Pokémon Evolutionary Agents

Dawn McKnight and Zeyi Wang

March 2020

Abstract

Artifical intelligences can be tailored for a variety of means, including performance, or how strong agents are, and explainability, or how human-comprehensible the resulting agents are. In this paper, we explore the usage of an evolutionary algorithm equipped with a domain-specific language for the purpose of finding explainable, relatively strong artificial intelligence for the popular video game series *Pokémon*.

Keywords— XAI, Explainable AI, Pokémon, RPG, Evolution, DSL

Introduction

The popularity and market share associated with computer games have elicited much work in designing automatic game-playing agents. While the default assumption might be that agents better at playing the game in general are most desirable, the reality is that agents need to be tailored for a variety of purposes. For example, some games attempt to imbue agents with personality through them favoring different tactics; other games require agents to have weaknesses in certain areas that players can learn to take advantage of; still others necessitate agents that feel fair to players.

A focus of recent years has been on *Explainable* AI, or XAI. Explainable AI aims for artificial intelligence that humans can understand and influence the design of in an easily-interpretable manner. This is in contrast to, for example, a neural network approach to artificial intelligence, which might leave users with a black-box architecture that resolves a given set of inputs with an appropriate output, but with no understanding as to why or how to easily influence the system.

One video gaming series that has been targeted with artifical intelligence creation for the past couple of decades is the popular *Pokémon*. However, most attempts for the series have been either manual, or via neural networks, the former of which has the downsides of requiring human involvement and domain expertise and the latter of which lacks explainability.

In this paper, we discuss the creation of an explainable artificial intelligence system for the video game series *Pokémon* through an evolutionary algorithm. The evolutionary algorithm uses a domain-specific language to produce Python scripts for battling. These scripts invoke calls to `poke-env`, an interface for interacting with the popular online battling system *Pokémon Showdown*, for battling with a local version of *Pokémon Showdown*. We present our results from repeated evolutionary generations and discuss what can be done going forward to improve our system.

Related Work

As mentioned, *Pokémon*'s popularity has meant that it has already attracted attention regarding artificial intelligence design. Aside from research papers investigating specific qualities of *Pokémon* battling, such as the best of a set of standard reinforcement learning techniques [1], the battle system was also selected as the target of a 2017 artificial intelligence tournamnet [2]. Explainability was the focus of neither of these, however.

The *Pokémon* battle system was also the target of a study for formally modelling sleeping and healing in combat [3], which found that using both of these techniques (which require more finesse to use than a greedy damage approach) improved performance, further indicating *Pokémon*'s nontriviality and, moreover, its efficacy as a research target.

Finally, in terms of results for DSL-derived agents, a project designing agents for solving Picross [4] found (through a technique other than ours, but from a DSL nonetheless) that they were able to derive simple, human-understandable rules that outperformed those suggested in numerous guidebooks for the puzzle type.

Background

Pokémon

Pokémon is a popular Japanese Role-Playing Game (JRPG) series wherein players befriend creatures called "Pokémon". A large part of the core games in this series are the battles: players form and train teams of Pokémon to engage others' teams. These teams are composed of up to six members, although at any given point in the battle each player (known as a "trainer") has one Pokémon active at a time. Battles are turn-based, with each player selecting a move and a variety of factors determining who goes first; at each round, players either select an action for their active Pokémon to take, or they attempt to swap Pokémon. The goal is to defeat all of your opponent's Pokémon through your actions, the consequences of each of which are determined by a Pokémon's stats (i.e. HP, Defense, Special Attack), types (i.e Fire, Water, Ghost), previous moves used, RNG, and more.

External Resources

We use three main external resources for our project: a version of the *Pokémon* Battling simulator *Pokémon Showdown*, a Python library for interacting with *Pokémon Showdown* known as `poke-env`, and the TrueSkill rating system.

Pokémon Showdown, or PS, is a website and code repository for simulating the battle system of *Pokémon*'s main-series games. We specifically use a lightly-optimized version of *Pokémon Showdown* by Haris Sahovic for use in machine learning.

`poke-env` is a Python interface for creating agents for and interacting with *Pokémon Showdown*. All of the agents we create extend `poke-env`'s `Player` class in order to interact with the PS server.

TrueSkill is a rating system for game players developed by Microsoft Research for ranking and matchmaking service. This system estimates players' skills by a Bayesian inference algorithm. We use the *TrueSkill* system to obtain a better approximation to scripts' strength than simply counting the number of wins.

Methodology

To create our artificial intelligence agents, we pit agents generated from a domain-specific language against each other and select for/create new agents through genetic programming.

Agents are tested against each other using a *Pokémon* battling simulator known as *Pokémon Showdown*.

Domain-Specific Language

All of our agents are built up from a *domain-specific language*, or DSL. Our domain specific language consists of a vocabulary and a set of rules for deriving scripts using that vocabulary.

