reference the Pokemon Showdown simulator for exact mechanics (e.g. priorities or order of operations), but we remove team-validation and other legality-checks, and we simplify the battle stream interface to reduce performance overhead. We also implement a basic Monte-Carlo Tree Search algorithm to showcase the engine, and we benchmark the agent's performance against its opponent in a couple pre-determined match-ups.

We specifically implement the mechanics and opponent from Pokemon Run and Bun, a ROM hack of the Pokemon Emerald game. We choose this game because of its popularity, relative modernity of mechanics, and most importantly, clearly documented opponent behavior, all of which set this specific game up for future research. However, an additional goal of the project is to produce a code-base that can be adapted to other Pokemon games' mechanics with minimal pain.

## 3 Implementation

### 3.1 Design

We begin by explaining our design choices and the motivations behind them.

#### 3.1.1 TypeScript vs. C++

The goal is to speed up simulation compared to Pokemon Showdown. The most important source of speed comes from simply changing languages from TypeScript to C++. We explain a few reasons for this: in short, TypeScript emphasizes portability while C++ emphasizes performance.

TypeScript has a far more flexible typing system. There's only one numerical type (which could be e.g. an integer or a floating point number), functions can be assigned to variables and passed around, null/undefined values are inherently supported for types and object properties, and it's possible to define a type as allowing any one of several unrelated types.

The developers of Pokemon Showdown use all these features to implement the effects of mechanics (e.g. moves) via a huge array of callbacks that can be defined per-mechanic in neatly organized dictionaries. Then, in each turn, actions are added to a priority queue as callbacks, popped one by one, and executed. These callbacks can, in turn, add more callbacks to the queue to be executed and so on.

The level of meta-programming means that a new contributor doesn't need to understand every single mechanic that has already been implemented if they want to add a new mechanic. An important side effect, though, is it that it's also harder to interpret how the system is functioning as a whole. For example, usually one calls `battle.runEvent('Name')` to run every applicable callback named `onName()` in a certain order. However, `runEvent` can also make calls like `onFoeName()`. There's even some common ability-related flag-checks hidden in the meta-programming, like those for "Mold Breaker." By scattering important and related logic far from each other, the code becomes opaque and susceptible to subtle bugs.

Data storage is also different between TypeScript and C++. In TypeScript, all objects are stored with dynamic allocation on the heap and properties for each object are stored in a hash-map (also located on the heap), risking memory fragmentation and requiring garbage collection to run continuously [Moz23], but allowing succinct expression that Pokemon Showdown developers make generous use of. C++ objects can be statically allocated on the stack if dynamic memory usage isn't needed. We therefore almost strictly use statically-allocated objects, exceptions being vectors that are bound in size and constant, static data structures used for reference.

These differences are largely because TypeScript and its underlying JavaScript runs with a just-in-time (JIT) interpreter while C++ is a compiled language. Compilers require more rigid rules around valid syntax to reach the level of machine-code optimization that C++ has (e.g. copy elision), and interpreted languages require the flexibility from heap allocation to run code without looking ahead. We also note that JIT interpreters have monitor overhead.

We also note that most modern artificial intelligence and machine learning applications and frameworks, including those used to build game-playing agents, are built for Python. In fact, a common pattern is to use Python to design models at a high level and handle performance-critical tasks with C++, leading to proven interfaces between the two like the compiler Cython [BBW+] and the library Pybind11 [Jak]. We built the engine as a C++-first library and used Pybind11 to expose the subset of enums, classes, fields, and member functions that an agent built in Python would need to interface with it. In contrast, there were no established alternatives for a Python-JavaScript interface. The best way to connect a Python agent to a Pokemon Showdown server was via socket, with the possible side-effect of introducing additional TCP/IP latency [Sah].

To summarize, TypeScript's flexibility allows fast development speed, effective adaptation to new Pokemon generations, and centralized storage of data (Pokemon, item, format, etc. mechanics), on top of Javascript's easy integration with front-end web-app development, all helpful to the developers of Pokemon Showdown. To accommodate compilers, C++ has more rigid rules which provide the raw performance we are looking for and make the internal battling logic more transparent. C++ also has a more direct interface with the language/frameworks of choice for artificial intelligence.

