GEOGUESSR PREDICTOR

Ethan Brown, Owen Greybill, Jared Krauss, Liam Waterbury

CSSE/MA416 Deep Learning

10 November 2023

ABSTRACT

Geoguessr [1] is a popular browser game involving a human's ability to recognize images and categorize them into the countries in which they were taken. As with most games, it poses an interesting question, “Can a computer do it better?” Building off of past work is important for efficiency, so using transfer learning with EfficientNet and VGG16 models was ideal. Other ideas include custom loss functions and predicting continents instead of countries. The main exploration occurred through modifying hyperparameters and image sizes fed into the VGG16 and EfficientNet models. The best results were produced by larger image sizes and more epochs with the EfficientNetB1 model. The final model produced a **44.08%** accuracy, a **19.05%** improvement from the **25.03%** baseline.

1.INTRODUCTION

Our project stems from the browser game Geoguessr [1]. Geoguessr is very popular, generating hundreds of thousands of views on platforms like Youtube and spawning competitions for thousands of dollars of prize money. Geoguessr involves showing a player a spot in the world with a **360** degree view. The player is given a time limit to guess where in the world they are located. The more accurate their selection on the world map is to the correct location, the higher their score. Our project aims to develop a network that accurately identifies the country depicted in a photo.

Training a model to do this poses some interesting problems. First, some of the images in our data set are relatively plain with few identifying features- the type of photos that are hard for a human to figure out as well. The images may be of a single road with fields/plains on each side, which is too nondescript to narrow down to a country. The second issue is that some countries, especially neighboring ones, look very similar to each other. Lastly, unlike an image recognition problem to recognize simple objects or numbers, the dataset is full of complex photos of many objects, signage, and other important details. This leads to a greater importance in having detailed, higher resolution images, which will slow down training and limit options for model construction.

The original pursuit of our goal was relatively standard and similar to other deep learning projects/topics. We began by preprocessing the data, establishing baselines, training models with a variety of different hyperparameters, and comparing them. Where we divulged from the main path was in the development of our custom loss function and seeking to predict continents as well as countries correctly. Taking inspiration from the geoguessr game, we designed a loss function based on the Haversine distance between the guess and the correct answer.

2.PROCESS

2.1 Data Source

The data source [2] consists of **49,997** images from the web game Geoguessr [1]. Each image is tagged with the feature of the country with which the image was taken. The goal is to predict the country of an image based on the raw pixel data. This is a classification problem with **124** classes, consisting of the countries and regions represented in the dataset. It is important to note that each country does not have equal representation. This ranges from **12014** images of the United States of America to **1** image of Belarus.

An additional data source [3] was extracted from a public github repo. This data consisted of a json file containing a mapping of countries to their centroids in longitude and latitude. This mapping required cleaning and was used to assist the creation of a custom loss function using haversine distance.

2.2 Preprocessing

The original dataset was organized into folders for each country with the **49,997** images divided into their respective country's folder. An .npz file was created that consisted of the images and their respective labels. After downloading **49,997** images to a local machine, we used sftp to transfer the folder of images to the Gauss server. We used OpenCV and the Image module from PIL to read and process the images. We used numpy to organize and save the data into a .npz file. We first created the .npz file using **32**x**32** images to allow the program to run faster. After completing, we chose **224**x**224** as the image side to allow for more precision. We were able to repurpose this code to create **128**x**128**, **64**x**64**, and **32**x**32** versions of the dataset for testing which pixel density produced the best accuracies in models.

The additional data source for country centroid data contained data for countries not represented in our image dataset. This extra data was removed and ignored. Furthermore, some countries were represented as different names between the image dataset and the centroid dataset. An example was Eswatini being the name used in the image set and Swaziland being used in the centroid dataset. This required the centroid dataset to be cleaned to have the same names as the image dataset. In addition, a few regions were not represented in the centroid dataset. In these cases, centroid data was manually added to the dataset. Finally, the longitudes and latitudes were represented in degrees so these were converted to radians.

2.3 Transfer Learning

After the preliminary work and the baselines were established, there are still many improvements that can substantially increase the performance of the models. One such improvement is utilizing existing architectures and weights to save training time and time fine tuning the architecture. This process is called transfer learning, and it can provide significant accuracy increases, especially for short-term projects like this. Although there exists more pretrained models that excel at image recognition, this paper opts to use Keras's VGG16 and EfficientNet [4] models. For both pretrained models, the imagenet weights were used.

