

EliteFourBot: A Pokemon Showdown Game Agent

Nicholas Allen
Department of Chemical
Engineering
Stanford University
naallen@stanford.edu

Ivan Salinas
Department of Symbol Systems
Stanford University
isalinas@stanford.edu

Henry Weller
Department of Mechanical
Engineering
Stanford University
hweller@stanford.edu

Task Definition

This paper explores the application of the Minimax Algorithm in the two-player Pokemon Showdown game. Our goal was to create a bot that can play competitively on the generation 1 random battle Pokemon Showdown servers, with a ranked ELO of at least 1600. We chose to narrow the scope of what generations our agent could play to strictly generation 1, so that we could work with the smallest rule set and create a competitive agent in the time we had.

Infrastructure

To create a simulator that we can test with, we forked the open source codebase of Pokemon Showdown¹, written in Node.js. The codebase contained detailed Pokemon stats, movesets, and other information that we would have had to recreate on our own without it; this allowed us to extend the basic simulator, which featured a bot that selects moves randomly, to include a bot that included both our baseline and the bot that chooses moves based on the minimax algorithm. Instead of randomly selecting the bot's moves, we pass all of the information we have about our Pokemon team, as well as the enemy Pokemon we can see, through a Flask endpoint to a python script which runs the entirety of our algorithm, and returns the selected action to the simulator (figure 1). Using this simulator, we were able to extensively test our algorithm using both randomly generated teams and custom test cases.

Despite node.js' non-blocking I/O model, we were able to pause the simulator to prompt for human user, but this was a poor interface from a user's perspective. We wanted to implement a more sophisticated GUI so that a human could play against the agent on the Pokemon Showdown server. In order to create this intuitive user interface, we implemented a minimized version of the Pokemon Showdown client, borrowing some functionality from an existing template we found online.

After the bot is initiated from the terminal, it loads the index file and instantiates a new client which takes in the server, port, and server ID we specified in the configuration file; the config file is also used to specify the agent's online username, connection timeout length, and some other settings. The client then creates a web socket connection and an emitter which listens for, and accepts, an opponent's battle request. The index file then continues to load the battle module, which is called when a new challenge is sent from the Pokemon Showdown server and each turn thereafter. When our agent module receives a request during battle, our agent selects a new Pokemon if necessary, or chooses the best available move by calling the Flask endpoint we specified above. For optimization purposes, our Flask endpoint is only called when there are at least two moves to choose from, and reverts to the baseline moves if any issues arise from our minimax algorithm.

To be able to use the minimax algorithm, we also implemented a pared-down simulator in Python so that we could simulate the results of actions without having to pass them back and forth between the Python code and the Node.js application. This simulator implements the basic functionality of Pokemon battles, by directly using move data and stats from Pokemon Showdown, with special code written for implementation of more elaborate moves. The simulator implements many of the mechanics of Pokemon, including move order, bonuses from Pokemon and move types, and status effects/stat boosts.

Notably, since Pokemon involves a good amount of randomness in damage calculation and in probability of applying effects, we generalize everything by taking expected values for damage, probability of having a status effect, and stat boosts. For instance, if a move has a 10% chance of applying the status effect *Sleep*, then the simulator assigns Sleep a value of 0.1, despite it being a boolean in the actual game. When a Pokemon is sleeping, it cannot move, so we model that by scaling accuracy in damage calculations by the probability that the status effect was applied (0.9 in this case). The same is done for all status effects, such as applying partial damage for effects that deal damage over time, as well as boost levels. We attempted using a version of the simulator where we would return all possible outcomes of a move with associated probabilities, but the state space ended up being too large to be able to run the agent with reasonable performance.