### 3.1.2 Statelessness

We model the interface from engine to agent as a stateless server-client relationship. That is, the agent passes a battle state and a choice to the engine, and the engine just returns a possible new state (as a copy) without maintaining any other internal state. One mechanic this enables is repeated queries: since state transitions are inherently stochastic, one might repeatedly pass the same state and action to the engine and obtain a distribution over successor states. Another benefit is reliable testing: rather than fully replicating the *path* to a state of interest, one can simply initialize the state as desired directly to test the engine's output. At its core, though, the model stems from the engine's primary use-case. Agents using, for example, MCTS, abstract the underlying game as a stochastic state-transition function, which translates to this model.

In comparison, Pokemon Showdown maintains the battle state in its server, requiring clients (players) to request to commit their selected choice. After all players commit their choices, it runs the turn and publishes the resulting state back to the clients. Once again, the motivation appears to be security: it is much harder for sneaky clients to modify the “true” battle state if it is kept safe in the server. However, we note that if copying proves to be too expensive in our engine, future versions of the engine may refactor to split running a turn into two calls: an explicit copy function and a state transition that modifies an existing state object.

## 3.2 A (brief) overview of Pokemon mechanics

To better understand how important it is to have transparent internal mechanics, we will review some of the mechanics we implement. Pokemon Run and Bun is implemented with Generation 8 (i.e. Sword/Shield, Brilliant Diamond/Shining Pearl, and Legends: Arceus) Pokemon and mechanics. There are 905 species of Pokemon (and 170 additional “Forme” variations of some of them). Of these 1075 entities, 409 are unobtainable by the player, and 182 are further never on an opponent’s team, many with idiosyncratic abilities. So even though the choice of game to implement reduces the scope of the project, it isn’t by much.

Generations 6 through 9 introduced Mega Evolution, Z-Moves, Dynamax/Gigantamax, and Terastallization, respectively. These are all supported in Pokemon Showdown even though no two mechanics have ever simultaneously been present in the same generation for legacy purposes, but we only implement Mega Evolution as it’s the only mechanic of these that Run and Bun includes.

At its core, a Pokemon’s move attacks a defending Pokemon, dealing damage based on the move’s inherent “Base Power” and the attacker’s and defender’s stats. Some moves can also apply “stat modifiers” to a Pokemon for as long as the Pokemon is on the field. Each stat can have an integer modifier in $[-6, +6]$, multiplying the stat by as low as 1/4 or as much as 4 in various calculations. We follow Pokemon Showdown’s terminology, referring to positive stat modifiers as “boosts” and negative modifiers as “unboosts.” The move’s Base Power can also be multiplied by other moves in that turn, abilities, or items, and the final damage value can further be multiplied by abilities, items, or environmental effects. Finally, the “Type” of the move multiplies its Base Power based on the Type of the attacking Pokemon (the “STAB” effect) or the defending Pokemon (in/super-effectiveness). Sometimes a move may even deal no damage due to these modifiers! For example, a Ground-type Pokemon is immune to Electric-type moves and a Ground-type move deals no damage to a Pokemon with an active Air Balloon item. In summary, multiple of these multipliers can apply to a move.

Since all operations/values are stored and computed as integers, this means that the order of operations matters, as we illustrate in Section 3.2.1. In fact, dozens of people have spent hundreds of hours carefully investigating what results are faithful to the original games, reverse-engineering their mechanics to re-implement in Pokemon Showdown [Smo].

### 3.2.1 Useful examples

A critical concern throughout implementation is ensuring correctness in order of operations. As an example:

1. A Pokemon using the move “Explosion” deals large amounts of damage but faints upon usage.

2. Once a player runs out of usable (i.e. non-fainted) Pokemon, they lose.

3. If the last Pokemon team A uses Explosion and deals enough damage to faint its opposing Pokemon on team B, who wins?

It turns out that the self-induced fainting happens first and team B wins. However, not all effects on the user's side are applied first. If both teams still have usable Pokemon remaining, they simultaneously select which Pokemon to switch in next (i.e. neither side is allowed to wait to see what their foe chooses and adjust their decision). Yet there are other situations in which both teams have pending decisions to make on which Pokemon to switch in, yet one side gets to see the other's choice before making their decision. Our implementation had to faithfully replicate all of these outcomes in the correct scenarios.