VGG16 is a convolutional neural network architecture that is primarily used for image recognition. It has **16** layers that have weights. It is unique in that it does not have a large number of hyper-parameters.

EfficientNet [4] is a collection of convolutional neural network architectures, specializing in providing a lightweight, yet accurate architecture. The EfficientNet collection contains multiple architectures, ranging from B0-B7. EfficientNetB0 is the smallest model. As the EfficientNet version increases, so does the architecture size, as well as the computing power needed to train and training time.

This paper also explores retraining some of the convolutional layers of EfficientNet. This process would result in a longer, more computationally intense training time, but potentially a higher accuracy. The layers of EfficientNetB1 were examined, and the layers to be retrained were hand selected. The retrained layers were selected starting from the end of the convolutional layers up until diminishing returns were found in the experiments. Only convolutional layers were set to be trainable. The ending convolutional layers were picked first since the beginning layers in EfficientNet are designed to pay attention to more general image recognition concepts that are consistent across many image recognition domains. EfficientNet's layers focus on more specific things deeper into the model.

2.4 Training Utilities File

To help with fine tuning and experimenting, a TrainingUtilities.py script was created. The utilities script makes it as easy as possible to train new models and architectures, save the model and the history data, as well as generate visuals from the training. The goal was to make the workspace as clean and easy as possible and so the amount of duplicated code is limited. Figure 1 shows example code for training a new model while being able to specify the input model, layers, number of epochs, validation split and batch size.

***Figure 1**: Example usage of TrainingUtilities.py*

2.5 Continent Classifiers

With the complexity involved in correctly classifying over 100 classes, we opted to also explore a simplified problem by switching to classifying continents. This reduced the number of classes to seven. We applied a mapping from country to continent in order to label the datasets with continents. Using these new models, we trained a VGG16 based model with a final dense layer output of 7. In addition, we used the existing country model and post processed the country prediction into a continent. This was to compare the accuracy of the models and whether focusing on a specific country could yield a more accurate model when converting to the continent after.

2.6 Custom Loss Function

The inspiration for this project was the web game Geoguessr. The original game places no weight on whether the player guesses the correct country; instead, scoring is decided based on the proximity of the player's guess to the actual location. As a result, we strove to design a custom loss function that worked more directly to solve this problem by minimizing the distance between the centroid of the predicted country and the actual country. Due to our centroids being stored as longitude and latitude values, euclidean distance is not a proper method for distance measurement. Instead, we opted to use haversine distance to measure the distance between two points [5]. Haversine distance functions for this use case based on the assumption that the Earth is a perfect sphere.

$$a = sin^2(\frac{\Delta\phi}{2}) + cos(\phi_1)\cdot cos(\phi_2)\cdot sin^2(\frac{\Delta\lambda}{2}) \quad \text{[eq 1]}$$

$$c = 2\cdot atan2(\sqrt{a}, \sqrt{1-a}) \quad \text{[eq 2]}$$

$$d = R\cdot c \quad \text{[eq 3]}$$

Where:

ϕ_1, ϕ_2 are the latitudes

λ_1, λ_2 are the longitudes

R is the Earth's radius (6,371 km)

d is the distance between the two points

In order to use the haversine loss function, the model was constructed as a regression model. Instead of passing in the labeled countries as the targets, the centroid of the country was passed in. As a result, the regression model outputs an estimate for longitude and latitude. There is no activation on the final dense layer.

2.7 Post-processing

As part of the experimentation, the output of a model trained to predict countries was post-processed to convert the country prediction to a continent prediction. This was in an effort to have a comparison for the continent classifier model. This was a question of whether the model would be more accurate with continent prediction if it was trained to be more specific in detecting a country.

3. EXPERIMENTAL SETUP AND RESULTS

3.1 Establishing Baselines

A key component of any deep learning project is establishing baseline performances with which developed models can be compared. The project features models that solve two problems (1) Predicting Country (2) Predicting Continent. For the country classification problem, the dataset produces a simple bias classifier with an accuracy of **25.03%** by predicting the United States of America each time. In addition, for the continent classification problem, the data produces a simple bias classifier with an accuracy of **36.20%** by predicting Europe. These

simple classifiers were elected to be baselines for this project.

3.2 Experiment With Transfer Learning Using EfficientNet

Transfer learning can provide a lot of strong, initial improvements to a model. Due to the short timeline for this project, transfer learning was explored to help facilitate a stronger model. The networks were trained to classify which of the **124** countries the image was taken in.