Similarly, when a move deals damage, we calculate the expected value for damage (taking into account the random factor in the damage calculation, as well as chance for "critical hits" which deal bonus damage) and apply that as the damage taken. This is problematic, as it does not properly model when a Pokemon would faint (e.g., when

health percentage is 0.0, the Pokemon is only expected to be knocked out about 50% of the time). We originally did not account for the probability of a Pokemon fainting, but that resulted in sub-optimal moves, as a Pokemon would weight taking damage at critical health levels the same as taking damage at high health. This meant the agent would make bad move choices, such as applying stat boosts at low health (when you want to maximize damage or heal yourself) or healing yourself at high health in anticipation of damage (when you want to be boosting stats and dealing damage, since you aren't at immediate risk of fainting). To remedy this, we defined an arbitrary non-linear function to return the probability of a Pokemon being fainted given its current health, $p_{fainted} = \frac{1}{1+e^{currhp*10}}$. We used this probability both for the scoring function, so that taking damage at high health impacted score less than taking damage at low health, and also to scale accuracy, such that a Pokemon with a 75% chance of being fainted as per our function has its expected damage output for all its moves reduced by 75%, and similarly with its probability of performing other secondary move effects such as status effects. We also used this factor to scale the impact of status effects and boosts on score, such that positive stat boosts and applying status effects have a higher impact on score at lower $p_{fainted}$. This was done based on the intuition that it is better to stat boost at higher health, when it is safer, and that status effects such as *Sleep* are much more devastating when applied to a Pokemon at full health, rather than one that is about to faint. These improvements were instrumental in getting our agent to perform moves such as stat boosts and moves that apply status effects when it is safe to do so, and also to prioritize self-preservation and knocking out the opponent before they can knock us out when we are at low health.

During the development process, we spun up a local server using the Pokemon Showdown source code and specified the localhost address in the configuration file. We later tested this in production using Pokemon Showdown's main server, sim.smogon.com. A node command from the Pokemon-Showdown-Bot directory is necessary to initialize the bot. The minimax server (`minimaxserver.py`) also needs to be running from the `sim` directory to expose the endpoint which returns the best move to our client. If a local server is needed, the `Pokemon-showdown` node module must also be started.

Running The Minimax Agent

Starting a Pokemon-Showdown Server

Go to the `Pokemon-Showdown` directory in your terminal and enter
``node pokemon-showdown``

This will start Pokemon-Showdown, using your computer as a local server. Detailed instructions on this process can be found in the Pokemon-Showdown README file.

Playing an AI Bot

Go to Pokemon-Showdown-Bot[./Pokemon-Showdown/Pokemon-Showdown-Bot] directory and enter ``node bot`` in your terminal.

This will start the AI bot. By default, the bot will look for the localhost:8000. You can change this in the Pokemon-Showdown-Bot config.js (./Pokemon-Showdown/Pokemon-Showdown-Bot/config.js) file. The main server is ``sim.smogon.com``. Instructions on running the server above.

The bot calls on a Flask endpoint to get the best decision after each turn. To get the flask server running, go to the sim (./Pokemon-Showdown/sim) directory and run enter ``python minimaxserver.py``. This will start the minimax server and expose the flask endpoint.

Approach

In evaluating the performance of our bot, it is important that we establish a reasonable oracle and baseline. We chose to use one of our team members, Henry, who is well-experienced at generation 1 Pokemon battles, as our oracle and a bot that performs marginally better than a random bot as our baseline. The baseline bot selects moves based on which move has the lowest possible number of usages, an indicator of rare or powerful special attacks. Repeatedly using the same move performed worse than random since the move with the lowest number of usages is not always the most effective against every type of Pokemon, and certain moves, like Hyper Beam and Rest, sacrifice a turn to achieve their intended effect. For this reason, we chose one of the two moves with the lowest number of usages at random. Upon our Pokemon fainting, we selected a healthy Pokemon with a type favorable against the enemy Pokemon's type. After implementing our baseline heuristics, we tested the bot which selects moves and Pokemon switches randomly. In 1000 Pokemon battles, our baseline bot defeated the random bot slightly under 60% of the time, which we deemed to be reasonably better than random. Our oracle was able to defeat the random bot 10/10 times, and the baseline bot 9/10 times.

