CS9670B - Final Project Report

Mathias Babin

April 24, 2021

An up-to-date version of the code can be found at this Github repository.

1 Introduction

The goal of this project is to apply reinforcement learning to the domain of procedural content generation in games. Specifically, we are interested in generating content as a mechanism for introducing balance to an otherwise unbalanced system. This approach involves generating new game elements that alter the interactions between existing elements instead of modifying them directly.

For this task, we have decided to use Pokemon's turn-based RPG combat system as our environment of choice due to the inherent type, move, or stat advantages that some Pokemon have over others. Therefore, the goal of this project is to train a RL agent that when given a pair of any two Pokemon, generate an original third Pokemon that results in all three having a 33% winrate when battling one another.

2 Implementation

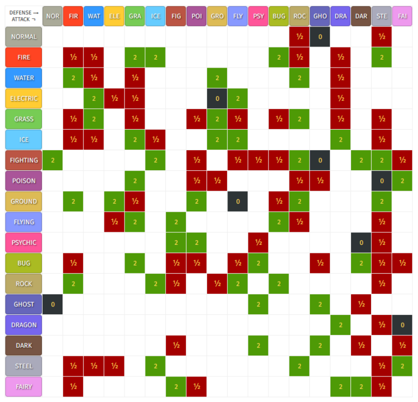
2.1 Environment

2.1.1 Pokemon Battle Simulation

Because no reinforcement learning environment exists for our specific task, we will implement our own. This involves writing a simple Pokemon battle simulator, as well as the class definition for a Pokemon and all of its attributes. For this work, we have reduced the number of Pokemon from the original 151 down to 27, and have a restricted move pool of size 19. In addition we have reduced the number of stats in the game down to Attack, Defense, and HP, have assume every move does physical damage, and does not apply any status effects. Finally, all Pokemon have the same stat distribution of 12 HP, 6 Attack, and 6 Defense. With these restrictions in mind, we have still implemented the game's original damage calculation:

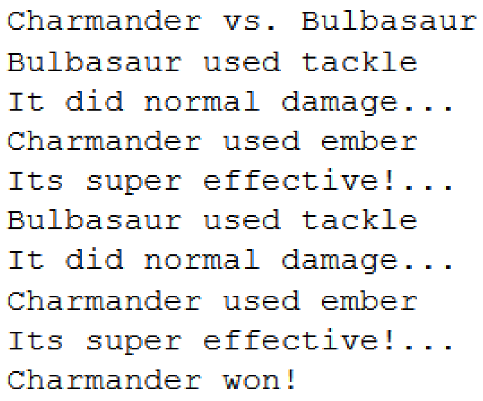
$$Damage = (((2 * Level/5 + 2) * Power * A/D)/50 + 2) * Modifier$$

where *modifier* is the *same-type attack bonus* (STAB), multiplied by type bonus, multiplied by a random float ranging from 0.85 to 1. We have also fully implemented the game's type advantage matrix as shown in figure 1a.



DEFENSE → ATTACK ↴	NOR	FIR	WAT	ELE	GRA	ICE	FIG	POI	GRO	FLY	PSY	BUG	ROC	GHO	DRA	DAR	STE	FAI
NORMAL													½	0			½	
FIRE		½	½		2	2						2	½		½		2	
WATER		2	½		½				2				2		½			
ELECTRIC			2	½	½				0	2					½			
GRASS		½	2		½			½	2	½		½	2		½		½	
ICE		½	½		2	½			2	2					2		½	
FIGHTING	2					2		½		½	½	½	2	0		2	2	½
POISON					2			½	½				½	½			0	2
GROUND		2		2	½			2		0		½	2				2	
FLYING				½	2		2					2	½				½	
PSYCHIC							2	2			½					0	½	
BUG		½			2		½	½		½	2			½		2	½	½
ROCK		2				2	½		½	2		2					½	
GHOST	0										2			2		½		
DRAGON															2		½	0
DARK							½				2			2		½		½
STEEL		½	½	½		2							2				½	2
FAIRY		½					2	½							2	2	½	

(a) The type advantage matrix.



```
Charmander vs. Bulbasaur
Bulbasaur used tackle
It did normal damage...
Charmander used ember
Its super effective!...
Bulbasaur used tackle
It did normal damage...
Charmander used ember
Its super effective!...
Charmander won!
```

(b) A battle between two Pokemon.

