Predicting Pokemon Primary Types

Alex Litchfield and Ethan Wong

Rensselaer Polytechnic Institute

COGS 4210: Cognitive Modeling

Dr. Stefan Radev

April 23, 2025

Introduction

Pokemon is a multimedia franchise created by Nintendo, Game Freak, and Creatures. Starting out as a single set of video game releases, the franchise has now released countless other titles as well as trading cards, anime, movies, toys, and more. At its core, Pokemon is a game about catching different creatures, or Pokemon, and training them to create a team that will progress them through the game and ultimately defeat the champion's team of Pokemon. Each Pokemon species is different from the rest, having their own base stats distribution and elemental types. Pokemon stats fall under HP (hit points), Attack, Defense, Special Attack, Special Defense, and Speed. Each species is either monotype (has 1 elemental type) or dual type (has 2 elemental types), pulling from the 18 different possible elemental types. (**Figure 1**). When a Pokemon is monotype, its singular type is referred to as its primary elemental type. For dual type Pokemon, this refers to the elemental type that is listed first in the game's data.

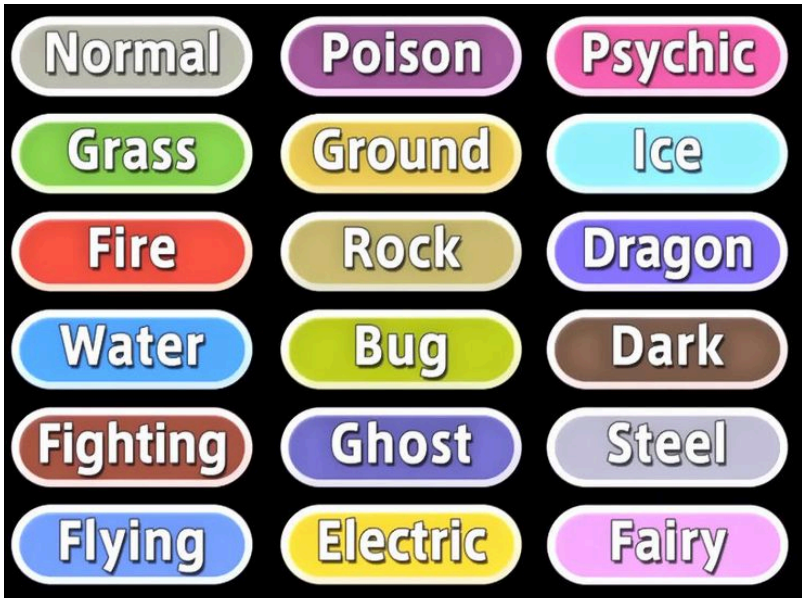


Figure 1. List of Different Pokemon Elemental Types

The "Pokedex" dataset, aptly named after the in-game dictionary of Pokemon, provides a collection of data on each of the 1025 different Pokemon featured in the franchise to date. The goal of this project is to create a model that can accurately predict a Pokemon's primary type using the base stats of each species using the Pokedex dataset and a neural network.

We will be utilizing a neural network. It is a machine learning model that is based on the human brain. It is made up of layers of interconnected "neurons" that process data to make predictions. In our case we will be analyzing the Pokémon database. This database provides specific statistics on the provided Pokémon. These stats include, hp, attack, defense, special attack, special defense, and speed. Pokémon are also given a designated primary type such as normal, fire, water, electric, grass, ice, fighting, etc. Given this dataset we want to see if a given neural network architecture can determine the designated primary type of Pokémon given its stats. The research question for the project is "How accurately can a neural network machine learning model, using the Pokedex dataset, predict a Pokemon's primary typing based on its base stats?".

Methods

Software and Libraries

The project was created using a Python Jupyter notebook and utilized many Python libraries. Numpy and Pandas were used for data loading and organization, Scikit-Learn and Imbalance-Learn for data preprocessing and model evaluation, Matplotlib and Seaborn to show visuals for the results, and Tensorflow/Keras for building the neural network.

Data Preprocessing

The initial dataset included 12 data columns for name, height, weight, hp, attack, defense, special attack, special defense, speed, elemental type, evolution set, and info (pokedex entry). Using pandas, some of these columns were removed since the data was irrelevant to the model, only keeping the names, base stats of the Pokemon (hp, attack, defense, special attack, special defense, speed), and elemental type. Additionally, the elemental type column was replaced with that of the primary type column, taking only the first entry of up to 2 types within the original elemental type column as the new primary type. Using Sklearn for the LabelEncoder the categorical labels for each type were converted into numerical values. From there Matplotlib was used to display the original distribution of the primary types shown for the 1025 Pokemon (**Figure 2**). Since it was clear that there was a great imbalance in the distribution of primary types among Pokemon, 2 more distributions were created. The second distribution built on top of the first utilized Imbalanced-Learn for RandomOverSampler (**Figure 3**). The third distribution built on top of the second then utilized Scikit-Learn for Standard Scalar (**Figure 4**). The three distributions were created to find the optimal primary type distribution to be used moving forward. The first distribution using only a Label Encoder, the second using a Label Encoder and over sampling, and finally the third using a Label Encoder, over sampling, and normalization. Without these modifications, the data likely would have led the model to only be able to accurately predict the most frequent types. Later in the report the a version of the model will be shown using no oversampling or normalization, running into this exact issue with Water, Normal, and Grass, the three most common types.