Our agent uses the Minimax algorithm to determine the optimal action on a given turn. It uses a modified Minimax algorithm which allows for simultaneous moves, as per in Pokemon battles, by performing alternating minimization and maximization over all possible combinations of our actions and the opponent's actions. To model this simultaneous interaction as turn-based, we created a list of all possible moves that can be made by our Pokemon, and the possible moves of the enemy Pokemon that is out, and pass each of these action pairs, as well as both of the Pokemon's current stats, into a function that resolves the actions and returns a score using a scoring function. Our scoring function evaluates our team's info along with the info of the opposing Pokemon we can see to produce a single number representing the current game state. Our scoring function is nonlinear: health recovery moves are prioritized when the Pokemon has low HP, while stat boosting moves are prioritized when our Pokemon has high HP. The agent then simulates gameplay up to three turns in the future using the scoring function to pick the move that maximizes an objective function representing game score. In this way, we can perform alternating minimization and maximization of the resulting game score to have our Pokemon pick a move, similar to the standard turn-based minimax algorithm. More on how the scoring function is implemented in the simulator can be found in the infrastructure section.

We tested our minimax implementation against the baseline and oracle by facing them against each other in battle. Over several iterations of 100 rounds, the minimax agent won 73% of the time against the baseline. Henry beat the minimax agent 7/10 times, a 300% improvement from the baseline. We also tested our agent against students and TA's during the poster session to see how it would perform against non-competitive players. The minimax agent was able to beat 6/10 human players, with a 100% victory rate against the TA's.

Literature review

The NYU Pokemon Showdown AI Competition² frames the Pokemon Showdown problem in a very similar infrastructure to the one we used, and also proposes several possible algorithms that can be employed towards creating a competitive agent. Of the proposed algorithms, the most interesting and relevant ones were their proposal of minimax, which differed from ours, as well as a One-Turn Lookahead algorithm.

The minimax algorithm is obviously most similar to what we chose, but is far more limited in its evaluation function and how states are defined. Each node in the search tree is defined as the worst case-scenario for a given move, whereas ours takes into

account the resultant state that occurs after both Pokemon make their move. The evaluation function also only takes into account the health of the two Pokemon that are out, with no consideration to stat boosting moves, status effect moves, or the condition of the rest of the ally Pokemon, as our does. We think that this is a naive approach relative to ours since there are many scenarios in which performing a stat boosting move or status effect move is preferable to purely damaging moves, and those will always be ignored in this model. The paper points out that this is a “knowledge-free” model, a limitation which is shared with our model since minimax algorithm can be considered an “ad-hoc” model that does not take into account past turns. The performance* of this model is quite effective against the other algorithms employed, winning the majority of battles against all the algorithms referenced except for One-Turn-Lookahead and Pruned BFS.

The One-Turn Lookahead strategy referenced in the paper is one that is closely associated with our desire to incorporate switching with our algorithm. It involves assessing the damage that the current Pokemon can do with its moveset, and switches Pokemon if an ally Pokemon has a more damaging move. This is a naive approach, since it doesn’t take into account that switching Pokemon causes us to lose an action where we can damage the enemy; we tried to model this in our algorithm by allowing the Pokemon to switch if that yielded a game state with higher score with lookahead of an arbitrary depth. However the depth to which we could lookahead was severely limited by the large branching factor that occurs on each turn; considering each of our Pokemon’s actions is incredibly computationally intensive, even with Alpha-Beta pruning, so we had to limit our depth to 2. At this depth, it is highly unlikely that the game state in two turns will yield a high score with switching, as switching almost always leads to the Pokemon switching in receiving damage, lowering the score for that subtree. Thus, we were not able to properly employ switching in a similar manner as this algorithm.

This algorithm was able to outperform all others, except for Pruned BFS, with about even performance against the minimax algorithm presented by this paper. This suggests that an optimized version of our algorithm that incorporates switching effectively could be incredibly effective against many competing algorithms.