When two Pokemon battle, move selecting is performed by a simple algorithm which always chooses the move which maximizes damage. An example of our combat system is shown in figure 1b.

2.1.2 Reinforcement Learning Environment

Our Reinforcement Learning environment is designed to be solved in only a single step. First, the environment is reset and a pool 2 Pokemon which can either be randomly sampled from the entire Pokedex or supplied by the user. The resulting *advantage* of each Pokemon is returned to the player as their initial state–*advantage* is simply each Pokemons' winrate subtracted from a target percentage of matches each should win for the system to be considered balanced (for a pool with two Pokemon, the target percentage is 50%). Next, the agent supplies a vector of 6 indexes as an action: the first two positions are types, the last four are moves. It should be noted that every Pokemon must have at least one type, and one attack. This is solved by adding a *none* type and move for the second type slot and latter three move slots. Stepping in the environment using this action vector will simulate a battle with the two Pokemon in the current pool as well as the agent's newly designed Pokemon. The reward returned from taking an action is calculated as:

$$Reward = -1 * \sum_{i=0}^{k} |advantage_i| - 0.02 * t - 0.02 * m$$

where k is the number of Pokemon in the pool, t is the number of repeated types, and m is the number of repeated moves. With the environment formulated in this fashion, solutions can be found after only a single step. If we wish to make the task iterative we would simply need to include the agent-generated Pokemon produced at time $t-1$ in the current pool; however, for the experiments conducted in this report we look for solutions generated after only a single step.

2.2 Learning Algorithms

For this initial stage of development we train our agent using two different training strategies: 1) an Evolutionary Strategy 2) The actor-critic method A2C. Both algorithms train the same network architecture which accept Pokemon embeddings (more on this in the following subsection) as input and supports 6 action branches for the output: 2 for types and 4 for moves.

2.2.1 Evolutionary Strategy

Developed by OpenAI, Evolutionary Strategies are found to be a viable alternative to reinforcement learning. The algorithm at its core is very similar to Neuro-Evolutionary algorithms, which produce a set a randomized network parameters at each iteration and update towards those which perform best in the environment.

In our experience, this algorithm is a good starting point to test the validity of our environment's design and our network architecture's ability to successfully complete the task, howeve, for more complex tasks, training times can be long unless a parallelizable implementation is provided.

2.2.2 A2C

We would also like to test a Reinforcement Learning algorithm on this environment and choose A2C as the algorithm is simple to implement and converges fairly quickly (it was also helpful that we already had an implementation on hand from our previous assignment).

2.3 Pokemon Embedding

Input into our networks comes in the form of embeddings generated for each Pokemon. We obtain these embeddings using a Word2Vec model, more specifically a Continuous Bag of Words model (CBOW). While normaly applied to text-based analysis, CBOW learns the semantic meaning of words based on their context. In our case, we first build a corpus of battle data from our reduced Pokedex of 27 Pokemon. Here, our model receives the names of two Pokemon as context, and the name of the winner as a target. The goal is to train the model to develop a certain level of semantic understanding about each target Pokemon based on which battles it wins and loses.

Ultimately we train our CBOW model using a dataset containing over 350,000 simulated battles, and produce an embedding vector of length 32 for each of the 27 Pokemon. When training our ES and A2C agents we provide them with the embeddings of both Pokemon we are looking to generate a balanced solution for. As an example, if we are looking to balance Charmander and Bulbasaur, we first find both of their embeddings and concatenate them into a single input of size 64 to either our ES or A2C agent, and receive the description of new Pokemon of length 6 (type0, type1, move0, move1, move2, move3) which should result in a 33% winrate for each of the three Pokemon.

3 Project Overview

This section provides an outline of this projects data and python files.

