**THE VERY BEST LIKE NO ONE EVER WAS: PLAYING POKÉMON WITH DEEP REINFORCEMENT LEARNING**

**PROGRESS REPORT II:**

Kenan Rustamov Jake Baumbaugh

Sean Steinle Gordon Lu

**WHAT WE’VE ACCOMPLISHED**

First and foremost, we have a basic reinforcement learning model working. This model is inspired by the PokeEnv tutorial for reinforcement models[[1]](#footnote-0), and much of the code is still similar. To train this model, we have also created a random agent, Randy, and a max damage agent, Max. We are using this first model as a sort of “default”, from which we will make changes to improve performance. We have already made significant changes to the model, most notably in the state (or battle) embeddings, and model analysis and logging, but also with small changes to a few hyperparameters and reward values.

To run the model with PokeEnv, we first had to set up the server. This involved cloning PokemonShowdown and using that to run a local implementation with NodeJS. Through this server we could conduct battles with random pokemon and get statistics and variables from the battle. With a local server, we avoid the hard limits set by the PokemonShowdown remote server such as move rate and number of concurrent battles. The code itself is written locally and managed through Git, a version-control software. We also develop a python environment with a strictly versioned requirements document for our pip installs. This ensures that all the code runs on the same environment, and that there are no issues with seperate versions or interference from other installs.

After we set up the server, the random agent was just an import away--PokeEnv actually has a random agent in its player module[[2]](#footnote-1). This was not the case with the max damage agent, but it was still a simple implementation. In fact, the pseudocode fits in just a line: for all of the moves a pokemon has, return the move with the highest base damage[[3]](#footnote-2). With these agents completed, we could now move onto training the reinforcement model. For more on the results from our initial run, see the fourth section. After successfully running the default model, we started making changes to the model’s structure.

Our model is implemented with Deep Q-Learning using an Epsilon greedy policy. This ensures that our model balances between exploration and exploitation when choosing moves. On top of this, we have a linear annealing policy wrapper. This wrapper has a max value and min value which specifies the starting and ending points for epsilon in the epsilon greedy Q policy. This wrapper starts the epsilon at the max value and linearly decreases the value until it reaches the min value while it is training. This attempts to make our model choose randomly at the beginning to try to discover moves that lead to high rewards, and then use learned moves more as it trains. The model has three dense layers of 1024, 1024, and 512 nodes respectively and a final layer. The hidden layers all use ReLU activations while the final layer uses softmax. ReLU is one of the most commonly used activation functions and we figured it would be fine for this task as it is for most deep learning tasks. For the optimization function we decided to use Adam which uses an exponentially decaying average of previous gradients. This allows us to quickly arrive at a minimum and avoid oscillations common with stochastic gradient descent. Sequential memory was also implemented to increase the chances of the model using a high reward action. It does this by storing the state and rewards of previous actions and using it in decision making in the future.

One place we’ve made a great deal of changes is the input vector for the neural network. The default input space discussed in the documentation of PokeEnv included only a couple details about each available move of the active Pokemon, as well as a count of the number of remaining Pokemon on each team. While this is absolutely useful information for the network to know, the complexity of Pokemon requires much more information to truly compete at a high level. Most notably, it is paramount to have knowledge of the other Pokemon on your team to decide which is best to battle the opponent’s active Pokemon. Our input as it stands now includes details about the statistics and type of each Pokemon on our team, along with their available moves and various statistics about them. While the available information about the opponent’s Pokemon is limited, we do keep track of what is known so that the model can maximize its strategy. Finally, the input space includes some knowledge about the battlefield itself; right now, it contains a one-hot vector describing the weather in the battle.

The provided simple mapping trains more quickly to reach a point of reasonable success, but these success rates were not at the level that we are searching for, so increased complexity is required, but takes much longer to train as a result. This is a challenge we intend to face and try to improve, which we discuss more in the third section.

Another significant feature we added to the model was improved logging. When the default model trains, there are a few different output options to the terminal, however none of these options produced a list of the reward value over time during training. We found this to be an important data because simply eyeballing the reward value going up and down through training is not very helpful in understanding if/how the model is improving. Instead, our logger now allows us to visualize our model’s reward over time with matplotlib. Even more good news, this work can easily be built upon to extract the number of moves per battle, the loss per step, and much more. You can read more about how this was accomplished in section two.

**ROADBLOCKS**

Through the development process, we hit a number of bumps, though luckily nothing too severe. The poke-env library we are using has been an enormous help in bringing our DQN bot to life, but it has plenty of small issues. We ran into a significant issue while creating our input vector for the network: occasionally the input size was incorrect. As it turns out, sometimes getting the moveset of each Pokemon would return five possible moves, when there are only four valid moves at any point in a battle. After tracking down this issue, we decided to omit the fifth move, as the first four are sufficient to convey the Pokemon’s moveset. One Pokemon, Ditto, only has one move, so we needed to ensure that three additional empty moves would be written into the vector to account for this particular case. This introduces a great deal of empty noise into the initial vector space for that particular pokemon, but writing zeroes into those fields should neutralize any particular effects.