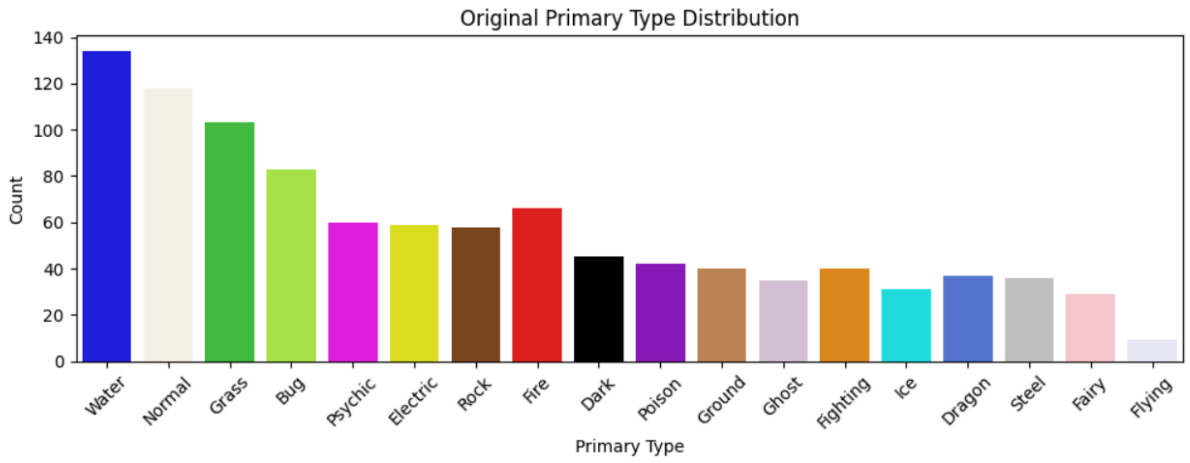


Figure 2. Original Distribution of Primary Elemental Types over 1025 Pokemon

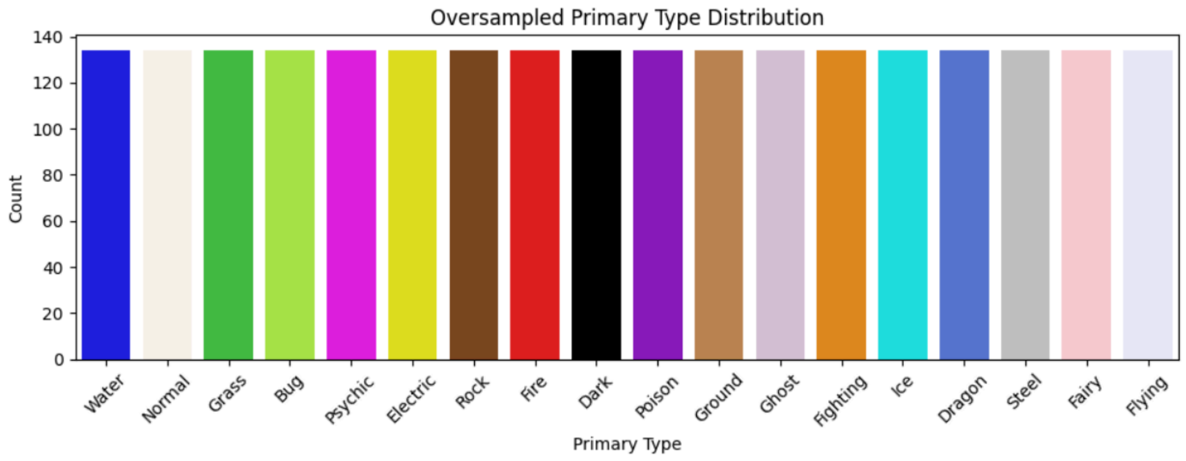


Figure 3. Distribution of Primary Elemental Types using Oversampling

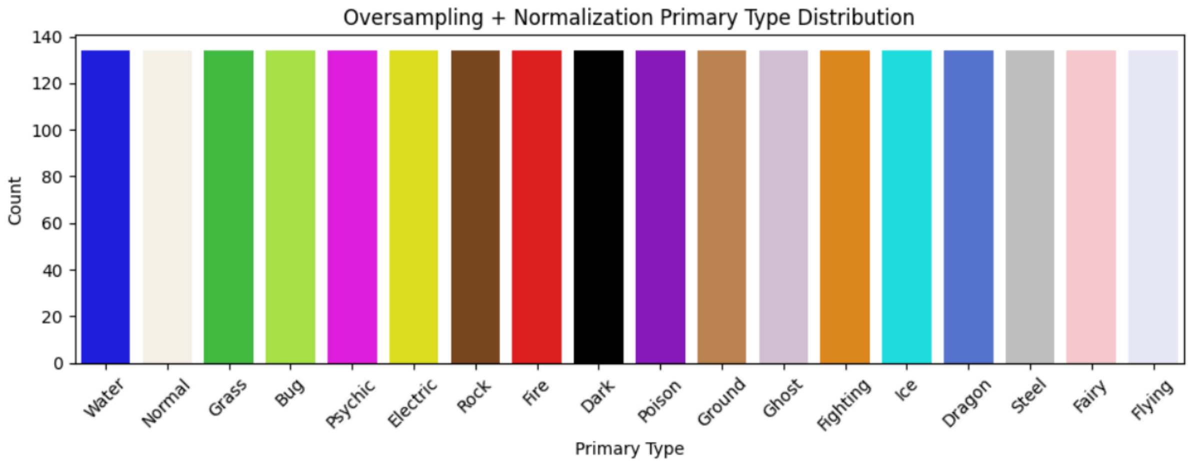


Figure 4. Distribution of Primary Elemental Types using Oversampling and Normalization