Error analysis

We ran some small, controlled tests to assess our bot’s performance in making correct decisions in specific scenarios. We tested three specific cases that pop up often in

Pokemon games: choosing moves based off priority/damage tradeoff, choosing when to boost stats and when to deal damage, and picking moves based on secondary effects.

The first case deals with moves that have higher priority (and thus, are used before any other moves with lower priority regardless of speed) in scenarios where they are necessary. For instance, if your Pokemon is at low health, and your Pokemon's speed is lower than your opponent's, then you would want to use a high-priority move like Quick Attack over moves with lower priority, because you're likely to get knocked out before you can deal damage if you don't use Quick Attack. For this test case, we set up a scenario with two Pokemon, where our Pokemon had a lower speed stat but otherwise equal stats, with our moveset consisting of Earthquake (a normal priority but high-damaging move) and Quick Attack (a lower-damaging but high priority move), and the opponent having Tackle in their moveset. When we tested this scenario with both Pokemon at full health, our Pokemon favours Earthquake, as expected. When at full health, we aren't in danger of being knocked out, thus we can play less safely and prioritize dealing damage. However, when both Pokemon are set to 10% health, our bot instead chooses Quick Attack, as this move bypasses the typical speed check and allows us a chance to knock out their Pokemon before they can knock us out. Furthermore, when we set our Pokemon's speed stat to be higher than theirs, our Pokemon goes back to favouring Earthquake. This is expected, since Earthquake deals more damage than Quick Attack, and will give us a higher chance of knocking their Pokemon out in a scenario where we go first anyway. The last test we ran was with both us and the opponent having Quick Attack in our movesets, with our Pokemon having a higher speed stat. As expected, our bot chooses Quick Attack again, to anticipate the opponent using Quick Attack and knocking us out before we can deal damage with Earthquake.

Next, we looked at the case of stat boosting. There are many moves in Pokemon that apply a multiplier to one of either your or the opponent's Pokemon's stats. These moves typically don't deal damage, so there is some tradeoff associated with their use. We set up a similar scenario with two Pokemon, both with the same stats, with our Pokemon having Swords Dance (a move that multiplies damage from physical attacks by 1.5) and Slash in its moveset, and their Pokemon having Tackle in their moveset. With both Pokemon at full health, and testing with depth set to 2, our Pokemon chooses Slash. This is relatively unsurprising, as we can't appreciate the benefits of multiplying damage by 1.5 over two moves. However, with a depth of 3, our Pokemon instead favours Swords Dance, as the lookahead is now far enough to where the minimax agent can see the payoff. We then tested the same scenario but with both Pokemon at 10% health. This time, at both depths, our Pokemon chooses Slash. This is also an expected

move a human player would make, since we don't want to be boosting our stats when we're at risk of getting knocked out. The last test we ran for this case was a sanity check to ensure that our Pokemon wouldn't boost unnecessarily by replacing Slash with Blizzard, which is a Special attack and thus is unaffected by the boost from Swords Dance. As expected, our agent always chooses Blizzard in this case, as using Swords Dance is unnecessary and would have no payoff.