Now consider order of operations in damage calculations. Consider a subset of the damage formula:

$$Damage = BasePower \times Critical \times Item,$$

where $BasePower$ is the base power of the move (after more complex modifiers), $Critical$ is a critical-hit multiplier (for Run and Bun, it usually has a 1/16 chance of happening), and $Item$ is the contribution of any item the user or target is holding. Suppose a move has 18 base power, the user is holding a 'Muscle Band' and using a physical attack (boosting its damage by 1.1×), and the move lands a critical hit (a 1.5× multiplier). An example case is "Fury Swipes." If the critical-hit modifier applies first, the move deals

$$\lfloor\lfloor 18 \times 1.5\rfloor \times 1.1\rfloor = 29$$

damage, whereas if the item modifier applies first it deals

$$\lfloor\lfloor 18 \times 1.1\rfloor \times 1.5\rfloor = 28$$

damage. If the opposing Pokemon has 29 HP at that moment, implementing the wrong version would matter a lot!

As a final, somewhat more complex and idiosyncratic example, consider the effects:

- A Pokemon with the ability "Mold Breaker" ignores the effects of its target's ability.
- The move "Dragon Tail" forces its target to switch with another (fixed) Pokemon on the opponent's team.
- The move "Spikes" lay spikes on the opponent's side of the field. When a Pokemon switches into a field with Spikes, it takes damage.
- Pokemon with the ability "Levitate" do not take damage from Spikes.

Now, suppose that a Pokemon with Mold Breaker sets up Spikes on its opponent's side, then uses Dragon Tail to switch its foe out and drag in a Pokemon that has Levitate; that Pokemon takes Spikes damage. But if Dragon Tail drops the foe to zero HP, and the opponent chooses to replace their fainted Pokemon with the Levitating Pokemon, the Levitating Pokemon does not take Spikes damage. From this interaction and others, the community researching Pokemon mechanics concluded that Mold Breaker applies for the duration of the Mold Breaking Pokemon's move, and that switching after fainting occurs after all moves have ended. The understanding has been incorporated into Pokemon Showdown, and we must also include it in our engine.

## 3.3 Monte Carlo Tree Search agent

Monte Carlo methods replace traditional heuristic or domain-specific value estimation methods with a randomized *playout* or *rollout* [SGSM21]. The idea is that in states with clear strengths or weaknesses, an agent that randomly selects actions will not perform that much worse than a more thorough agent but is much cheaper to simulate. A further innovation used Markov decision process theory to refine the random playout, instead using an upper confidence bound to eventually avoid surely-bad states in what is now known as the UCB1 algorithm [CFHM05]. This gives the algorithm a smarter balance between exploring less-explored actions and exploiting actions that have proven to be good. We implement a UCB1 Monte Carlo Search Tree agent with minor modifications, including state-bucketing and randomized mid-turn choices.

From a given state, we give our agent 0.1 seconds to make its choice. In that time, it builds a tree that stores the estimated value $Q(s, a)$ of choosing each action $a$ from each state $s$. On each iteration, it chooses where to either expand the tree and/or add a playout example by stepping down. At each node $s$ that it passes, it chooses its next node by picking the action $a$ that maximizes

$$Q(s,a) + C\sqrt{\frac{\ln(n+1)}{\ln(n_a+1)}},$$

where $n$ is the number of visits to the node $s$ and $n_a$ is the number of times the action $a$ was chosen at $s$. $C$ is a constant. We chose $C = \sqrt{2}$; other values may perform differently. Our formula slightly differs from the theoretical version to avoid domain errors for logarithm or division. Finally, once the agent's time has run out, it chooses the action it visited most because visit-counts have less variance than the learned $Q$-values.

In Pokemon, turns don't perfectly denote states, even if we restrict to states in which a player has to make a choice. For example, if a Pokemon faints, their player chooses their next Pokemon to switch in before the following turn starts and players can choose to use moves. We simplify the model to only add nodes to the tree for start-of-turn states. Our agent makes mid-turn choices (i.e. switches when its active Pokemon faints) at random.

We replicate the documented opponent logic (designed by high-ranking competitive players) for our MCTS agent to play against. The opponent is a memory-less, stochastic agent; that is, their decision at a fixed state is possibly random but the distribution over their choices is consistent.