Model Development

The first iteration of the neural network model had an architecture involving 4 hidden layers, each with 256 neurons and “relu” activation”. There was also an input layer designed to take in the correct number of features (the base stats) and an output layer with 18 neurons for each of the 18 elemental type classes with a “softmax” activation. The current iteration of the neural network is nearly identical to the first except the number of hidden layers was reduced to 3 and the number of neurons in each was changed to 128. Additionally, dropout rates of 0.2 for the first 2 hidden layers and 0.3 for the last hidden layer were added to prevent overfitting during the model’s training process. The “sparse_categorical_crossentropy” loss function was used since this was a multi label classification problem where the labels were converted into integers. For both the first and second iteration of the model there are 5 versions, each using a different optimizer to show the effects it could have on the results and what would be the best choice for the task. The different optimizers were Stochastic Gradient Descent (SGD), SGD with momentum, Adam, Adagrad, and Root Mean Squared Propagation (RMSprop). Each optimizer used their default learning rate, which is 0.01 for SGD, SGD with momentum, and Adagrad. The default learning rate for Adam and RMSprop is 0.001. SGD momentum also used a momentum value of 0.9

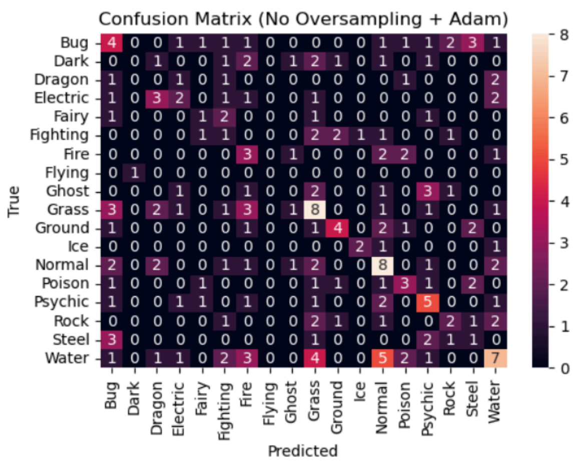
Model Evaluation

The dataset was split into 3 sets for training, test and validation. The training data is 80% of the dataset, the testing data is 20% of the dataset, and the validation data is 10% of the training data. The model trains with a batch size of 64, updating its weights every 64 samples. Each model version was then trained on 500 epochs by default unless otherwise specified and the performance was evaluated through accuracy and loss trajectory on the training and validation sets. Each model version also displays a confusion matrix and a classification report showing accuracy, precision, recall, and F1 score.

Results (No Oversampling Vs. Oversampling with Adam Optimizer)

In this report, only the second iteration of the neural network will be discussed due to the large amount of content and the fact that each version of the model performed better on iteration 2 than iteration 1. Data on iteration 1 can be found in the modelV1.ipynb Jupyter notebook in the github repository for this project, and it is encouraged that readers visit this to see the change between the 2 iterations. Data on iteration 2 can be found in the modelV2.ipynb Jupyter notebook in the same location.

Three versions of the model were run using the Adam optimizer to show the effects that oversampling and normalization had on the results. The first did not use oversampling and performed rather poorly with an accuracy of about 25%. The confusion matrix indicates that the model was only able to handle predictions well for 3 of the 18 types. (**Figure 5**). In addition, the validation loss and training loss did not converge, indicating that the model was overfitting (**Figure 6**). The expectations for this model given its optimizer were that the training loss would decline quickly and smoothly from the very first epoch. It was also expected that the validation and training accuracy curves would climb rapidly and plateau at higher levels, maintaining only a small gap between them. This represents Adam's ability to balance fast convergence, stability, and robust generalization across both common and rare types. However, the original data made it impossible for Adam optimizer to find any patterns in the data.