3.1 Data

Any files prefixed with *database* were obtained from an online Kaggle repository. Files beginning with *battle_meta* were generated for training our Word2Vec model, and all other data set were hand created to simplify the larger *database* files:

- **battle_meta_small_advantage.csv**: A data set generated to train our CBOW model using the advantage scores of Pokemon as the target.
- **battle_meta_small_winner.csv**: A data set generated to train our CBOW model using the winners of each battle as the target.
- **database_moves.csv**: A database containing all the current generation's moves.
- **database_movesets.csv**: A database containing all the current generation's movesets for all Pokemon.
- **database_pokemon_151.csv**: A database containing the original 151 Pokemon's stats.
- **moves_small.csv**: A database containing 20 moves for our simplified training scenario.
- **pokemon_small.csv**: A database containing 27 Pokemon, their stats, and moves for our simplified training scenario.
- **pokemon_small_embeddings.csv**: A database containing embeddings for all 27 Pokemon in our reduced Pokedex, generated using our CBOW model.
- **type-chart.csv**: The type-advantage matrix used for calculating damage during battle.

3.2 Python Scripts and Notebooks

We encourage that you explore each of these notebooks to gain a more thorough understanding of the methods used during this project. The following will provide a brief description of each:

- **Balance_A2C-10000.ipynb**: A notebook containing the training of our A2C agent for 10000 episodes using randomly generated battle pools of Pokemon.
- **Balance_A2C-1500.ipynb**: A notebook containing the training of our A2C agent for 1500 episodes using randomly generated battle pools of Pokemon.
- **Balance_A2C-Non-Random.ipynb**: A notebook containing the training of our A2C agent for 500 episodes using a predetermined battle pool of Pokemon.
- **Balance_ES.ipynb**: A notebook containing the training of our ES agent for 120 episodes using a predetermined battle pool of Pokemon.
- **Data_Advantage.ipynb**: A notebook used to generate **battle_meta_small_advantage.csv**.
- **Data_Winner.ipynb**: A notebook used to generate **battle_meta_small_winner.csv**.
- **Pokemon2Vec.ipynb**: A notebook containing the training of our CBOW model.
- **A2C_agent.py**: A python file containing the implementation of A2C agent.
- **es_agent.py**: A python file containing the implementation of ES agent.
- **model.py**: A python file containing the neural networks for both our A2C and ES agents.
- **pokemon.py**: A python file containing the class definitions for moves, Pokemon, and our battle simulator.
- **pokemon_env.py**: A python file containing our RL training environment.

4 Results

This sections outlines the training results of our CBOW, ES, and A2C models. When training our ES and A2C agents, we first began with a predetermined pool of 4 matchups as a toy-example of the problem:

- Charmander vs. Bulbasaur
- Pikachu vs. Squirtle

- Lapras vs. Pidgey
- Ekans vs. Diglett

After we succeeded with this simplified scenario, we attempted to train our A2C model with randomized pools of size 4 sampled from our reduced Pokedex.

4.1 CBOW Performance

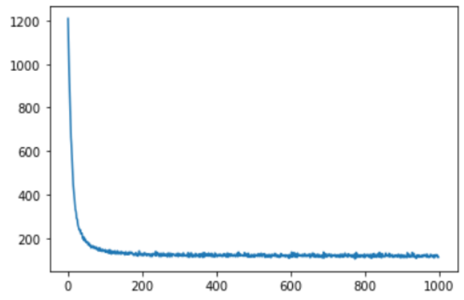


Figure 2: Loss (Y) over Time (X) of our CBOW model.

As shown in figure 2, our CBOW model's loss does appear to decrease until the 200th epoch of training. We surmise that its loss does not drop to 0 due to fact that some battles between the same pairs of Pokemon can have different winners, especially when the Pokemon are evenly matched such as Rattata and Meowth, where the winner is based purely on which goes first. It should be noted that when testing the cosine similarities of identical Pokemon such as Squirtle and Poliwag, similar vectors were not always produced. After experimenting with the size of the embedding, varying results were obtained, leaving this a topic to be addressed in future works.