The reward function for the agent gave -5000 points for a loss and 1000 points for a win. If the agent won, they received an additional 500 points per remaining Pokemon. Thus, the agent was highly incentivized to win, but preferred wins with more Pokemon remaining. The rewards were inspired by the Nuzlocke setting, in which losses force the player to restart the game entirely and fainted Pokemon are unusable for the rest of the run.

### 3.3.1 Scenarios

The first opponent was Youngster Calvin, from the second battle in Pokemon Run and Bun. He has three Pokemon of levels 5-6, and his team has moves with potentially non-trivial effects like flinching ("Bite") to potentially prevent the agent's Pokemon from moving, priority ("Quick Attack") to move before the agent even if his Pokemon is slower, and confusion ("Swagger") to potentially force the agent's Pokemon to hit itself.

Because of the "Endless Candy" and "level-cap" mechanics in Run and Bun, any trainer should have level 12 Pokemon when facing Youngster Calvin. In addition, by the time the player faces him, they should have caught 4-5 Pokemon in addition to their starter Pokemon, so usually the battle is easy to complete. For our tests, we gave the agent far weaker teams to make winning non-trivial.

In the first test, the agent faced Youngster Calvin with just a level 8 Piplup. We also tested giving the agent a level 8 Piplup and a level 8 Poochyena, and in another test, just a level 10 Piplup. Having a higher-level Piplup is more powerful because it can outspeed more of Calvin's Pokemon, and having more Pokemon available is useful since switching removes confusion and any unboosts (i.e. accuracy drops from "Sand Attack").

The second opposing team was that of the first "boss" trainer faced in Pokemon Run and Bun, the Route 104 Team Aqua Grunt. He has three Pokemon of levels 11-12 (player Pokemon are still capped at level 12). His Carvanha is fast and does huge damage with each hit, KO-ing most available Pokemon before they can move. His Croagunk has an ability that lets most of its move inflict the "Poison" status on the player and a berry that boosts its Speed stat when damaged to half-health, and eating the berry unlocks its massive 120 base power move "Belch." Lastly, his Exeggcute has the "Harvest" ability, which recycles its held "Oran Berry," allowing it to constantly heal 10 HP if the player lacks the damage output to quickly KO it. This fight is difficult for most players; beating it requires nontrivial understanding of unusual Pokemon mechanics and some manipulation of the opponent's decision-making.

In each test, the agent faced the Team Aqua Grunt with four Pokemon caught in a real Nuzlocke run of Pokemon Run and Bun: Paras, Buneary, Piplup, and Ralts, each with real randomized IVs (random values assigned to each stat of each Pokemon caught that affect their actual base stats). In one test, the agent faced the boss with this team as-is. In a second, we gave each of the agent's Pokemon Oran Berries, healing their Pokemon in a pinch. In the third test, the agent's Pokemon got Oran Berries and maxed IVs (i.e. implausibly good versions of the same Pokemon), so the agent's team in each test was strictly stronger than the previous.

## 4 Results

### 4.1 Engine performance

We measured how long our engine took to run a turn in certain scenarios and compared them to Pokemon Showdown. The results are displayed in Table 1.

Table 1: Mean and standard deviation of time taken, in microseconds, to run turns of different types on Pokemon Showdown and our engine. Note that fewer trials were conducted on Pokemon Showdown since they had to be run manually.

| **Experiment** | **System** | **Trials** | **Mean Time** ($\mu s$) | **Std. Dev.** ($\mu s$) |
|---|---|---|---|---|
| Start battle | Pokemon Showdown | 20 | 1914 | 17.218 |
| | Engine | 1000 | 0.307 | 0.000910 |
| Run turn | Pokemon Showdown | 20 | 2768 | 23.922 |
| | Engine | 1000 | 17.542 | 4.56 |
| Switch | Pokemon Showdown | 20 | 461 | 63.318 |
| | Engine | 1000 | 1.964 | 0.228 |

Our engine was, for the regular turns that make up most of battling, two orders of magnitude faster than Pokemon Showdown. Starting a battle (e.g. switching in the initial active Pokemon) was almost four orders of magnitude faster. We suspect that this is because Pokemon Showdown loads in each Pokemon's data from the "PokeDex" (a global dict of Pokemon, items, abilities, conditions, etc.) which includes bloat data and callback function objects. If loading copies the data into a new object, this requires both memory allocation (which in a system under heavy load could run into fragmentation-induced slowdowns) and the copy operation itself.