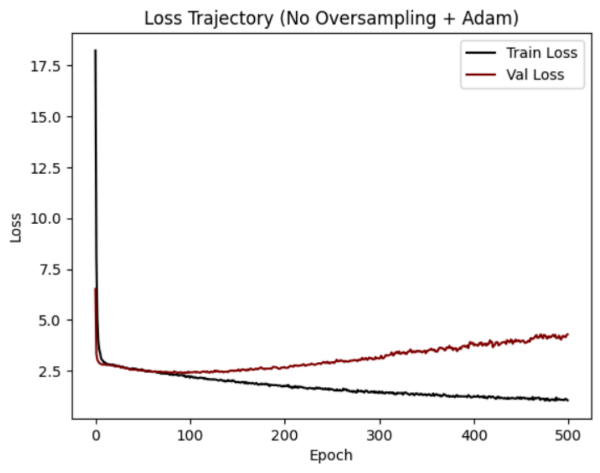
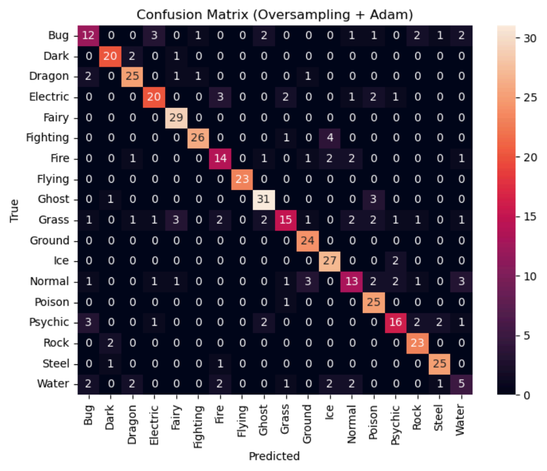


Figure 5. Confusion Matrix for No Oversampling + Adam Optimizer

Figure 6. Training Loss Trajectory Vs. Validation Loss Trajectory for No Oversampling + Adam Optimizer

The second version used oversampling but did not normalize the data. The overall accuracy of the model improved significantly to about 75%, and was generally able to predict 10 of 18 types this time (**Figure 7**). The validation and training loss did not converge, but were much closer than before (**Figure 8).**



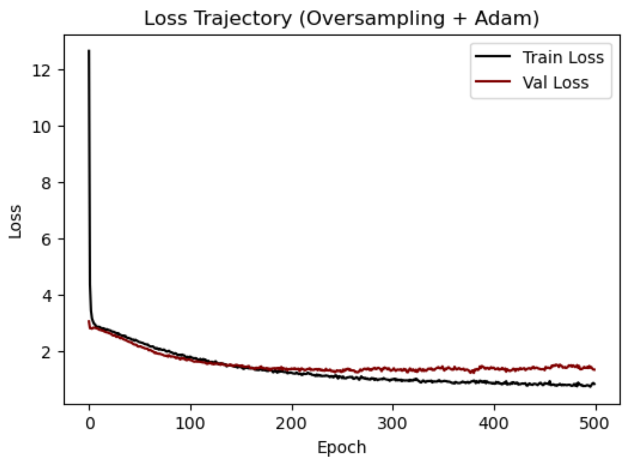
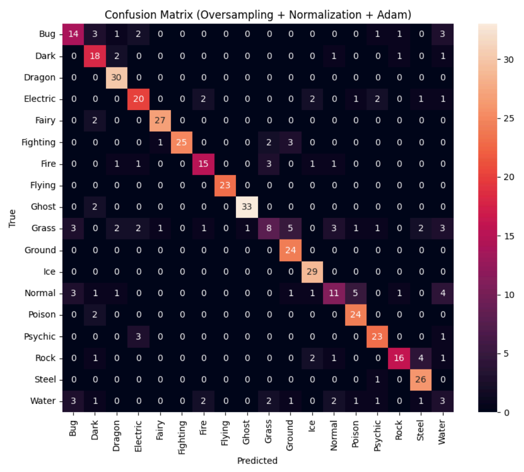


Figure 7. Confusion Matrix for Oversampling + Adam Optimizer

Figure 8. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Adam Optimizer

The third version used oversampling normalization on the data, boasting an accuracy of 73%, with once again only 10 of the primary types being predicted accurately (**Figure 9**). With Adam, the optimizer combines momentum (first-moment estimates) and adaptive learning rates (second-moment estimates), bias-corrects both, and uses them to compute per-parameter step sizes. The loss validation and loss trajectories also did not converge, once again moving slightly farther away from each other than in the previous model (**Figure 10**). Despite this slight hiccup with normalization, it was deemed necessary to include normalization in future models to ensure that the loss trajectories would converge. It is also worth noting that the validation accuracy was still lower than the training accuracy, indicating that the model was memorizing the data and not learning it. The Adam optimizer combines momentum (first-moment estimates) and adaptive learning rates (second-moment estimates), bias corrects both, and uses them to compute per-parameter step sizes. Every model moving forward would use oversampling and normalization on the data since it was believed it would improve performance metrics, with each version only differing by the optimizer used in the model.



