An AI For The Pokémon Trading Card Game

By Daniel Evert

MOTIVATION

The Pokémon Trading Card Game (TCG) is a computationally complex zero-sum *real-world* game with imperfect information. Similar *TCG's* such as *Magic the Gathering* have been recognized as one of the most computationally expensive games known today.

This project's focus was to explore the effects of *AI* generalization in an efficient manner given the computational complexity of the Pokémon *TCG*. A primary question was, given deck specifications, can an AI learn via mini-batch and also generalize well, specifically when conditioned on the composition of the deck?

A further motivation for this project was to determine how best to integrate the AI with the massive number of *Actions* (and sequences of these *Actions*) allowed per player per turn. Bette said, can there be *higher-order* strategems and heuristics at play that can efficiently run down the AI's available moves and act as reasonable as a human player would?

CONTACT

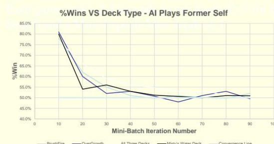
Daniel Evert
Stanford Univesrity
Email: Dje334@Stanford.edu
Phone: (512) – 680 - 2638

PROBLEM DEFINITION

Players assume the role of a Pokémon trainer and use their Pokémon to battle their opponents. Players play Pokémon to the field and attack their opponent's Pokémon.

A Pokémon that has sustained enough damage is knocked out, and the player who knocked it out draws a Prize card. There are six Prize cards, and the primary win condition is to draw all of them. Other ways to win are by knocking out all the Pokémon the opponent has on the field such that the opponent has none left, or if at the beginning of their opponent's turn there are no cards left to draw in the opponent's deck.

RESULTS



Deck Trained VS Predicted Deck

		Misty's Water	BrushFire	OverGrowth	All Three Decks
Trained On	Misty's Water Deck	50.3%			
	BrushFire	44.1%	51.2%		
	OverGrowth	39.8%	30.7%	49.2%	
	All Three Decks	48.0%	49.2%	50.2%	49.7%

Logistic Standardized Coefficients

1st Most Important Predictor	Material Difference
2nd Most Important Predictor	totalActiveAttackPower
3rd Most Important Predictor	numBenchedPokemon

Each game took ~2-3 seconds to run given $K=5$ for Beam Search and a Mini-Expecti-Max depth of 3.

REFERENCES

1. "Pokemon-tcg-sdk-python" https://github.com/PokemonTCG/pokemon-tcg-sdk-python
2. Churchill, A., Biderman, S. & Herrick, A. "Magic: The Gathering is Turing Complete" https://arxiv.org/pdf/1904.09828.pdf
3. "Texel's Tuning Method" https://www.chessprogramming.org/Texel%27s_Tuning_Method

DATASET

Used **pokemontcgsdk** to acquire the base set of the *150* original Pokémon Cards from the Python API

- *219* unique Attacks
- 145 unique Pokémon
- 45 unique Trainer Cards
- 9 unique Energy types

This project developed a Pokémon *TCG* game generator in Python

- Simulate Randomized Decks
- Decks chosen a-priori
- Records for training were the final end-states between turns (yielding 105 maximum records per game)
- Each Record included the extracted Input-variable between each turn
- 12,000 simulations yielded ~361k records by 50 columns per AI

CONCLUSIONS & FUTURE WORK

Conclusion – The *materialDifference* input was the most significant variable, which aligns well to the objective of the game

Conclusion – An AI can be efficiently trained, but decks with the most *latent* attacks, meaning damage is taken between turns without direct attacks, have worse generalization

Conclusion – Mini-Batch effectively converged given enough iterations at an *alpha = 0.1* for the Logit.

Future Work – Bring in hidden information and build out an Expecti-Max so that an AI can play a human "fairly" without knowledge of either the *AI's* next cards or the *Opponent's*.

Future Work – Expand to include a wider net of Pokemon and Trainer card coverage

Future Work – Attempt other *constraint satisfaction problems* outside of Beam search to optimize the sets of Actions

CHALLENGES

Challenge One: *N-Actions* per Turn

Unlike your typical *Mini-Max*, TCG allows Players to take a set of allowable *Actions* per turn. Further, every action can generate new Actions given a Player's turn, thus the set of all potential *Actions* must be discoverable via a Search Tree, where order in which the *Player* takes the action matters.

Challenge Two: Imperfect information

Dealing with expectations of what is in the *Opponent's* hand or assuming an unknown deck order adds *n-number* of cards of actions per player per recursion, thus increasing the *Action Space*.

Challenge Three: Large State-Space

- *Approximately 8.1* billion unique States of Pokemon can be∑in play at any given time
- *Approximately* (54 choose *m*) hand States
- *(60-U)!* unique ways to shuffle a deck, where *U* removes the effects of duplication

APPROACHES

All approaches can be shown in Figure 1. One way to imagine the *N-Actions* per turn dilemma was to pose the problem as optimizing a *constraint-satisfaction* problem and use the ***Beam Search*** algorithm. One supporting reason for this lays in the fact that one must traverse an *Action-Tree,* where each action can be seen as a weight with the objective to maximize one's state. By using ***Beam Search*** one could greedily take a subset of the *K-best* actions and return to the Mini-Max tree "reasonable" (aka local optimums) states. Effectively, this yielded *K-actions* per a *Player's* node; whereas the *Opponent* would be assumed to make *K-actions* themselves.

Once gameplay commenced, the AI's would face-off using their respective algorithm and generate data to update the weights using mini-batch gradient descent on a *Logistic Regression Model* and a *Neural Network* using Python' Sci-Kit's package.

The only *Expecti-Max* portion for the game came from whether a Pokemon successfully applied a condition (i.e. flipping of a coin to paralyze the opponent). All deck order information was known to the AI and the opponent.

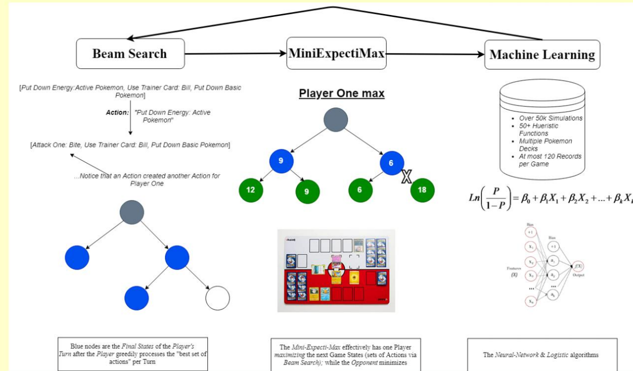


Figure 1

Poster Template Designed by Genigraphics ©2012 1.800.790.4001 www.genigraphics.com