Final Project

Group 3: Aaditya Reddy Anugu, Justin Kang, Nathaniel Alexander Koehler, Patrick Soo, Zhixuan Wang

Introduction

Geoguesser is a game in which players are randomly placed somewhere in Google Street View and need to guess what their exact location is.

Previous literature [3][5] discusses using techniques like CNNs and transfer learning to analyze other image datasets to identify pneumonia in x-rays and find skin lesions and skin cancer from pictures. These papers have used these techniques to identify features that can help models classify an image. However, one of the shortcomings previous literature emphasizes is a lack of sufficient data in training models.

The following dataset, GeoLocation - Geoguessr Images (50K), found through Kaggle, contains 50,000 streetview images of the world, with every image belonging to 1 of 150+ countries. The data itself is not uniform as there are more images within certain countries compared to others, but we plan to combine datasets and prune folders with insufficient data.

Problem Definition

We are interested in seeing if we can train a model to accurately perform this task of identifying key objects that belong to only specific parts of the world, and correctly identifying which country the street view image is from.

This brings us to our problem - there may be certain circumstances in which it would be helpful to determine a relative location given a set of images, such as crime investigations. Thus, our motivation towards a potential solution to this is to start by using the Geoguessr dataset found through Kaggle, and train the dataset to determine which country it is in.

Methods

Data Cleaning

Before our preprocessing step, we decided to first clean the dataset deleting non-uniform resolution images, and then deleted folders (classes) that had less than 100 images, as we believed it would be hard to classify images of those classes because of the small amount of data given. Lastly, due to the large dimensionality of the dataset, we decided to resize every remaining image into ⅓ of its original size. The original image dimensionality was 1536 x 662, but after resizing, the dimensionality became 512 x 220. Next, for the actual preprocessing step, we implemented standardization across the entirety of the remaining data. Due to issues with the memory when creating the datasets, we decided to utilize Tensorflow Datasets, which helped with memory as it doesn't load the entirety of the dataset in the variable at once.

Detailed Steps:

1. Manually went through all folders of original dataset A, removed classification folders that had < 100 images and created new dataset B.
2. Using a python resize script, scaled each image of the new dataset B into ⅓ of its original size, resulting from 1536 x 662 to 512 x 220.
3. Python script used t o convert dataset to tensorflow dataset
4. Resulting dataset C:
    a. Tensorflow dataset type
    b. Containing folders only 100+ images
    c. Each image resized to 512 x 220

Preprocessing/Model #1

Preprocessing Method: Image Standardization

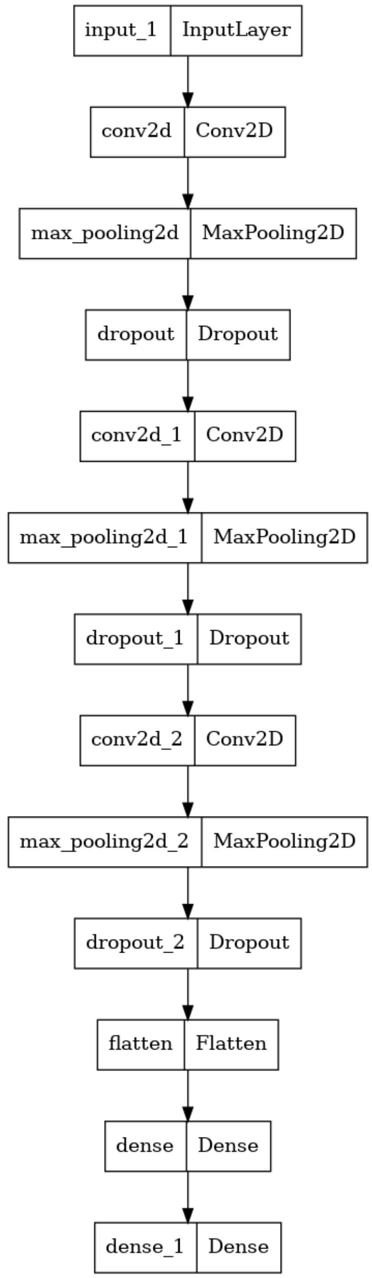
Image standardization is defined by $X' = \frac{X - \mu}{\sigma}$, and this preprocessing method will help us reduce the lighting and exposure for training. By doing this, it allows us to have uniformity across all images and may improve convergence during training. We divided the dataset into 70% training, 15% validation, and 15% testing. We fit a standard scaling (z-score) layer from Tensorflow Keras to the training set and transformed all three sets of data with this fitted layer. This resulted in our standardized training, validation, and testing datasets.

Detailed Steps:

1. Using sklearn's train_test_split, split the image-label into training, validation, and testing
    a. First split image-labels to 75% training, and 25% testing
    b. Split 20% of training into validation
    c. Resulting in the following dataset split:
        i. 60% training + 15% validation + 25% testing