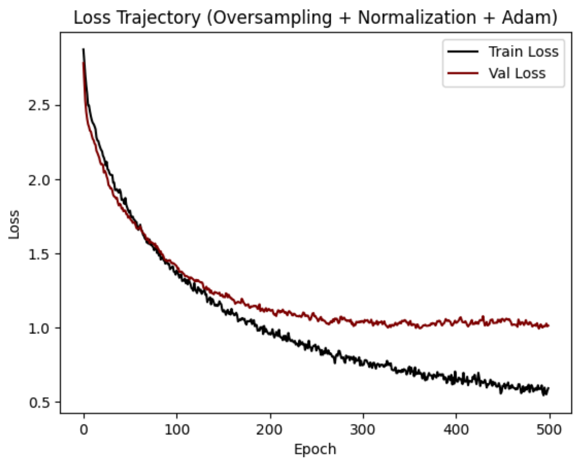


Figure 9. Confusion Matrix for Oversampling + Normalization + Adam Optimizer

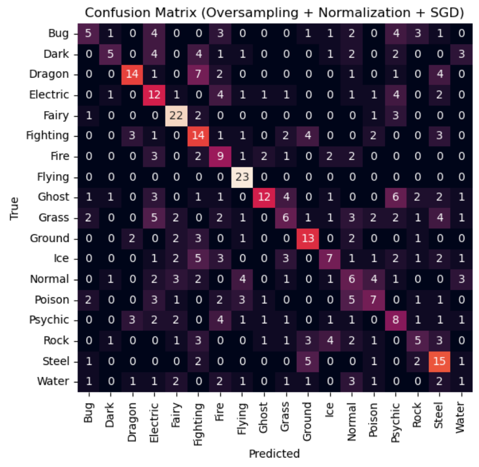
Figure 10. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + Adam Optimizer

Results

Results (Oversampling and Normalization with Different Optimizers)

Once it was determined that each model henceforth would use oversampling normalization on its data, the next step was to use different optimizers for each model to determine which version could predict the primary types most accurately. The optimizers used were Adam, SGD, SGD with momentum, Adagrad, and RMSprop. Since the current model configuration was already tested with Adam in the previous model, it will not be shown again. The next 4 versions of the model will use the remaining optimizers.

The fourth version of the model used SGD as its optimizer. This optimizer updates the model's weights by computing the gradient of the loss over small, randomly sampled mini-batches and applying a fixed global learning rate at each step. The expectation was that the training loss would decrease steadily but more slowly than with adaptive optimizers. The validation loss would mirror this but plateau sooner and experience more noise. Training accuracy should climb gradually with validation accuracy improving at a slower pace and plateau sooner if overfitting occurs.This model only had an accuracy of about 37% and had a 10% difference between the validation and training accuracies. Despite this gap, the loss trajectories very nearly converged, showing promise for the SGD optimizer (**Figure 11**). Neither the accuracies or the loss trajectories plateaued, suggesting that should the model be run on a greater number of epochs, the data would converge and improve without overfitting. The confusion matrix only around 3 types were accurately predicted (**Figure 12**).



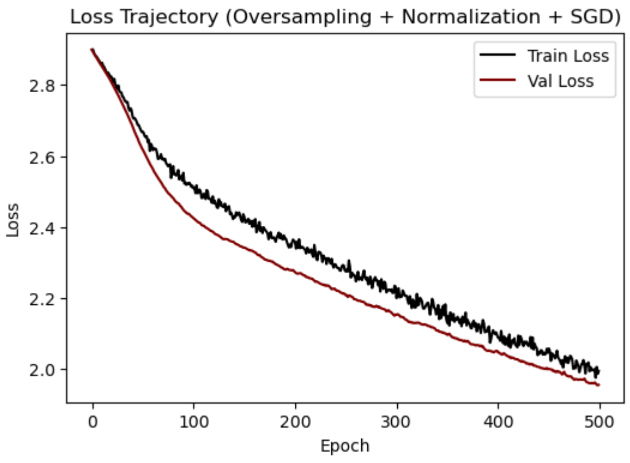
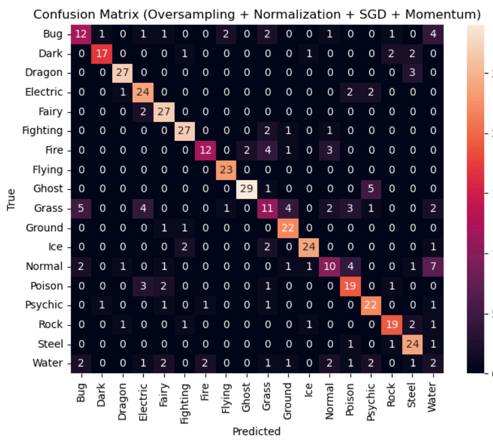


Figure 11. Confusion Matrix for Oversampling + Normalization + SGD Optimizer

Figure 12. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + SGD Optimizer

The fifth version of the model used the optimizer SGD with momentum, where the momentum was 0.9. This optimizer augments the standard weight update by maintaining a velocity vector. Each step is a combination of the current gradient and a fraction of the previous update. This damps oscillations and accelerates progress along consistent descent directions. The expectations for this optimizer were that the training loss would drop faster and more smoothly than SGD. The validation loss was also expected to fall quickly and plateau at a lower value, indicating improved generalization. Training accuracy would ramp up earlier and more smoothly, while validation accuracy would climb sooner and peak at a high value. This model seemingly performed better than that of SGD, with converging accuracies of about 73% and lower loss trajectories (**Figure 14**). The confusion matrix also showed that 7 of the 18 types were now accurately being predicted (**Figure 13**). However, both graphs began to plateau and the loss trajectories had not yet converged. While SGD with momentum managed to have better accuracy, SGD still seemed to be more promising if provided more epochs to run over, since SGD with momentum was already overfitting.