Another task that gave us issues was extracting values during training. To visualize the change in reward over time, we needed a way to capture the reward value at each step of training. This proved rather difficult because Keras-RL (our training library) abstracts all training away to the model.fit() method. Thus, there wasn’t an obvious way to extract step-wise information. Upon further research, we found the Callback class, which allows for training time customization--though there were very few tutorials on the topic. The results of this endeavor are discussed in section four and are stored in *callbacks.py*.

**DEVELOPMENT AGENDA**

Despite the leaps through the roadblocks encountered thus far, there is still room for improvement. As seen from the results from the evaluation section, comparing the performance of our model to the max damage model, our model clearly falls short. Aside from what was discussed to improve win rate, such as accounting for more events in a battle that can influence the player’s optimal move, one overlooked factor has been the choice of hyperparameters. For our neural network, the hyperparameters of interest are: the number of hidden layers, the number of nodes per layer, the optimization algorithm, memory limit, reward, learning rate, and the number of training steps. In lecture, the optimal hyperparameters can be selected through different techniques such as through Grid Search, Random Search,. What we seek to use is to use three search strategies to find the optimal hyperparameters. The first search strategy we seek to use is Random Search, which will be used to establish a baseline for the other search methods. Another search strategy to be used is Bayesian Sequential-Based Optimization, in which a surrogate model is built to attempt to predict the optimal hyperparameter. At each new iteration, the surrogate we will become more and more confident about which new guess can lead to improvement. We will use a Gaussian Process as the surrogate, as not only produce the prediction as a value, but produce a range of uncertainty in the form of mean and variance4. However, by far the most popular technique used in Machine Learning to find the optimal hyperparameters is cross-validation. A Popular choice for the number of folds for cross-validation are five and ten. With this in mind, we plan on running cross-validation with different sets of hyperparameters, and use folds of five, ten, and fifteen. In particular, we will use holdout validation, in which our data set will be partitioned into a training, test and validation set. We will utilize the validation set to tune different sets of hyperparameters and run cross-validation using our different sets of folds and evaluate the error. One each of the respective search techniques are performed, we will compare the accuracies and errors of the model using said hyperparameters. Based on the search technique that yields the hyperparameters that produce the best error and accuracy on the model, we will use this technique for future data sets. One thing that is crucial to note is that on an update to the model itself, the search techniques will need to be run again to find the optimal hyperparameters again, and a different search technique may be selected than from a prior model.

Another suspect that may be causing our model’s growth to stunt is the complex input space. Naturally, some values will have much stronger effects on the outcome, and many values have important relations between each other which are difficult to learn when the input space is so large. One of the most important aspects of a Pokemon battle is the “type” of a Pokemon or move. These types, which include fire, water, poison, electric, and many more, have advantages over other types, making some moves especially powerful against the opponent’s Pokemon. A more basic model which simplifies these types by calculating the advantage of each move is able to learn much more quickly how to utilize this information. Our model as it currently stands, on the other hand, would need to learn all of the combinations of type advantages, even multiple times between the many one- (or two-) hot vectors we use for types. One of our big steps moving forward will be to simplify our input vector, streamlining values where possible, in hopes of eventually introducing a few more details about each battle. Pokemon is a complex game after all, with many different variables affecting the game state and the player’s optimal move, so some extra details such as status effects and battle terrain will be helpful. As our model grows in complexity, we will consistently update hyperparameters to be consistent as well as to demonstrate changes in accuracy and error. [[4]](#footnote-3)

**EVALUATION METHODS**

In evaluating our model, we have decided that it is important to analyze our model both objectively and subjectively. Our objective measures are fairly standard across the machine learning world, whereas our subjective standards are inspired by concepts like adversarial learning.

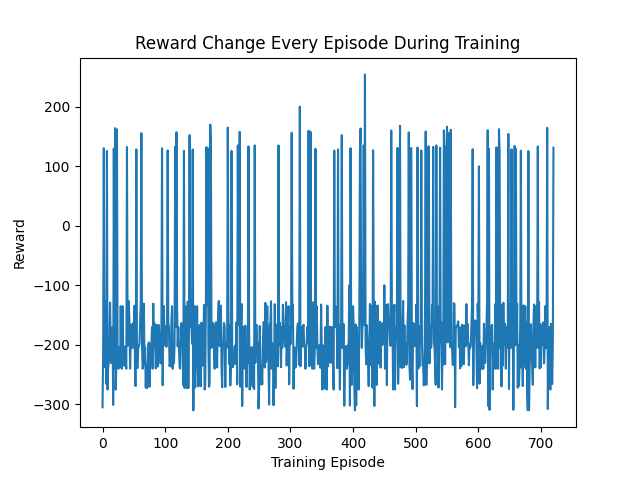
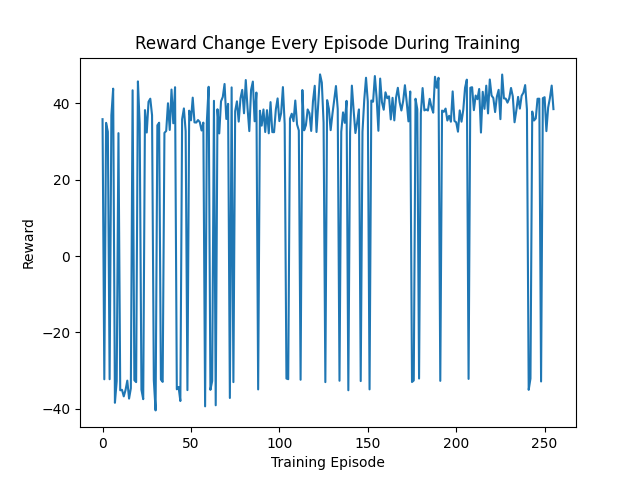
Our objective measures differ slightly from the metrics we often use in class, but this is to be expected of our reinforcement learning algorithm. For example, the total loss is probably a more important statistic in traditional algorithms than in ours. Instead, our two main objective measures are reward, which represents something similar to loss, as well as win rate. Reward is essentially a customized loss function that should translate to winning battles, in the same way that low loss for a computer vision model should translate to more pictures classified correctly.