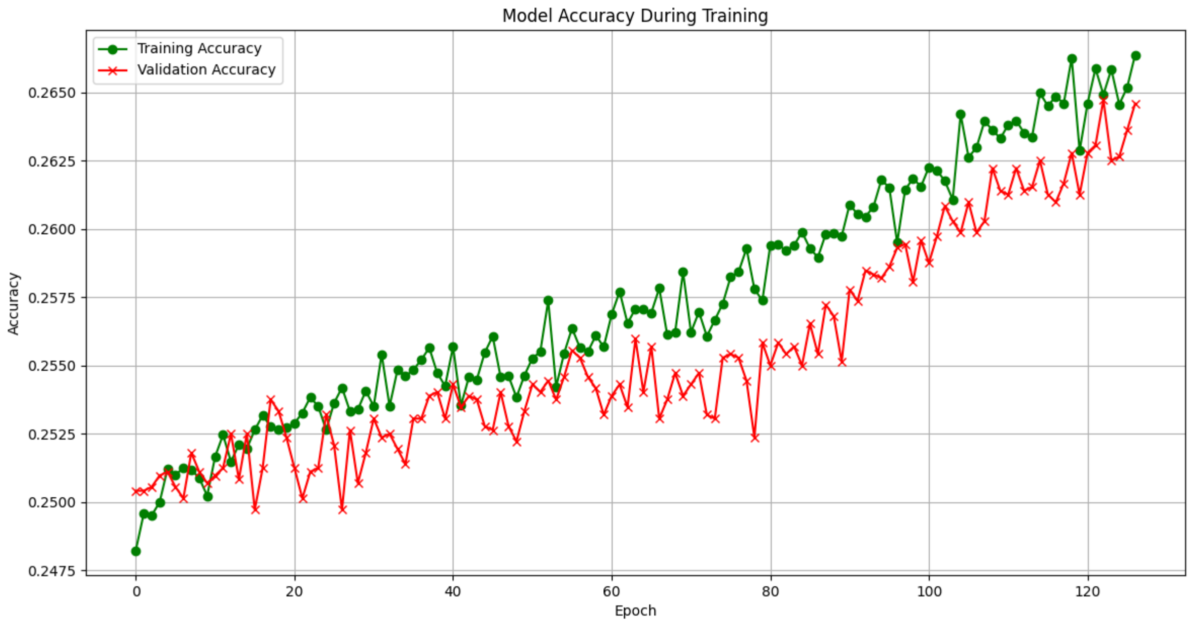
Model: Convolutional Neural Network

Our first machine learning algorithm that we used was a Convolutional Neural Network, which was a type of supervised learning. We chose to do CNN because of its efficacy with handling image data. Widely known image classification models such as the ResNet or DenseNet also employ convolution layers. Conv2D layers take filters to extract the information and essentially summarize them into a pixel. MaxPooling layers have been known to perform well with Conv2D layers, and they also reduce the dimensions of the image. The model architecture is shown in the diagram below. In order to prevent overfitting, we used L1 regularizer and Dropout layers. The final layer has a softmax function that allows the image classification. As for the activation functions of the Conv2D layers, we used ReLU, as it is faster than Sigmoid to compute and also doesn't have Sigmoid's vanishing gradient issue.