The last case we wanted to test was how our agent chose moves with status effects. Status effects are statuses that are applied to Pokemon by certain moves, often with an associated probability of applying the effect. Status effects include Freeze, which prevents a Pokemon from moving, and Burn, which deals periodic damage and halves the effective Attack stat. We first decided to test when our agent chooses moves that can apply the Freeze status. As before, we set up two Pokemon with the same stats, except our Pokemon was a Water-type, and theirs had their moveset consisting of Thunder, putting us at risk of taking heavy damage. Our Pokemon had the moves Blizzard, a move with 90% accuracy, and a 10% of applying Freeze to the opponent, and a modified version of Blizzard with 100% accuracy, but no chance of applying Freeze. With both Pokemon at full health, our agent favours the original form of Blizzard, to take the chance of applying Freeze and preventing us from taking any further damage. However, with both Pokemon at 10% health, the agent favours the modified Blizzard, as we want to prioritize knocking them out versus freezing them, since they're already at low health anyway. We then tested a similar case with Burn. As mentioned, applying Burn to a Pokemon greatly reduces its physical attack damage. Thus, we set up a test case with two Pokemon, where our Pokemon knows Flamethrower, a move that has a chance to apply Burn, and a modified Flamethrower move that deals more damage but doesn't apply burn. The opponent had either Slash, a physical damage-dealing move, or a modified Slash which instead deals Special damage, thus is unaffected by Burn, and is otherwise identical. In the latter case, with the modified Slash move, our agent favours the modified Flamethrower move, since applying Burn would be less impactful, so instead the agent favours dealing more raw damage. However, in the case of the opponent having only the Physical damage-based Slash move, our agent favours the original Flamethrower, in an attempt to greatly reduce the opponent's damage output through applying Burn.

Conclusion

Our current implementation only considers switching when the agent's Pokemon faints. A crucial next step would be to incorporate Pokemon switching within the minimax algorithm before our agent's Pokemon faints, as switching is an important part of

Pokemon strategy. Although our code does have support for switching, and we have run experiments with switching, we find that switching results in greatly reduced performance. The agent appears to favour switching repeatedly in order to spread damage out among all Pokemon, rather than actually performing moves with the Pokemon. This is likely due to a quirk in the scoring function, where changes in health impact the score far more at low health than at high health. Possible ways to fix this could be to modify the scoring function for switched-out Pokemon to be more linear, or applying a score penalty for every switch action performed, thus disincentivizing repeated switching.

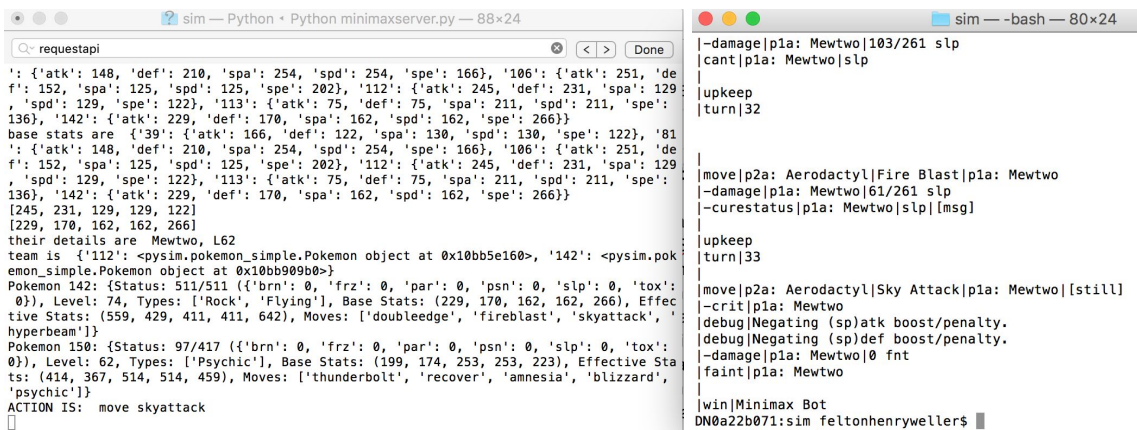
As previously mentioned, our agent considers a depth of 2 when maximizing its battle performance due to the state's large branching factor. We believe a greater depth is ideal for long multi-turn exchanges including complicated status-boosting strategies. As rules continue to increase with each successive generation to include items, weather, terrain, mega-evolutions, and so on, this state space will only grow, as will the complexity of our scoring function. While we achieved modest success against the baseline and inexperienced players, and saw relatively large improvements against Henry, this solution will not scale adequately to battle competitively in future generations. Rather than adding more heuristics and further complexities to our scoring function, we believe that approximating the scoring function using a neural network would be a better approach to scaling our code in the future.