Because there will be experiments to optimize EfficientNet, a baseline performance for EfficientNet models was created. The architecture was trained using EfficientNetB0 on the original data set of **49,997 224x224** images. For training, **20** epochs, a batch size of **32**, and a validation split of **0.2** was used.

This model resulted in a maximum validation accuracy of **39.28**%.

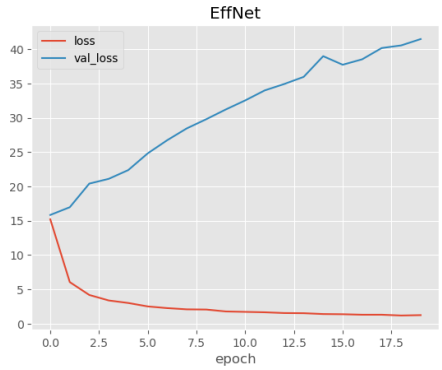

***Figure 2**: Preliminary EfficientNetB0 model loss and validation loss across epochs*

Experiments were run to explore benefits of using smaller image sizes with larger batch sizes. All **49,997** images were used for training, with the image and batch size being variable across the different models. Additionally, only a single dense layer was used as the output layer for **124** nodes.

***Table 1**: Results from using different image and batch size combinations with EfficientNet*

Image Size	BS	Val Acc	BS	Val Acc	BS	Val Acc
128 x 128	60	0.3717	80	0.3677	120	0.3817
64 x 64	150	0.3082	200	0.2942	300	0.2855
32 x 32	300	0.2852	400	0.2668	N/A	N/A

From these results, trading off image resolution for a larger batch size does not result in a more accurate model. For the rest of the EfficientNet experiments in this section, an image size of **224**x**224** was used.

With Keras' EfficientNet architecture being a lightweight and very popular architecture for image recognition, there were a lot of opportunities to improve the accuracy of the model. To improve the neural network, optimizing the batch size during training, comparing the B0, B1 and B2 EfficientNet models, adding additional dense layers, and retraining some of EfficientNet's convolutional layers were all experimented with in order to try and find the best final architecture.

To start, EfficientNetB0 was used to ensure the model could be trained in a reasonable time frame. However, the ability to use a more powerful model such as EfficientNetB1 or EfficientNetB2 could improve the accuracy of the model.

For this experiment, both B1 and B2 were trained using the same layers and hyperparameters in order to directly compare the two architectures. Each network architecture contained a single dense output layer with **124** nodes and softmax as the activation function.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
efficientnetb1 (Functional)  (None, 7, 7, 1280)        6575239

flatten (Flatten)            (None, 62720)             0

dense (Dense)                (None, 124)               7777404

=================================================================
Total params: 14,352,643
Trainable params: 7,777,404
Non-trainable params: 6,575,239
_________________________________________________________________
```

***Figure 3**: EfficientNetB1 summary*

```
Layer (type)                 Output Shape              Param #
=================================================================
efficientnetb2 (Functional)  (None, 7, 7, 1408)        7768569

flatten (Flatten)            (None, 68992)             0

dense (Dense)                (None, 124)               8555132

=================================================================
Total params: 16,323,701
Trainable params: 8,555,132
Non-trainable params: 7,768,569
_________________________________________________________________
```

***Figure 4**: EfficientNetB2 summary*

Additionally, the original **49,997** image data set was used. The number of epochs for both networks was **10**, with a validation split of **0.2** and a batch size of **40**.

The results from training EfficientNetB1 resulted in a maximum validation score of **40.83%**, which is an increase from the EfficientNetB0 model. The maximum validation accuracy when training the EfficientNetB2 model was **38.66**%.

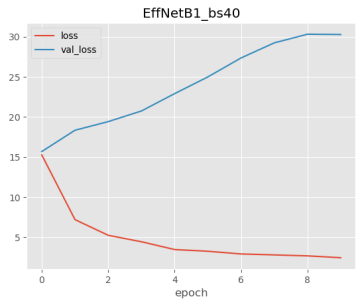

***Figure 5:** EfficientNetB1 loss and validation loss across epochs*

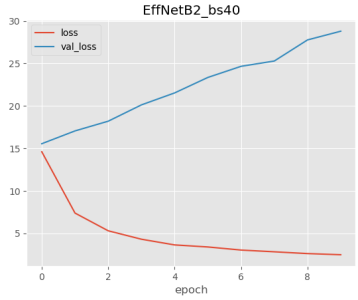

***Figure 6**: EfficientNetB2 loss and validation loss across epochs*

Now that it is determined that EfficientNetB1 shows the most promising results, we tuned the architecture to maximize the accuracy. This iteration of improvements to the network made some of the convolutional layers from the EfficientNetB1 model trainable so the weights could be adjusted. Additionally, more dense layers were added to try to improve the accuracy. The best results from these tests came from changing EfficientNetB1's layers with the indices shown in figure **7** to trainable, with a model architecture described in figure **8**.