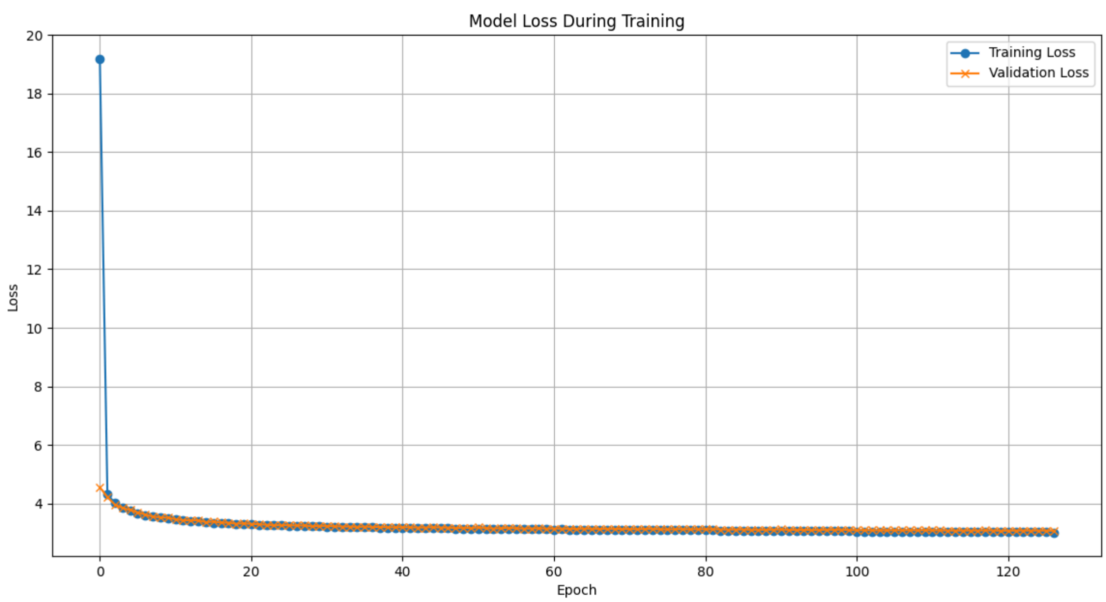


Results and Discussion

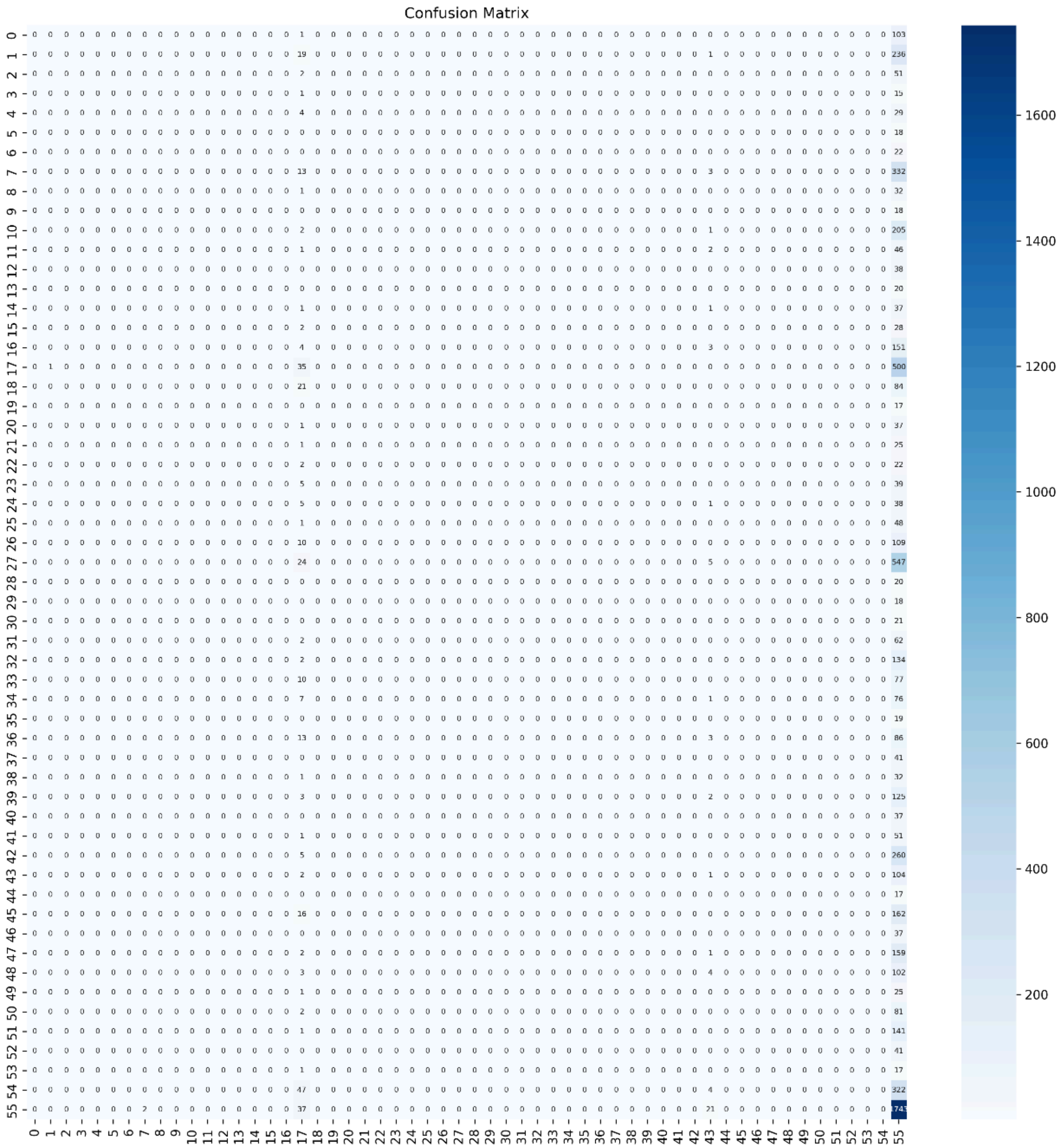
[Figure 1.2] Model Accuracy vs. Training Epoch



[Figure 1.2] Model Loss vs. Training Epoch



[Figure 1.3] Confusion Matrix

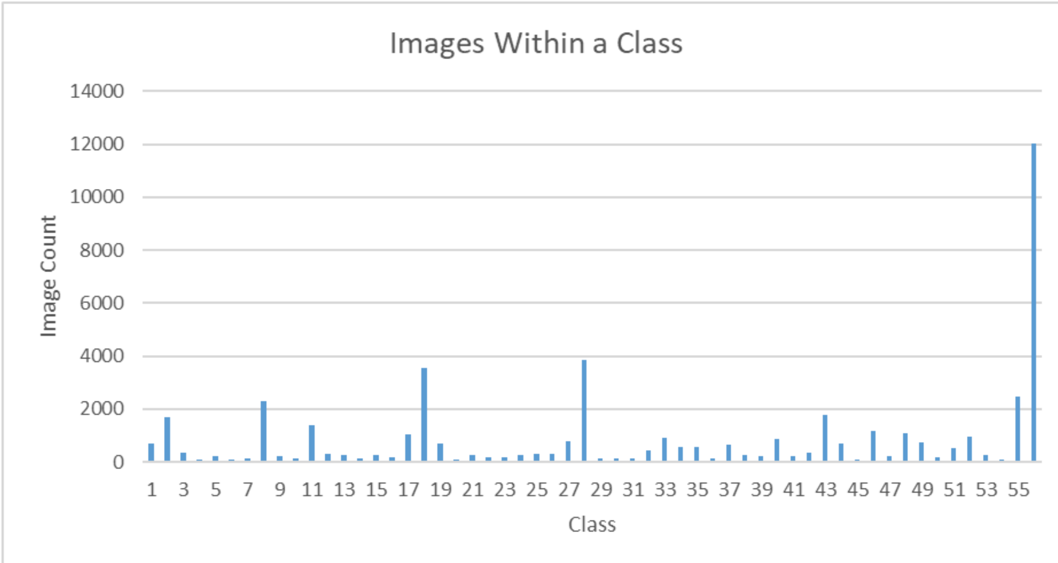


[Figure 1.4] Precision Score Chart

	precision	recall	f1-score	support
0	0.0000	0.0000	0.0000	104
1	0.0000	0.0000	0.0000	256