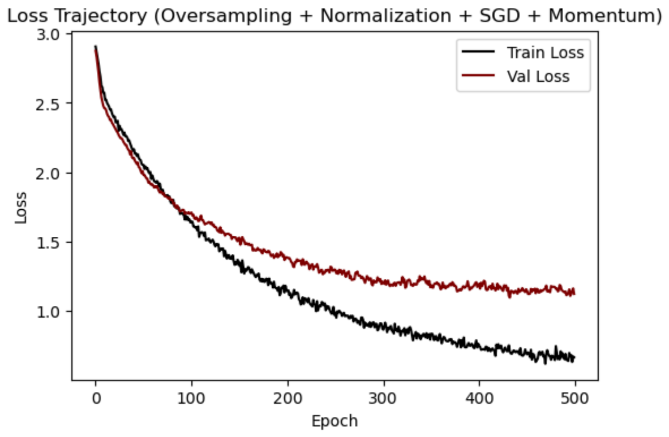
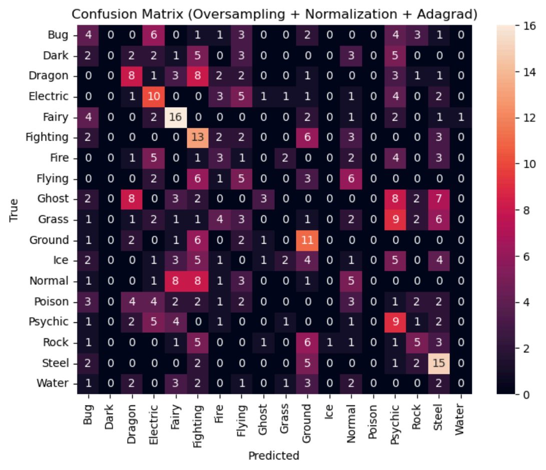


Figure 13. Confusion Matrix for Oversampling + Normalization + SGD with Momentum Optimizer

Figure 14. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + SGD with Momentum Optimizer

The sixth version of the model used the Adagrad optimizer. This optimizer adapts the learning rate for each parameter individually based on the historical sum of squared gradients, giving infrequently updated parameters larger steps and frequently updated the ones with smaller steps. The expectation was that the training loss curve would have a steep initial drop, followed by an eventual plateau as the adaptive rates got smaller. Validation loss would follow this as well. Training accuracy would rise steeply prior to its plateau, with validation accuracy mirroring until it flattens sooner. This would cause a narrow gap to form between the two curves. This model performed similarly to that of the one with the SGD optimizer, having a low accuracy of 15%, but with a validation accuracy 10% greater. The loss trajectories were nowhere near convergence and were still pretty high as well (**Figure 16**). However, it could be seen that there was still room for this model to improve given enough epochs, as a plateau had not yet occurred. The confusion matrix also only showed 3 types being accurately predicted, further implying that the model could be improved (**Figure 15**). Like SGD, this model would move to the final stage of testing.



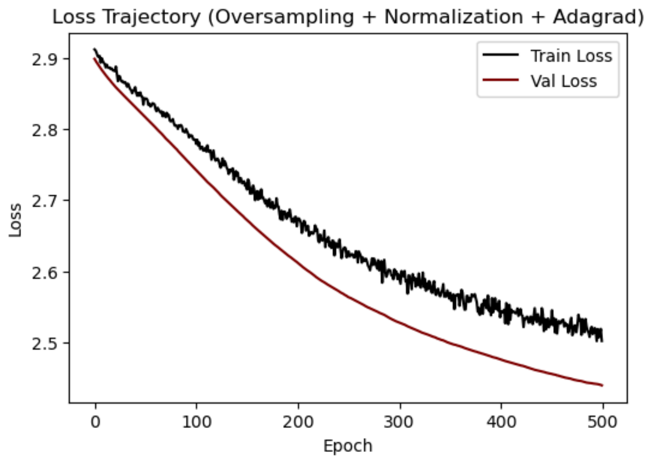
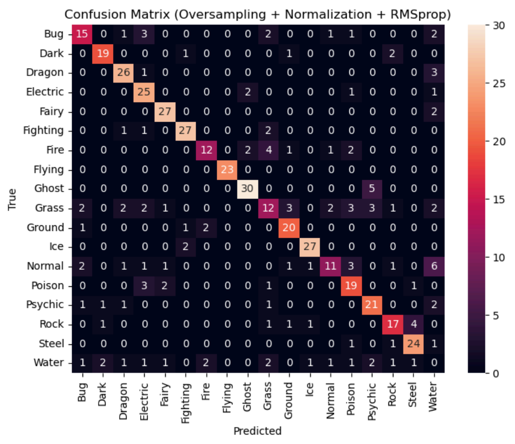


Figure 15. Confusion Matrix for Oversampling + Normalization + Adagrad Optimizer

Figure 16. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + Adagrad Optimizer

The seventh model used the RMSprop optimizer, which performed very similarly to SGD with momentum. This optimizer was expected to cause the training loss curve to experience a sharp drop due to the adaptive learning rate mechanism. As training progressed, the rate of loss reduction would slow and gradually flatten. The validation loss would follow the same trend. Training and validation accuracy would rise quickly at first and slowly plateau. A small gap may remain due to generalization. This model showed convergence on the accuracies at about 75% and lower loss trajectories, though they were extremely distant (**Figure 18**). The confusion matrix showed 8 types being predicted accurately (**Figure 19**). Like SGD momentum, this model seemed to struggle to find convergence before plateauing.