We did not have time to implement an interface with Pokemon Showdown from Python for our experiments. Together with Pokemon Showdown's stateful-server model, this meant we had to manually run each experiment iteration for Pokemon Showdown, whereas our engine could easily be timed repeatedly by starting each trial with a copy of a fixed initial state. As a second consequence, we had to time Pokemon Showdown internally, i.e. by Javascript `console.time()` and `console.timeEnd()` calls. For consistency, we also measured our engine's performance from within our C++ testing suite. As discussed in Section 3.1.1, the C++-Python link our engine uses is quite likely to be faster than the socketed JavaScript-Python link Pokemon Showdown relies on. However, further work must be done, for example by using [Sah], to measure this.

## 4.2 MCTS agent performance

We evaluated the MCTS agent in several situations. The agent was allowed up to 1 second for each turn, a very short amount of time for Pokemon battling (in competitive play, each player is allowed 30 seconds to make their decision per turn) to account for the relative simplicity of the test cases. The results are summarized in Table 2.

When facing the first opponent, we gave the agent 0.1 seconds to make its decision each turn and ran it for 500 trials. In general, the agent was able to run 55-70 rollouts in 0.1 seconds. On the last one or two turns of the game, the agent could run 100-120 rollouts. Given a longer time to decide each turn, or even multiple workers (like in [Wan24]), the agent would reasonably be able to easily achieve tens of thousands of rollouts on each turn.

Even with just a few dozen rollouts per turn, the agent could defeat Youngster Calvin 91% of the time with just a level 10 Piplup, and 36% of the time with a level 8 Piplup and a level 8 Poochyena. It could only squeeze out a win 14% of the time with just a level 8 Piplup, which is fair given that "Sand Attack" accuracy unboosts can stack without either the speed to KO before the opponent can move or another Pokemon to switch to and reset the accuracy unboosts.

When facing the boss (Team Aqua Grunt), we had to give the agent more time to make its decision. Since battles took about twice as many turns as against Youngster Calvin, more rollouts were required to similarly explore each node of the tree and each rollout took longer. We gave the agent 1 second to make its decision each turn and ran it for 100 trials.

In general, the agent was able to run over 500 rollouts in 1 second. On the last one or two turns of the game, the agent could run 1000-1500 rollouts. The agent could defeat Team Aqua Grunt 53% of the time with its team as-is, 72% of the time with Oran Berries, and 83% of the time with both Oran Berries and maxed IVs. Note that stronger teams resulted in a strictly higher win-rate but not necessarily fewer Pokemon fainting, as a result of the reward function penalizing losses far more than the reward for remaining Pokemon.

We also produced in-depth logs of a few example battles against the Team Aqua Grunt, and we saw that the agent discovered interesting strategies against this boss. For example, the agent found that using "Pluck" on the Croagunk would steal its berry, disabling Belch permanently and boosting Piplup's speed. It also consistently used Piplup to bait Exeggcute's "Bullet Seed," switching into Paras to hardly take damage and KO with the super-effective "Bug Bite." Both are techniques we used to find a somewhat more consistent approach to the fight ( 85% chance to win while losing only 1 Pokemon with berries but without max IVs), displaying the "ingenuity" that MCTS can develop in the Pokemon setting.

Table 2: Data from running the MCTS agent with varying teams and against various opposing teams. Averages are over 500 games with 0.1 seconds of decision time against Youngster Calvin and over 100 games with 1 second of decision time against the Route 104 Team Aqua Grunt. We show win rates, the average number of rollouts achieved per-turn, the average number of turns in a game, and the average number non-fainted Pokemon conditional on a win.

| **Team Name** | **Opponent** | **Win rate** | **Rollouts** | **Turns** | **Left** |
|---|---|---|---|---|---|
| Piplup8 | Calvin | 14% | 70 | 6.48 | 1 |
| Piplup10 | Calvin | 91% | 62 | 7.72 | 1 |
| Piplup8+Pooch8 | Calvin | 36% | 57 | 11.69 | 1.15 |
| NoItem | Aqua Grunt | 53% | 537 | 10.9 | 2.08 |
| Berries | Aqua Grunt | 72% | 475 | 12.41 | 2.33 |
| Berry+MaxIV | Aqua Grunt | 83% | 451 | 13.45 | 1.58 |

## 5 Conclusion

### 5.1 Discussion

Our engine was able to achieve two to four orders of magnitude in speedup over the current state-of-the-art, Pokemon Showdown, by:

1. leveraging the implicit speed of the lower-level language, C++, over TypeScript,
2. strongly preferring stack allocation of objects over heap allocation, and
3. using simplified, streamlined control flow instead of a complex callback system.