Our DSL's vocabulary contains a set of symbols that can be used to build up our program. More intuitively, each of these symbols is either a method call used to check information about the battle, a literal comparison value (such as 4), or an operator for joining these together. For example, the phrase in the following script

```
def choose_move(self, battle: Battle):
    for move in battle.available_moves:
        score = 0
        #phrase start-------------------------
        if DSL.type_multiplier(battle, move)>=4 and DSL.gets_STAB(move):
            score += 10
        #phrase end---------------------------
        move_scores.append(score)
```

is built up from the vocabulary elements '`DSL.type_multiplier`', ' '`(`', '`battle`', '`,`', '`move`', '`)`', '`>=`', '`and`', and '`DSL.gets_STAB`' (the symbols '`score+=`´, and '`10`´ are generated as part of a separate system). These come from a larger vocabulary, and a *context-free-grammar* defines how they can be laid out.

Each of the checks created from our DSL's components are boolean checks on whether to increment/decrement the "score" for an action our agent is considering. Due to this usage, we alternatively refer the generated boolean part of each of these checks as an *if-clause*.

Genetic Programming

Regarding genetic programming, our process is as follows: first, we initialize random agents using our DSL/grammar. Next, we take the best-fit of those agents (the elites) and save them for the next round. We randomly breed our population using a crossover method and perform random mutations on the offspring. We repeat the competition, crossing over, and mutating for some number of generations, and the elites of the last round are the agents produced by our genetic algorithm.

Initialization

We initialize the population P with k randomly sampled scripts based on the DSL and the derivation rules.

Fitness Function

We evaluate the population by having the scripts play with each other. We first initialize the *TrueSkill* rating of all scripts in P to a default value. We then randomly form pairs of scripts and have each pair of scripts finish a game. We repeat the game playing for n times and obtain $\frac{|P|}{2} * n$ battle records. We iterate through the records and update the scripts' *TrueSkill* ratings. The *TrueSkill* values are stored for future usages.

Generations

We use elitist selection, also known as *elitism*, to ensure survival of the fittest agents by keeping the e elites with the highest *TrueSkill* value in the next population. We then reproduce scripts by randomly sampling tournaments of size t, selecting the best two scripts in the tournaments, and using the two scripts as crossover parents. We then mutate the children obtained by crossover and add the mutated children to the next population. We repeat this process for g generations to obtain a total of $k * g$ scripts.

Crossover

We cross over two scripts by randomly swapping sub-trees of the scripts' abstract syntax trees (ASTs). We first compute the intersection of node types of two ASTs. We randomly select one node type in the intersection. We then randomly select one node in each AST of that type, and use the selected nodes as the sub-tree roots. We swap those two sub-trees and obtain two new ASTs, and re-render the ASTs to scripts.

Mutation

We mutate a script by re-deriving a sub-tree of the script's AST. We first randomly select a node in the AST that does have a fixed derivation rule. We then remove the sub-tree rooted at the selected node and create a new sub-tree by re-deriving based on the rule associated to the node type.

Code Snippet

In Figure 1 we provide a snippet of Python code of the algorithm with some details hidden.

Results

We use the same evaluation function used for evaluating population fitness. However, this time we have all scripts from all generations battle with others. We then compare the relative performance of each generation by aggregating the performance of the scripts within the generations.

For each set of experiment, we only change one parameter and keep the other parameters their default values.

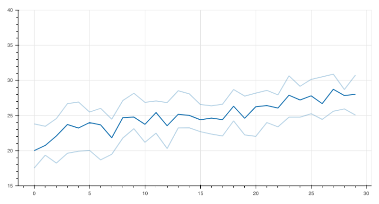
```
num_elites = e
tournament_size = t
population_cap = k
epochs = g
num_games = n

population = get_random_population(population_size=population_cap)
generations = []
for i in range(epochs):
    evaluate_population(population, num_games)
    generations.append(population)
    next_population = []
    next_population.extend(get_elites(population, num_elites))
    while len(next_population) < population_cap:
        left, right = get_tournament_parents(population, tournament_size)
        children = crossover(left.tree, right.tree)
        for child_tree in children:
            child_tree = mutate(child_tree)
            next_population.append(exec_tree(child_tree))
    population = next_population
```

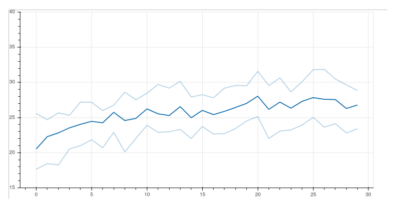
Figure 1: A description of our genetic algorithm

We use number of generations $g = 30$, tournament size $t = 5$ for all experiments. We use number of elites $e = 2$, size of the population $k = 32$, number of games $n = 5$ as the default values.

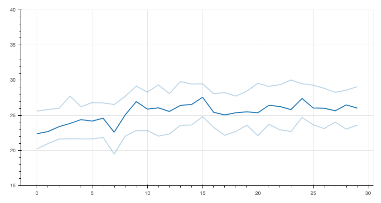
For all graphs, the x-axis is the generation number ranging from 0 to 29, and the y-axis is the *TrueSkill* value of scripts. For each graph, there are three lines. The top line is the skill value on 75% percentile. The middle line is the median of skill values. The bottom line is the skill value on 25% percentile.