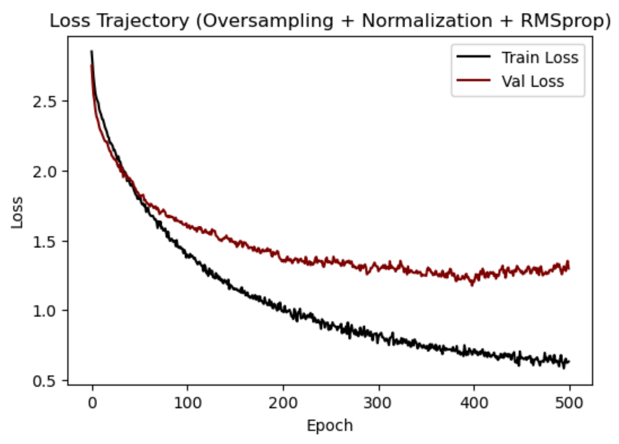
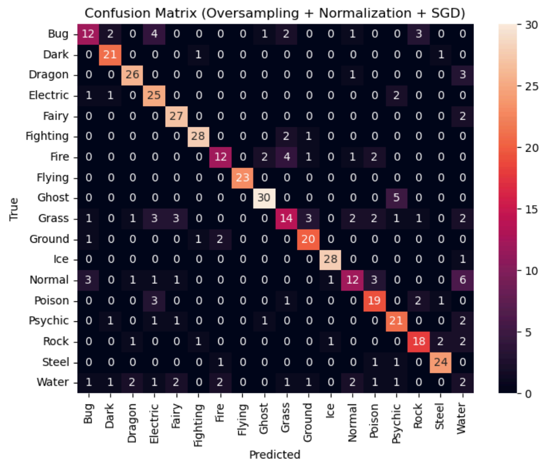


Figure 15. Confusion Matrix for Oversampling + Normalization + RMSprop Optimizer

Figure 16. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + RMSProp Optimizer

Results (SGD Vs. Adagrad Expanded)

Previous models showed that the SGD and Adagrad optimizers could yield better performance if given more epochs to run on. Given this, it was decided to attempt using SGD over 5,000 epochs and Adagrad over 20,000 epochs. For SGD, the accuracy graph showed 2 small plateaus at about 800 and 1,500 epochs. Similarly, the loss trajectories remained converged until about 800 epochs, continuing to trend farther and farther away from each other as the number of epochs grew (**Figure 18**). This likely meant that the data began to overfit after around 800 epochs. At that point, the model achieved around 50% accuracy, which still isn’t the best. At 5,000 epochs, the model achieved about 73% accuracy, though the validation accuracy began to plateau while the training accuracy continued to increase, further suggesting that the model began to memorize the data. The confusion matrix showed that only 8 types were predicted accurately, which is in line with the 50% accuracy margin (**Figure 17**).



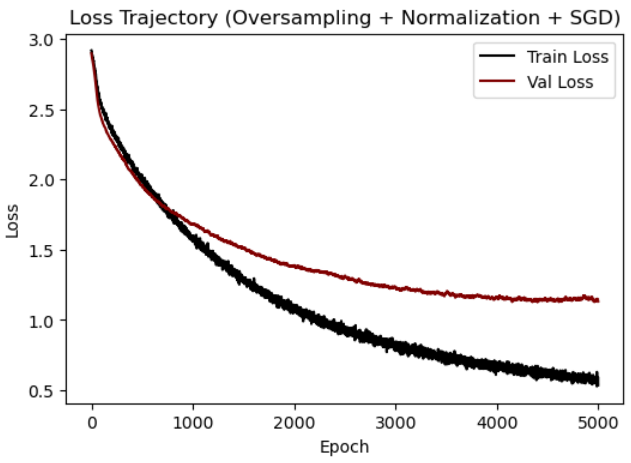
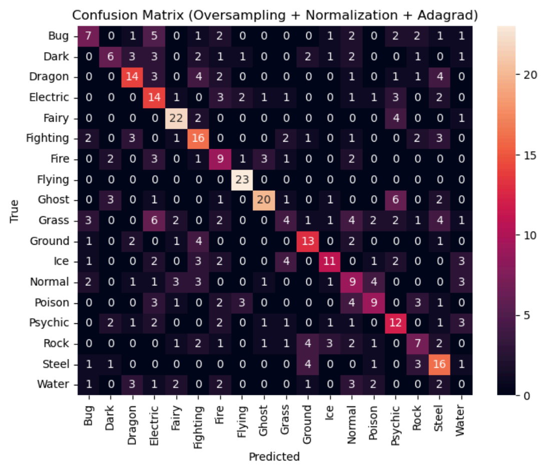


Figure 17. Confusion Matrix for Oversampling + Normalization + SGD Optimizer (5000 epochs)

Figure 18. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + SGD Optimizer (5000 epochs)

Using the Adagrad optimizer at 20,000 epochs, the validation accuracy peaked at about 50% accuracy, with the training accuracy shortly behind at 44%. The validation accuracy began hitting lengthy plateaus at about 10,000 epochs, while the training accuracy continued to go up until the very end, but at a much lower slope. The loss trajectory for both the validation accuracy and training accuracy converged at around 10,000 epochs and maintained convergence for about another 7,500, with validation loss only slightly exceeding that of the training loss by epoch 20,000 (**Figure 20**). By this point one could argue that the model began to overfit the data a bit. However, one could argue that it would be worth attempting 30,000 epochs with this model and the adagrad optimizer to see if they continue to rise and converge as well. Since this model took an extremely long time to run, this was not feasible for this project. The confusion matrix showed about 10 types being accurately predicted, which is in line with the first adagrad run

(**Figure 19**). Overall, adagrad certainly improved with greater epochs, but the runtime is too great for it to be feasibly used.