We also used our engine to implement an MCTS agent that could feasibly scale to tens of thousands of rollouts per second before optimizations.

Still, it's somewhat unclear exactly how much more cost-effectively an agent can be trained and/or run with the current performance gains. One could compare an agent with Pokemon Showdown as a backend engine against an agent using our engine to obtain more direct measurements of rollout times, in particular taking into account socket I/O for the former vs. Pybind I/O for the latter.

More fundamentally, many agents use a combination of MCTS with deep neural networks. The longer that an agent spends computing or waiting for a model's inference, the less that any performance gains in the actual simulator matter. For current state-of-the-art agents, simulation is in fact the bottleneck at inference [Wan24]. If this changes in the future, a proposal to better leverage the speed of our engine is to simulate a single state-action step dozens of times to obtain a distribution over possible successor states before passing them together into the neural network; in essence, batching many step-simulations for a single neural network estimate.

### 5.2 Further work

Future work on the project includes implementation of more idiosyncratic behavior (e.g. "Instruct" and other move-copying moves) and much more in-depth testing of mechanics and interactions, both on the engine side and the opponent decision-making side. We also note that Pokemon Run and Bun has a substantial number of bugs, in which its behavior differs from the behavior in the core games, that may need explicit control flows to replicate. We also foresee potential performance gains from improving cache locality, for example by storing data that is on inactive Pokemon and that are frequently accessed by active Pokemon together, and by storing them separately from other Pokemon data.

Finally, we note that once a successful battler is implemented and trained, there is the distinct challenge of combining it with a team-builder and/or encounter-router to potentially design a system that can beat Run and Bun in a Hardcore Nuzlocke, a feat that has only been accomplished 55 times to date since the game's release two years ago.

## 6 Source code

The source code and documentation for the project can be found at `https://github.com/derekdong623/CPSC490`.

## References

[BBW$^+$] Stefan Behnel, Robert Bradshaw, David Woods, Matúš Valo, and Lisandro Dalcín. Cython. `https://github.com/cython/cython`.

[CFHM05] Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Oper. Res.*, 53:126–139, 2005.

[HL19] Dan Huang and Scott Lee. A self-play policy optimization approach to battling pokémon. In *2019 IEEE Conference on Games (CoG)*, pages 1–4, 2019.

[Jak] Wenzel Jakob. pybind11. `https://github.com/pybind/pybind11`.

[Lin19] Norstrom Linus. Comparison of artificial intelligence algorithms for pokémon battles. Master's thesis, Chalmers University of Technology, 2019.

[LT17] Scott Lee and Julian Togelius. Showdown ai competition. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 191–198, 2017.

[Moz23] Mozilla Contributors. Memory management - javascript — mdn. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management`, 2023. Accessed: 2025-04-20.

[pkm] Pokemon. `https://www.pokemon.com/us`. Accessed: 2025-01-28.

[RP20] Sebastian Risi and Mike Preuss. From chess and atari to starcraft and beyond: How game ai is driving the world of ai. *KI - Künstliche Intelligenz*, 34, 02 2020.

[Sah] Haris Sahovic. Poke-env: Pokemon ai in python. `https://github.com/hsahovic/poke-env`.

[SGSM21] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931, 2021.

[sho] Pokemon showdown. `https://pokemonshowdown.com/`. Accessed: 2025-01-28.

[Smo] Smogon Contributors. Scarlet violet research - smogon. `https://www.smogon.com/forums/threads/scarlet-violet-battle-mechanics-research.3709545/`. Accessed: 2025-04-20.

[SRLR20] David Simões, Simão Reis, Nuno Lau, and Luís Paulo Reis. Competitive deep reinforcement learning over a pokémon battling simulator. In *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 40–45, 2020.

[Wan24] Jett Wang. Winning at pokémon random battles using reinforcement learning. Master's thesis, MIT, 2024.