2	0.0000	0.0000	0.0000	53
3	0.0000	0.0000	0.0000	16
4	0.0000	0.0000	0.0000	33
5	0.0000	0.0000	0.0000	18
6	0.0000	0.0000	0.0000	22
7	0.0000	0.0000	0.0000	348
8	0.0000	0.0000	0.0000	33
9	0.0000	0.0000	0.0000	18
10	0.0000	0.0000	0.0000	208
11	0.0000	0.0000	0.0000	49
12	0.0000	0.0000	0.0000	38
13	0.0000	0.0000	0.0000	20
14	0.0000	0.0000	0.0000	39
15	0.0000	0.0000	0.0000	30
16	0.0000	0.0000	0.0000	158
17	0.1122	0.0653	0.0825	536
18	0.0000	0.0000	0.0000	105
19	0.0000	0.0000	0.0000	17
20	0.0000	0.0000	0.0000	38
21	0.0000	0.0000	0.0000	26
22	0.0000	0.0000	0.0000	24
23	0.0000	0.0000	0.0000	44
24	0.0000	0.0000	0.0000	44
25	0.0000	0.0000	0.0000	49
26	0.0000	0.0000	0.0000	119
27	0.0000	0.0000	0.0000	576
28	0.0000	0.0000	0.0000	20
29	0.0000	0.0000	0.0000	18
30	0.0000	0.0000	0.0000	21
31	0.0000	0.0000	0.0000	64
32	0.0000	0.0000	0.0000	136
33	0.0000	0.0000	0.0000	87
34	0.0000	0.0000	0.0000	84
35	0.0000	0.0000	0.0000	19
36	0.0000	0.0000	0.0000	102
37	0.0000	0.0000	0.0000	41
38	0.0000	0.0000	0.0000	33
39	0.0000	0.0000	0.0000	130
40	0.0000	0.0000	0.0000	37
41	0.0000	0.0000	0.0000	52
42	0.0000	0.0000	0.0000	265
43	0.0185	0.0093	0.0124	107
44	0.0000	0.0000	0.0000	17
45	0.0000	0.0000	0.0000	178
46	0.0000	0.0000	0.0000	37
47	0.0000	0.0000	0.0000	162
48	0.0000	0.0000	0.0000	109
49	0.0000	0.0000	0.0000	26
50	0.0000	0.0000	0.0000	83
51	0.0000	0.0000	0.0000	142
52	0.0000	0.0000	0.0000	41
53	0.0000	0.0000	0.0000	18
54	0.0000	0.0000	0.0000	373
55	0.2542	0.9667	0.4025	1803
accuracy			0.2462	7226
macro avg	0.0069	0.0186	0.0089	7226
weighted avg	0.0720	0.2462	0.1067	7226

[Figure 1.5] # Images In a Class



(Note: This graph goes from 1 to 56 instead of 0 to 55)

Our confusion matrix shows very little relationship in terms of having a visible diagonal. This possibly implies that the models were not accurate in predicting each class, and were heavily predicting class 55/56 (United States), which was most likely due to the disproportionate amount of data it had compared to the other classes.

Analysis of Convolutional Neural Network

Overall, the visualizations show and imply that the model's accuracy was very low, and it can be inferred that it is most likely linked to the way we cleaned our data and or preprocessed it. As previously mentioned, our dataset was also not uniform; it shows within the confusion matrix with the lack of diagonal and heavy weightage on class 55. Furthermore, the precision score chart (Figure 1.5) showed that class 55 had a staggering high difference in precision in comparison to the others because it had over 12,000 images while most others had around an average of 200~600 images, which made the model more biased in predicting class 55.

As for the model itself, the first two visualizations indicated that the model was doing well on the data in relation to the disproportionate dataset. Figure 1.1 showed that the validation and training accuracy were both going up, and both were increasing at a relatively same rate and value (disregarding outliers such as epoch 22 and 79). This meant that the model was learning, as the accuracy kept on improving overall as the number of epochs increased.

For Figure 1.2, the model loss graph is very similar to an average model loss graph. It has a sharp dip at the beginning, in which the validation and training loss converges towards a small value in the end. Because the loss of the training and validation showed convergence, this gave signs that the model was not overfitting the data.

As a result, the accuracy and loss graphs showed that the model was consistent in handling the data, but the model was biased because it was trained on a high number of images within class 55. This shows just how important data cleaning and preparation is, and if the data itself is not good, it will most likely imply that the model training will not do well either.

Preprocessing/Model #2

Further Data Cleaning*

Because of the vastly disproportionate amount of images from the USA compared to the other countries, we decided to further clean the data by lessening the amount of images within the USA before putting the dataset into training. Thus, we came up with 2 options

1. Augmenting the smaller classes by transforming them
2. Undersampling USA

However, we decided against augmenting smaller classes, as it would further increase the dimensionality (hence memory-intensity during training) of the dataset.

Preprocessing Method: Min-Max Scaling

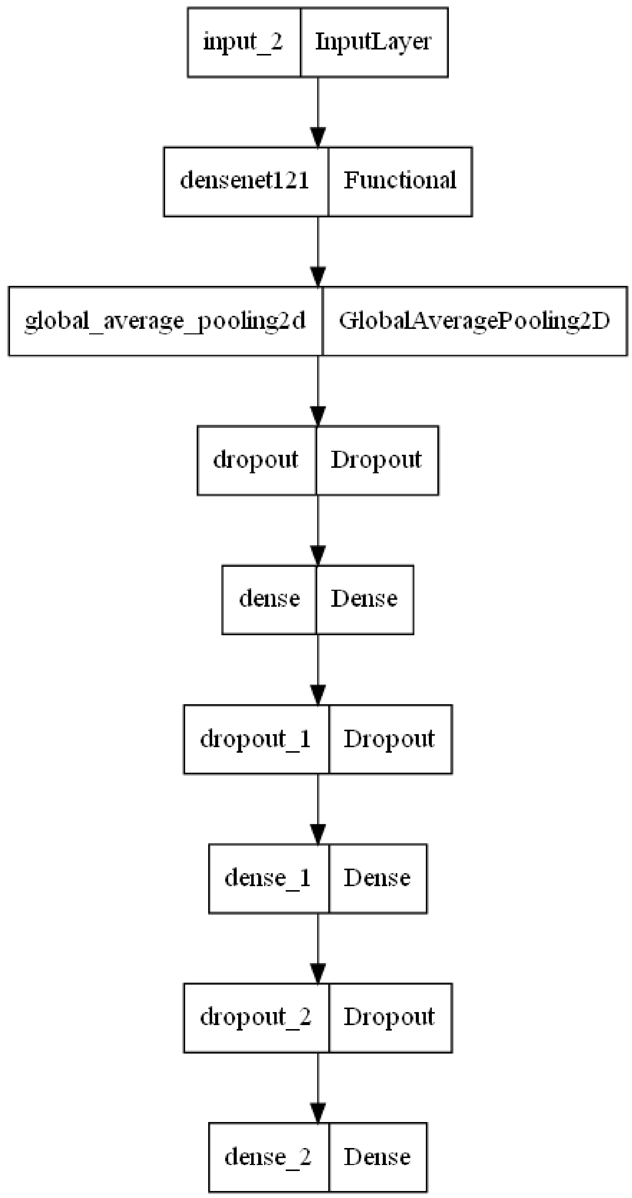
Min-Max scaling is defined by $\sqrt{X'} = \frac{X - X_{min}}{X_{max} - X_{min}}$. This preprocessing method will normalize the RGB color pixel range of 0-255 to 0-1. Like with image standardization, we hope that this method will help us with uniformity, and we would like to see how it will affect the training process and results.