4.2 Evolutionary Strategy Performance

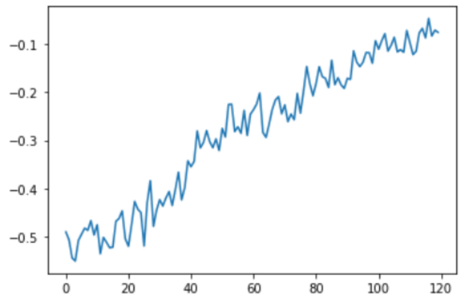


Figure 3: Rewards (Y) over Time (X) for our ES agent.

Figure 3 the average reward over time for 120 episodes of training for our ES agent. On the predetermined pool, the algorithm converges to the optimal solution (a reward of 0) after 120 episodes.

4.3 A2C Performance

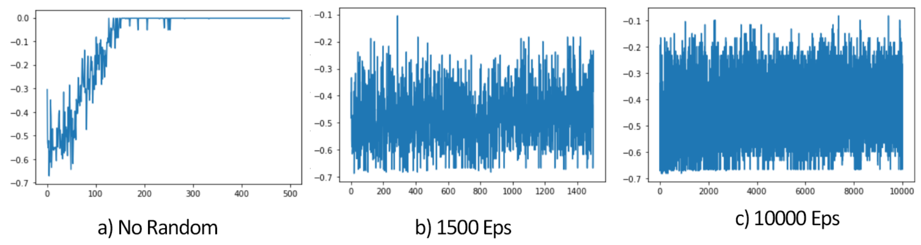


Figure 4: Rewards (Y) over Time (X) for our A2C agents.

Figure 4 shows the reward received by our A2C agent. In A), the agent trained using the same predetermined pool as the ES agent, and converged to the optimal solution after approximately 150 episodes. In B) we attempted to training our A2C agent for 1500 using the random pool scenario, and similarly C) for 10000 episodes.

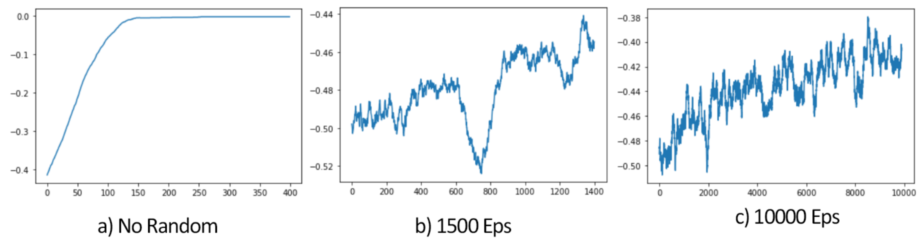


Figure 5: 100 Episode Avg. (Y) over Time (X) for our A2C agents.

Figure 5 provides a 100 episode average for our A2C agents, which does show a positive trend in all 3 cases; however, it is clear that a solution was found only in the simplified scenario. After looking at the Pokemon generated by the Agent it appears that the network is only capable of generating a single Pokemon with very little variation, meaning that it converged to a single solution that performs well on average instead of multiple solutions that perform optimally based on the input to the network. A potential explanation for this maybe that the certain combinations of Pokemon were appearing less frequently than others based on random sampling. A potential solution to this problem would be to run through all of our 350 possible combinations of Pokemon every episode, or at least sample more than 4 at a time. If it is found that the solution still converges down to a single Pokemon, an interesting experiment would be to pre-train the model with human authored examples in a supervised manor. This might *prime* the weights to generate different solutions that could be expanded through further training via reinforcement learning.

5 Future Works

Future works for this project will include:

- expanding the simulation to support all 151 original Pokemon and their full movesets, support for a more sophisticated battle simulation including status effects.
- Training the model to produce a more diverse range of solutions.
- Training the model on pools greater than 2 Pokemon at a time.
- Adapt the output of the model to support the generation of more than one Pokemon at a time.
- Retrain embeddings such that their cosine similarity accurately reflects the similarities/difference between two Pokemon.