(a) $k = 32$

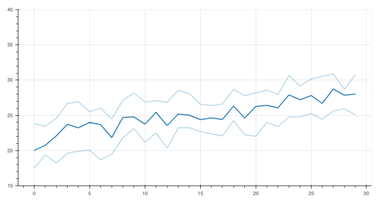


(b) $k = 48$

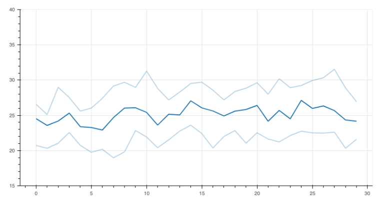


(c) $k = 64$

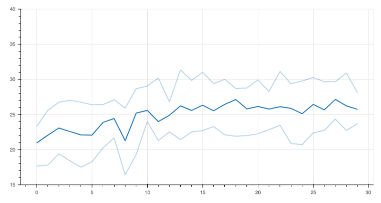
Figure 2: Changing Population Size k



(a) $e = 2$

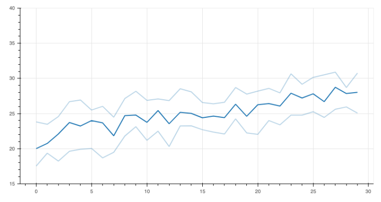


(b) $e = 4$

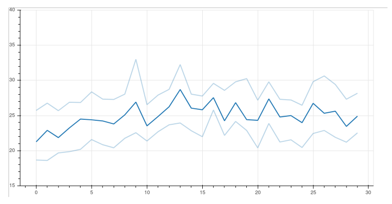


(c) $e = 8$

Figure 3: Changing Number of Elites e



(a) $n = 5$



(b) $n = 10$

Figure 4: Changing Number of Games n

```python
class Script_db990e3ff9cb47beb735a0064190e9a9(Script):
    def choose_move(self, battle: Battle):

        available_moves = battle.available_moves
        if not available_moves:
            return random.choice(battle.available_switches)

        dsl = DSL(battle)
        move_scores = []
        for move in battle.available_moves:
            score = 0
            if ((dsl.player_has_status_effect(Status.TOX) or dsl.move_is_status(move))):
                score += -8
            if (dsl.move_is_physical(move)):
                if (((dsl.move_accuracy(move) >= 50) or (dsl.move_accuracy(move) >= 90))):
                    score += 3
                if (dsl.check_move_sds_if_hits_opp(move)):
                    score += 6
                if ((dsl.check_move_rainy(move) or dsl.opp_has_status_effect(Status.SLP))):
                    score += -1
            if ((dsl.gets_stab(move) and (dsl.move_base_power(move) <= 233))):
                score += 4
            if ((dsl.move_is_physical(move) and (dsl.move_type(move) != PokemonType.GHOST))):
                if ((dsl.player_health_percent() >= 0)):
                    score += 5
                if (((dsl.type_multiplier(move) >= 1) or
                    dsl.check_move_inflicts_status_condition(move, Status.SLP))):
                    score += 7
                if ((dsl.check_move_sds_always(move) and dsl.check_move_sunny(move))):
                    score += -3
            move_scores.append(score)

        best_move = available_moves[move_scores.index(max(move_scores))]
        return best_move
```

Figure 5: One of scripts produced by our algorithm

Discussion

As seen in figures 2-4, our evolutionary approach does work in producing successively better scripts- to an extent. There is occasionally some noise, however, as evidenced by Figure 4(b), which logically shouldn't be doing any worse from a growth perspective than Figure 4(a).

More interesting are the scripts produced. These have the advantage of being heuristics that a player could feasibly follow. However there are some aspects that don't make sense, such as the check at the end for a move that "selfdestructs always" and that "creates sunshine" (such a move does not exist). Many of the checks do make sense, though; the first check ("has status effect Toxic") is logical because a Pokémon inflicted with Toxic takes successively more damage the more consecutive turns they are active, so switching them out is a potentially sound strategy.

Going forward, we would like to experiment with how well the AI can perform without the scoring system (to eliminate one more thing a player following an AI's strategy would have to follow). We would also like to see if we could get better results on an earlier generation of *Pokémon*'s battle system, which was less complicated and had a smaller state space. Finally, we would like to see, given more time, what the best possible agents we could achieve under our system would be.

References

[1] L. Lin K. Khosla and C. Qi. Artificial intelligence for pokémon showdown. 2015.

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

[3] S. Xu and C. Verbrugge. Heuristics for sleep and heal in combat, 2016.

[4] Eric Butler, Emina Torlak, and Zoran Popović. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, New York, NY, USA, 2017. Association for Computing Machinery.

Code

All code for the project can be found on GitHub at https://github.com/dem1995/pokemon-evolutionary-agents.