Detailed Steps:

1. Getting the undersampled and augmented dataset layer, we applied a normalization layer that divides the values by 255 to make all of the value ranges to be 0 to 1 only. This layer was applied to both the training and validation layer.
2. After applying the layers, we had our new min-max scaled tensorflow dataset ready to be trained for the next model.
3. Resulting Dataset: Min-max scaled images, undersampling USA to 300 images

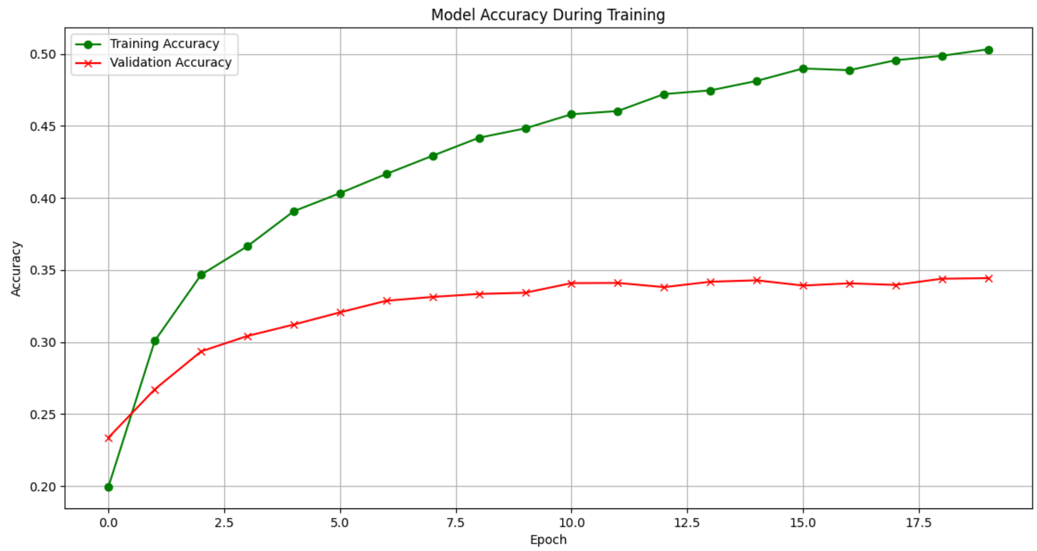
Model: Transfer Learning Model

Transfer learning is a technique that utilizes previously trained machine learning models' pretrained weights to create another model for a different use. In our case, we used a DenseNet121, which was trained on the ImageNet dataset. By removing the top of the pretrained model and adding some layers at the end, the transfer learning model boasts a quick train time (as the pretrained weights are frozen) and a relatively high accuracy. We also added some dropout layers in order to reduce overfitting, which was a pretty common problem with this dataset.