As such, we’ve made it a high priority to visualize and understand this loss for each model we train (see sections 1 and 2). Additionally, we’ll be experimenting with different rewards values in order to stimulate new battle strategies for our model. For example, we may try to raise the reward value for inflicting a negative status higher than usual in order to promote drawn-out, status-heavy battles.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Figure 1 | Trained Against | Wins vs. Randy | Wins vs. Max | Learning Rate | Training Time | Training Steps |
| Default | Randy | 91/100 | 72/100 | 0.00025 | ~15 minutes | 10,000 |
| Our Model | Randy | 57/100 | 11/100 | 0.001 | 39 minutes | 25,000 |
| Our Model | Max | 44/100 | 11/100 | 0.001 | 42 minutes | 25,000 |

In terms of performance, our changes to the model really flopped. The model run from the source code mentioned in section one performs significantly better and even trains faster. Our changes to the battle embeddings are the most likely cause of this dropoff, but smaller changes to the architecture and reward values may have also influenced our poor performance as well.

On the next page, you can see two graphs of the models over time. The first graph is of the default model, which seems to understand the game by the 30th battle or so. From then on, the model seems to be consistently winning games--with its losses being those big, momentary dropoffs. On the other hand, our model trained against the max damage agent cannot even muster a zero reward on almost three times the number of battles. These graphs make it pretty clear that our model isn’t learning much.



No matter how objectively successful our model is, we cannot guarantee it is a smart model without some subjective tests. Our motivation for these tests is that we do not want our model to emulate a simple algorithm like max damage. Instead, we want the model to learn a successful and creative strategy. To test this, we have three checks. First, a group member will watch our model play a certain number of games, an equal number of wins and losses. Second, a group member will play against the model (this is inspired by adversarial learning/testing). Finally, if time and servers permit, we will test our model online against human players. These tests are all in an effort to better understand our model qualitatively, so that there are not obvious weaknesses for our model.

We have not been able to perform many subjective tests so far, but from what we have seen, our model is essentially random. This makes sense, as our model has almost a 50% win rate against Randy. The default model, on the other hand, uses type advantage as its main weapon. This is something that neither the random nor the max damage agents know about, which provides a huge competitive advantage to the model.

**APPENDIX**

It’s worth mentioning the structure of our codebase so far. Currently we have three unique (non-git) files:

1. *rl\_agent.py* - This is our largest file by far, and it contains all of the code necessary for building and training our model. It is a priority for us to downsize this file into smaller components. As of now, it sits at around 250 lines.
2. *max\_agent.py* - This file is the source code for the max damage agent. It’s around 40 lines, but some of that is a tester main method.
3. *callbacks.py* - This file contains a custom callback class that is used to extract information during training from the mode.fit() call. We will write any other callback classes in the future into this file.

Note that our current repository has more files, but they will not be present in our final repo. You should be able to view our Github repository here: [KenanRustamov/TheVeryBestDeepLearningBot (github.com)](https://github.com/KenanRustamov/TheVeryBestDeepLearningBot)

1. [Reinforcement learning with the OpenAI Gym wrapper — Poke-env documentation (poke-env.readthedocs.io)](https://poke-env.readthedocs.io/en/stable/rl_with_open_ai_gym_wrapper.html#rl-with-open-ai-gym-wrapper) [↑](#footnote-ref-0)
2. [Cross evaluating random players — Poke-env documentation (poke-env.readthedocs.io)](https://poke-env.readthedocs.io/en/stable/cross_evaluate_random_players.html#cross-evaluate-random-players) [↑](#footnote-ref-1)
3. [Creating a simple max damage player — Poke-env documentation (poke-env.readthedocs.io)](https://poke-env.readthedocs.io/en/stable/max_damage_player.html#max-damage-player) [↑](#footnote-ref-2)
4. [Practical guide to hyperparameters optimization for deep learning models.](https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/)  [↑](#footnote-ref-3)