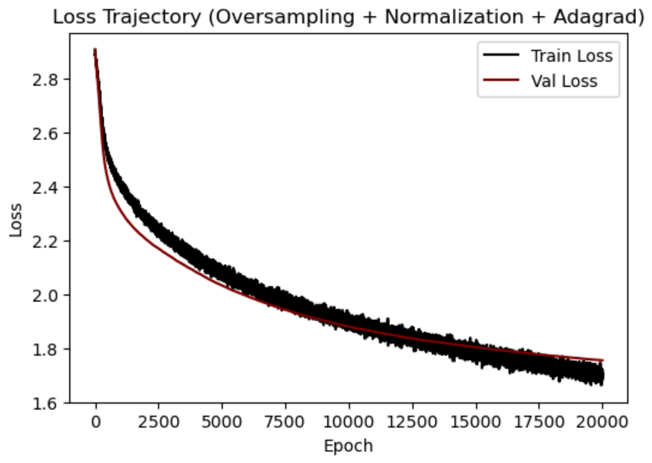


Figure 19. Confusion Matrix for Oversampling + Normalization + Adagrad Optimizer (20000 epochs)

Figure 20. Training Loss Trajectory Vs. Validation Loss Trajectory for Oversampling + Normalization + Adagrad Optimizer (20000 epochs)

Discussion

Some models performed much better than others, as there were large differences in the accuracy achieved between models. Considering there were 18 types that the model could choose from for classification, nearly every model achieved over 30% accuracy, which would have been an acceptable margin for the amount of classification labels used in this project. However, many models failed to converge on their loss trajectories before they plateau, indicating that an increase in epochs would not be a viable fix and that the models would need to be improved in another manner.

Key Takeaways:

Each version and iteration of the model provided key insights into how this project could be developed and what would need to be done moving forward to improve the model overall. First, the two iterations of the model showed that complex models tend to overfit the data because they have enough capacity to fit the training data too well, including noise. The goal is to balance model complexity with generalization by using regularization, sufficient data, and validation monitoring. The first iteration of the model tended to memorize the data instead of improving on its generalization, hence the change. Each version of the model showed the effects that preprocessing and optimizers can have on model performance. The original data was skewed due to the uneven distribution of primary types in the Pokemon franchise, which resulted in the model having a difficult time learning the data. It also created sampling biases for types that were much more frequent than others. Through artificial oversampling and normalization of the data, this distribution was corrected and therefore greatly improved the accuracy of the model. Though five different optimizers were used, it could be seen that some optimizers would exhibit the same trends as others. While it is currently difficult to say which optimizer resulted in the best model overall, it can be said that simpler optimizers can be just as effective as more complex ones. The results have shown that the primary type of a Pokemon can be predicted by its base stats the majority of the time. The analysis revealed that this model, while not extremely accurate, provides promise to future iterations given its accuracy over 18 different elemental type labels. With so many labels this would already be a challenging task, and given that there are only 1,025 Pokemon it became even more difficult. Yet, the model proved that it is possible to generally predict with about 50-75% accuracy a Pokemon's primary type.

Future Improvements:

The biggest hurdle for the model to overcome was its ability to predict rare primary types. In each new release of Pokemon titles, there typically is a similar distribution of Pokemon types added to the franchise. This means that the gap between the number of Pokemon with the most common and least common types will only get exponentially larger, further requiring the preprocessing done in this project if this were to be repeated after a new set of releases. However, as more titles are released, more Pokemon of each type would be added to the franchise, which could help improve the model's accuracy for that type. An improvement that could be made in the present would be experimenting with different learning rates. In this project, each optimizer utilized its default learning rate, which could have affected convergence rates and accuracy scores. The model architecture itself could be altered as well, as it seems that even the architecture from iteration 2 of the model is still too large for certain optimizers. Custom architectures could be made for each optimizer to see the effects it has on convergence. Finally, it may be worth looking at which features were used most in the model's predictions. It is possible that certain base stats did not really have any effect on the model's performance, effectively hindering it by increasing its runtime.

Project Resources

Kaggle Dataset:

https://www.kaggle.com/datasets/rzgiza/pokdex-for-all-1025-pokemon-w-text-description/code

Team Github:

https://github.com/Ewoggerts/Cognitive_modeling_assignments

Project Folder:

https://github.com/Ewoggerts/Cognitive_modeling_assignments/tree/main/project

Pokemon Type Chart:

https://i.pinimg.com/736x/cd/18/4d/cd184d5c77d35d099b14e38a4fa0a5fc.jpg