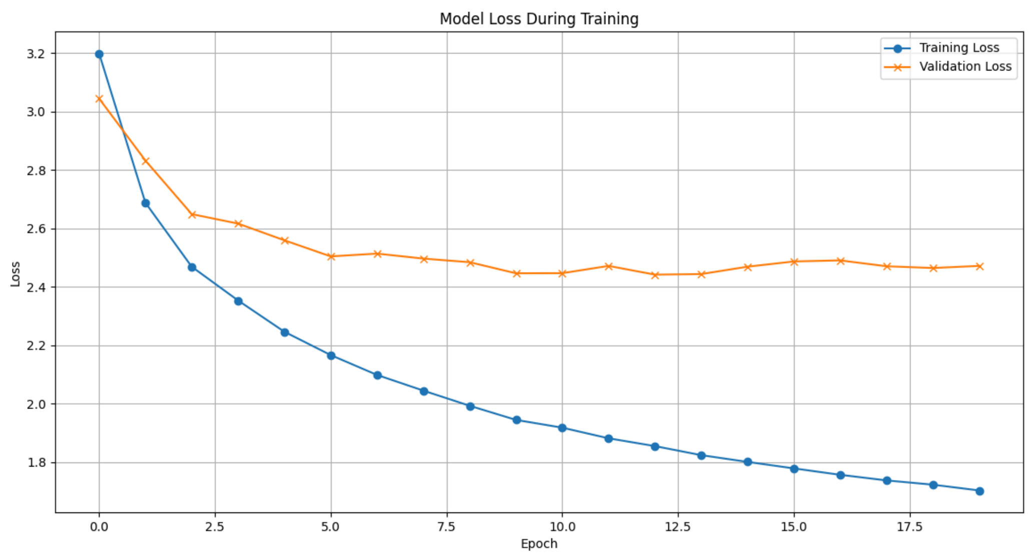


Results and Discussion

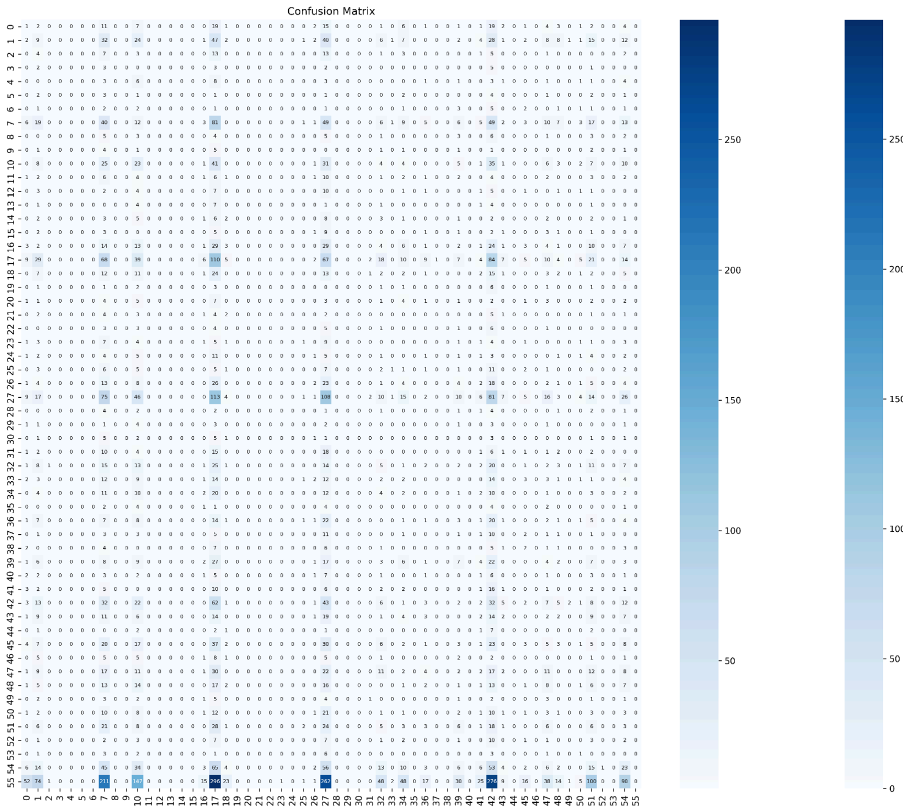
[Figure 2.1] Model Accuracy Over Epochs



[Figure 2.2] Model Loss Over Epochs



[Figure 2.3] Confusion Matrix



	precision	recall	f1-score	support
0	0.0085	0.0096	0.0090	104
1	0.0286	0.0352	0.0315	256
2	0.0000	0.0000	0.0000	53
3	0.0000	0.0000	0.0000	16
4	0.0000	0.0000	0.0000	33
5	0.0000	0.0000	0.0000	18
6	0.0000	0.0000	0.0000	22
7	0.0474	0.1149	0.0672	348
8	0.0000	0.0000	0.0000	33
9	0.0000	0.0000	0.0000	18
10	0.0387	0.1106	0.0574	208
11	0.0000	0.0000	0.0000	49
12	0.0000	0.0000	0.0000	38
13	0.0000	0.0000	0.0000	20
14	0.0000	0.0000	0.0000	39
15	0.0000	0.0000	0.0000	30
16	0.0213	0.0063	0.0098	158
17	0.0826	0.2052	0.1178	536
18	0.0000	0.0000	0.0000	105
19	0.0000	0.0000	0.0000	17
20	0.0000	0.0000	0.0000	38
21	0.0000	0.0000	0.0000	26
22	0.0000	0.0000	0.0000	24
23	0.0000	0.0000	0.0000	44
24	0.0000	0.0000	0.0000	44
25	0.0000	0.0000	0.0000	49
26	0.0769	0.0168	0.0276	119
27	0.0966	0.1875	0.1275	576
28	0.0000	0.0000	0.0000	20

29	0.0000	0.0000	0.0000	18
30	0.0000	0.0000	0.0000	21
31	0.0000	0.0000	0.0000	64
32	0.0281	0.0368	0.0318	136
33	0.0000	0.0000	0.0000	87
34	0.0118	0.0238	0.0158	84
35	0.0000	0.0000	0.0000	19
36	0.0149	0.0098	0.0118	102
37	0.0000	0.0000	0.0000	41
38	0.0000	0.0000	0.0000	33
39	0.0556	0.0538	0.0547	130
40	0.0000	0.0000	0.0000	37
41	0.0115	0.0192	0.0144	52
42	0.0302	0.1208	0.0483	265
43	0.0377	0.0187	0.0250	107
44	0.0000	0.0000	0.0000	17
45	0.0492	0.0169	0.0251	178
46	0.0000	0.0000	0.0000	37
47	0.0576	0.0679	0.0623	162
48	0.0000	0.0000	0.0000	109
49	0.0000	0.0000	0.0000	26
50	0.0000	0.0000	0.0000	83
51	0.0182	0.0423	0.0255	142
52	0.0000	0.0000	0.0000	41
53	0.0000	0.0000	0.0000	18
54	0.0735	0.0617	0.0671	373
55	0.0000	0.0000	0.0000	1803
accuracy			0.0536	7226
macro avg	0.0141	0.0207	0.0148	7226
weighted avg	0.0304	0.0536	0.0359	7226

Analysis of Transfer Learning Model

Firstly, compared to CNN's confusion matrix, DenseNet121's transfer learning model suggests that it has a higher chance of guessing correctly, albeit still only vaguely resembling a diagonal matrix. However, because the guesses are spread out, it implies that there is less bias within the model.

Taking a further look at the first two loss/accuracy plots, we can see that there is also a significant difference in trends compared to CNN. While both loss and accuracy seemed to converge after a few epochs within CNN, there seems to be a plateau in both loss and accuracy for our current transfer learning model (especially validation loss), which may suggest that the model could be overfitting its data and will have trouble predicting new data.

As mentioned before, transfer learning utilizes pretrained weights to train on a different use case. Since DenseNet121 was trained on the ImageNet dataset, which is an image dataset,we think it performed slightly better than a CNN (less complex than a DenseNet). The extreme dip in accuracy compared to CNN is most likely due to CNN's high bias towards guessing USA without data cleaning (as there were a lot more USA's in the testing set for the CNN than for the transfer learning model), which may have inflated the model's accuracy.