```
trainable_layers = [-3, -7, -9,
                    -10, -15, -18,
                    -20, -22, -27,
                    -30, -34, -36]
```

***Figure 7**: EfficientNet layer indices that will be retrained*

```
Layer (type)                 Output Shape              Param #
=================================================================
efficientnetb1 (Functional)  (None, 7, 7, 1280)        6575239

flatten (Flatten)            (None, 62720)             0

dense (Dense)                (None, 150)               9408150

dense_1 (Dense)              (None, 120)               18120

dense_2 (Dense)              (None, 124)               15004

=================================================================
Total params: 16,016,513
Trainable params: 11,836,234
Non-trainable params: 4,180,279
_________________________________________________________________
```

***Figure 8**: Model summary from best EfficientNetB1 model*

This model's training resulted in an accuracy of **44.08**% .

3.3 Experimenting and Fine Tuning VGG16

To determine the best accuracy of VGG16 we experimented with several different methods. We looked at image size, batch size, and epochs. One of the main limitations of the project was sharing resources, leading to issues with run-time and memory. We hoped that after decreasing the image size, we would be able to approach the same accuracy as the larger image by increasing either epochs or batch size.

Our first foray into this query was an attempt to narrow down promising results for further testing. We trained all data points with **3** epochs. These initial results are displayed in the table below.

***Table 2**: Results from using different image and batch size combinations with VGG16*

Image Size	BS	Val Acc	BS	Val Acc	BS	Val Acc
224 x 224	100	0.3290	60	0.3190	N/A	N/A
128 x 128	500	0. 2914	200	0.2749	100	0.2562
64 x 64	1000	0.1736	500	0.1655	N/A	N/A

The results ruled out **64x64** images and any images with fewer pixels as viable options. While the run-time of the fewer pixel images was significantly faster, the accuracy was still below the **224x224** images. The most promising result was that of the **128x128** images with a batch size of **500** which started to approach the **224x224** images with a batch size of **60**. The **128x128** images also completed their average epoch around **4** times faster than the **224x224** images.

Following this, we decided to dive deeper into the highest accuracy result of the **224x224** and **128x128** categories. As previously mentioned, the **128x128** images completed epochs **4** times faster, so we decided to train them with **40** epochs compared to the **10** epochs of the **224x224** images. The results from this dive are depicted below.

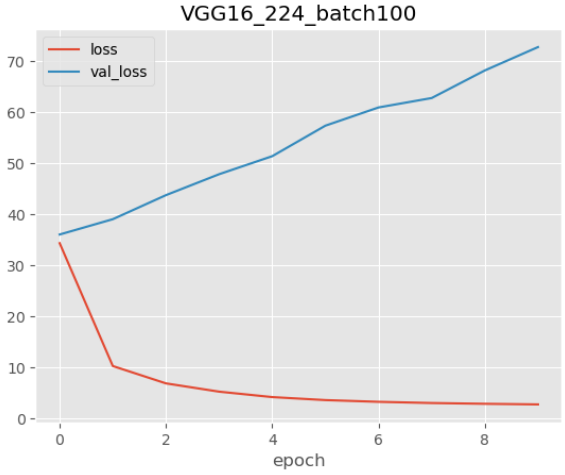

***Figure 9**: VGG16 loss and validation loss across epochs for **244x244** images with a batch size of **100** and **10** epochs*

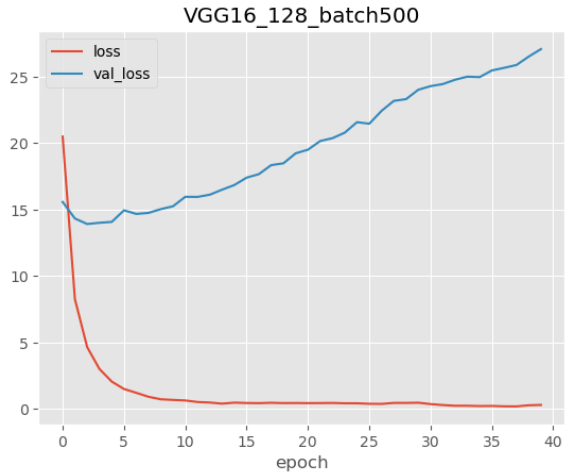

***Figure 10**: VGG16 loss and validation loss across epochs for **128x128** images with a batch size of **100** and **10** epochs*

The **224x224** run resulted in a value accuracy of **34.95%** in a training time of **1.25** hours. The **128x128** run resulted in a maximum value accuracy of **29.74%** in a training time of **1.7** hours. Interestingly, despite the extra epochs, the **128x128** accuracy increased less than the accuracy of the **224x224** run. This led to the conclusion that the **224x224** images should be used for better accuracy even if constraints limit the epochs and batch size of that run.

3.4 Experimenting With A Meta Model