Appendix A: References

1. <https://github.com/Zarel/Pokemon-Showdown>: Pokemon Showdown server
2. http://game.engineering.nyu.edu/wp-content/uploads/2017/02/CIG_2017_paper_87-1.pdf:

This paper assesses the performance of algorithms against one another both in a per round basis (does the Pokemon choose a move that is better than its opponent), and one a battle basis. Given performance is more closely tied to winning games than single round outcomes in our opinion, we will only consider the battle statistic.

Appendix B: Figures



```
sim — Python • Python minimaxserver.py — 88x24
requestapi
{
  'atk': 148, 'def': 210, 'spa': 254, 'spd': 254, 'spe': 166, '106': {'atk': 251, 'de
f': 152, 'spa': 125, 'spd': 125, 'spe': 202}, '112': {'atk': 245, 'def': 231, 'spa': 129
, 'spd': 129, 'spe': 122}, '113': {'atk': 75, 'def': 75, 'spa': 211, 'spd': 211, 'spe':
136}, '142': {'atk': 229, 'def': 170, 'spa': 162, 'spd': 162, 'spe': 266}}
base stats are {'39': {'atk': 166, 'def': 122, 'spa': 130, 'spd': 130, 'spe': 122}, '81
': {'atk': 148, 'def': 210, 'spa': 254, 'spd': 254, 'spe': 166}, '106': {'atk': 251, 'de
f': 152, 'spa': 125, 'spd': 125, 'spe': 202}, '112': {'atk': 245, 'def': 231, 'spa': 129
, 'spd': 129, 'spe': 122}, '113': {'atk': 75, 'def': 75, 'spa': 211, 'spd': 211, 'spe':
136}, '142': {'atk': 229, 'def': 170, 'spa': 162, 'spd': 162, 'spe': 266}}
[245, 231, 129, 129, 122]
[229, 170, 162, 162, 266]
their details are Mewtwo, L62
team is {'112': <pysim.pokemon_simple.Pokemon object at 0x10bb5e160>, '142': <pysim.pok
emon_simple.Pokemon object at 0x10bb909b0>}
Pokemon 142: {Status: 511/511 ({'brn': 0, 'frz': 0, 'par': 0, 'psn': 0, 'slp': 0, 'tox':
0}), Level: 74, Types: ['Rock', 'Flying'], Base Stats: (229, 170, 162, 162, 266), Effec
tive Stats: (559, 429, 411, 411, 642), Moves: ['doubleedge', 'fireblast', 'skyattack', '
hyperbeam']}
Pokemon 150: {Status: 97/417 ({'brn': 0, 'frz': 0, 'par': 0, 'psn': 0, 'slp': 0, 'tox':
0}), Level: 62, Types: ['Psychic'], Base Stats: (199, 174, 253, 253, 223), Effective Sta
ts: (414, 367, 514, 514, 459), Moves: ['thunderbolt', 'recover', 'amnesia', 'blizzard',
'psychic']}
ACTION IS: move skyattack

sim — -bash — 80x24
|-damage|p1a: Mewtwo|103/261 slp
|cant|p1a: Mewtwo|slp
|
|upkeep
|turn|32
|
|move|p2a: Aerodactyl|Fire Blast|p1a: Mewtwo
|-damage|p1a: Mewtwo|61/261 slp
|-curestatus|p1a: Mewtwo|slp|msg]
|
|upkeep
|turn|33
|
|move|p2a: Aerodactyl|Sky Attack|p1a: Mewtwo|still]
|-crit|p1a: Mewtwo
|debug|Negating (sp)atk boost/penalty.
|debug|Negating (sp)def boost/penalty.
|-damage|p1a: Mewtwo|0 fnt
|faint|p1a: Mewtwo
|
|win|Minimax Bot
DN0a22b071:sim feltonhenrywellers$
```

Figure 1. Minimaxserver.py serving actions to battle-stream-minimax.js