<u>Preprocessing/Model #3</u>

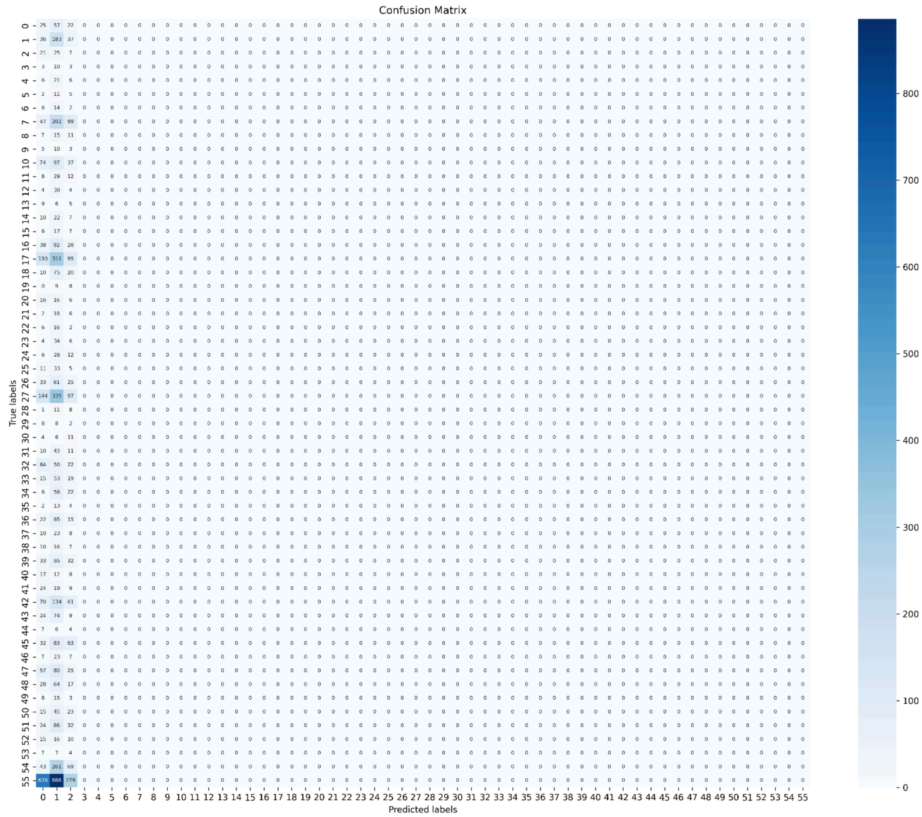
For this portion of preprocessing, we simply applied f(x) = log(x) to the entire dataset, with x being the 3 RGB values ranging from 0 to 255. This was done by applying a map layer to both training and validation tensorflow dataset with the log function. In addition, we also decided to keep with undersampling and size reduction, resulting the following:

- Dataset that has log(x) applied to all, undersampling USA to only 300 images, and reduced to 1/3rd of the original size.

Model: K-Nearest Neighbors

For our last machine learning algorithm, we decided to use K-Nearest Neighbors because it vastly differentiated from the other two models. While both CNN and Transfer Learning Models require iteration, KNN differs from the two because it does not utilize iteration. Instead, it rearranges the dataset to points that are similar to each other. We chose this model because KNN does well with general classification, which is what we are trying to do with images. Since SKLearn's KNN model does not take in Tensorflow Datasets, we also had to convert the tensorflow dataset into numpy arrays. Before doing so, we applied the image preprocessing steps mentioned above and then fit the data into the SKLearn's KNN model to be saved into a pickle file. Then, using a testing set, we called the model to make predictions, and the following confusion matrix was the result of the predictions:

[Figure 3.1] KNN Confusion Matrix



	precision	recall	f1-score	support
0	0.0134	0.2404	0.0253	104
1	0.0458	0.7148	0.0861	256
2	0.0051	0.1321	0.0099	53
3	0.0000	0.0000	0.0000	16
4	0.0000	0.0000	0.0000	33
5	0.0000	0.0000	0.0000	18
6	0.0000	0.0000	0.0000	22
7	0.0000	0.0000	0.0000	348
8	0.0000	0.0000	0.0000	33
9	0.0000	0.0000	0.0000	18
10	0.0000	0.0000	0.0000	208
11	0.0000	0.0000	0.0000	49
12	0.0000	0.0000	0.0000	38
13	0.0000	0.0000	0.0000	20
14	0.0000	0.0000	0.0000	39
15	0.0000	0.0000	0.0000	30
16	0.0000	0.0000	0.0000	158

17	0.0000	0.0000	0.0000	536
18	0.0000	0.0000	0.0000	105
19	0.0000	0.0000	0.0000	17
20	0.0000	0.0000	0.0000	38
21	0.0000	0.0000	0.0000	26
22	0.0000	0.0000	0.0000	24
23	0.0000	0.0000	0.0000	44
24	0.0000	0.0000	0.0000	44
25	0.0000	0.0000	0.0000	49
26	0.0000	0.0000	0.0000	119
27	0.0000	0.0000	0.0000	576
28	0.0000	0.0000	0.0000	20
29	0.0000	0.0000	0.0000	18
30	0.0000	0.0000	0.0000	21
31	0.0000	0.0000	0.0000	64
32	0.0000	0.0000	0.0000	136
33	0.0000	0.0000	0.0000	87
34	0.0000	0.0000	0.0000	84
35	0.0000	0.0000	0.0000	19
36	0.0000	0.0000	0.0000	102
37	0.0000	0.0000	0.0000	41
38	0.0000	0.0000	0.0000	33
39	0.0000	0.0000	0.0000	130
40	0.0000	0.0000	0.0000	37
41	0.0000	0.0000	0.0000	52
42	0.0000	0.0000	0.0000	265
43	0.0000	0.0000	0.0000	107
44	0.0000	0.0000	0.0000	17
45	0.0000	0.0000	0.0000	178
46	0.0000	0.0000	0.0000	37
47	0.0000	0.0000	0.0000	162
48	0.0000	0.0000	0.0000	109
49	0.0000	0.0000	0.0000	26
50	0.0000	0.0000	0.0000	83
51	0.0000	0.0000	0.0000	142
52	0.0000	0.0000	0.0000	41
53	0.0000	0.0000	0.0000	18
54	0.0000	0.0000	0.0000	373
55	0.0000	0.0000	0.0000	1803
accuracy			0.0298	7226
macro avg	0.0011	0.0194	0.0022	7226
weighted avg	0.0019	0.0298	0.0035	7226

Analysis of K-Nearest Neighbors

Overall, KNN was probably the least effective training model to fit for this dataset because of the high dimensionality of the dataset. Each image consists of 1536 x 662 x 3 data points, and we had over 35,000 images combined, even after undersampling the USA. With the other models, we were able to utilize tensorflow datasets, which made memory allocation significantly easier. In addition, because KNN forced the dataset to be numpy arrays, unpacking them took a significant amount of memory as well and time as well. As a result, we had to further resize the images to 56 x 384.

KNN is typically considered good for general classifications, but due to the sheer amount of data points, it made the model not only hard to run, but also harder for the model to correctly classify labels because it showed a significant decrease in accuracy despite better data cleaning. This is shown through our confusion matrix, as it mirrors CNN in the way that it tends to guess towards only 1-2 specific labels.

Conclusion

Overall Analysis and Comparison of Algorithms

Overall Model Accuracy Comparison Table

	Image Standardization + CNN	**Min-Max + Transfer Learning**	**Log Scaling + KNN**
Test Accuracy	24.62%*	5.36%	2.98%

* Test accuracy high for CNN potentially due to very unbalanced testing dataset

Overall, we can see that the KNN performed the worst, and the Convolutional Neural Network. One main reason for this could be the sheer number of features of the dataset. Image classification, especially with a problem as difficult as ours, requires a complex model, and as the transfer learning model was the most complex model, it seems that it was the most effective in solving the problem.

Taking a look at the confusion matrices as well, transfer learning seemed to also resemble an identity matrix the most, which suggests that it was more likely to conduct correct guesses compared to CNN (which often kept on guessing USA), and KNN (which often kept on guessing Argentina and Australia). Lastly, as previously mentioned, although CNN had the highest accuracy, it was heavily biased without undersampling the USA class. However, the low accuracy of the models show just how important dataset cleaning and augmenting can be.

Next Steps

Dataset Modifications

- Granted that our dataset was actually quite small, we could expand more on the dataset by adding more images to the set. This can be either done manually through getting

images from google street view, or creating a script to run GeoGuesser to get new images and data from there directly.
- Testing out each preprocessing method on the same model to see which one does best to fine tune the preprocessing step.
- Make overall data within each folder to be uniform

Model Selections/Changes
- Given that we now know how to manipulate numpy arrays and tensorflow datasets better, we would like to explore using other models that run tensorflow datasets more easily.
- Adjusting certain layers to models to see which combinations result in better accuracies

Final Thoughts

Our project really goes to show the importance of data itself. While different preprocessing methods and machine learning/deep learning methods did make a difference, the overall performances of these models were subpar. This was mostly due to the dataset being unbalanced, as well as having not that many data points. Of course, it is more difficult to have a lot of data points with image data as compared to other datasets such as numerical data, but the lack of balanced data did cause a poor performance from our models.

GanttChart (1)

Project Proposal Contribution Table

Group Member	Contributions
Aaditya Anugu	Intro and Background
Justin Kang	Problem definition, Methods, Potential Dataset
Nathaniel Koehler	Github Page, Presentation Slides
Patrick Soo	Problem definition, Methods, Potential Dataset, Video Creation
Zhixuan Wang	Problem Definition, Potential Dataset, Video Creation

Project Midterm Contribution Table

Group Member	Contributions

Aaditya Anugu	Intro, Background, Gantt Chart
Justin Kang	Methods, Results, & Discussion
Nathaniel Koehler	Methods, Github Pages Website
Patrick Soo	Methods, Results, & Discussion
Zhixuan Wang	Methods, Results, & Discussion

Project Final Contribution Table

Group Member	**Contributions**
Aaditya Anugu	Methods, Gantt Chart, PowerPoint Slides, Discussion
Justin Kang	Methods, Results, Discussion, Preprocessing
Nathaniel Koehler	Methods, Github Pages Website, Preprocessing
Patrick Soo	Methods, Results, Discussion, Model Training
Zhixuan Wang	Methods, Results, Preprocessing, Discussion

References

[1] R. K., “Geolocation - Geoguessr images (50k),” Kaggle, https://www.kaggle.com/datasets/ubitquitin/geolocation-geoguessr-images-50k (accessed Feb. 20, 2024).

[2] A. Bhandari, “Feature scaling: Engineering, Normalization, and standardization (updated 2024),” Analytics Vidhya, https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalizations tandardization/ (accessed Feb. 20, 2024).

[3] Kim, H.E., Cosa-Linan, A., Santhanam, N. et al. Transfer learning for medical image classification: a literature review. BMC Med Imaging 22, 69 (2022). https://doi.org/10.1186/s12880-022-00793-72

[4] “Normalization | Machine learning | Google for developers,” Google,

https://developers.google.com/machine-learning/data-prep/transform/normalization (accessed Feb. 20, 2024).

[5] Weyand, T., Kostrikov, I., Philbin, J. (2016). PlaNet - Photo Geolocation with Convolutional Neural Networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds) Computer Vision – ECCV 2016. ECCV 2016. Lecture Notes in Computer Science(), vol 9912. Springer, Cham. https://doi.org/10.1007/978-3-319-46484-8_3