After creating two models with comparable accuracy, the optimized EfficientNet and VGG16 models, we wanted to combine them into a single more accurate model. We hypothesized that a meta model would learn the strengths and weaknesses of each base model and produce better outputs than each individually. We created a meta model to perform the same task, but were trained on the predictions of each optimized model rather than images and labels. We used **50%** of data (**24998** images) for training the optimized EfficientNet and VGG16 models. An unused **30%** (**14999** images) of the data was used as validation that the trained base optimized models predicted on. The meta model was trained on the output of both models from the **30%** validation data. Once the meta model was trained, the remaining **20%** (**10000** images) were used to test the meta model's accuracy.

This initial approach for training the meta model was Logistic Regression, which is well-suited for classification and can effectively handle the output probabilities from the two pre-existing models. In our implementation, the logistic regression model encountered an error due to a mismatch in the expected and actual dimensions of the input data. The error "setting an array element with a sequence" suggested that the shapes of features and labels were not compatible with the requirements of the LogisticRegressionCV model. This incompatibility prevented the successful fitting of the logistic regression model to our data.

Switching to an alternate approach, we chose a machine learning approach using cross validation with a Random Forest Classifier to lead to more accurate and robust country predictions. After successfully fitting the data and testing on the **20%** meta model test data, the meta model had a **36.55%** accuracy. While **11.52%** better than baseline, the meta model's accuracy was **7.53%** worse than the optimized EfficientNetB1 model. Our hypothesis for this attempt was proven incorrect in beating the performance of either individual model.

After the machine learning implementation, we trained a meta neural network on the same output the Random Forest Classifier used for training. We used TenorFlow's Keras for building the network. We chose a Sequential model with multiple Dense layers, leveraging the ReLU activation function for its ability to prevent collapsing of linear layers. The final layer used a softmax activation, suitable for multi-class classification and Adam as an optimizer.

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_19 (Dense)            (None, 2000)              498000

 dense_20 (Dense)            (None, 2000)              4002000

 dense_21 (Dense)            (None, 512)               1024512

 dense_22 (Dense)            (None, 275)               141075

 dense_23 (Dense)            (None, 200)               55200

 dense_24 (Dense)            (None, 124)               24924

=================================================================
Total params: 5,745,711
Trainable params: 5,745,711
Non-trainable params: 0
```

***Figure 11**: This chart shows layers, output shape, and parameters for the meta neural network.*

The end validation accuracy was **42.23%**, only **1.85%** worse than the optimized EfficientNetB1 model. While our hypothesis proved to still be false, attempting another approach allowed us to improve our meta approach to the problem.

3.5 Continent Models

For the continent training we used our basic VGG16 model with a single fully connected layer (Batch Size: **10**, Epochs: **10**).

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 7)                 175623

=================================================================
Total params: 14,890,311
Trainable params: 175,623
Non-trainable params: 14,714,688
_________________________________________________________________
```

***Figure 12**: Trained Continent Model*

Trained on a dataset of **49,997** images with a validation split of **0.2** yield a validation accuracy of **54.44%**.

For the post-processed continent model, in order to be able to test the post-processed value, we trained the model on **40,000** images. Leaving the remaining images to be manually validated. This validation resulted in an accuracy of **52.80%**.

It was interesting to test whether the directly trained or post-processed model would perform better. Ultimately, the validation scores are very similar; however, the directly trained model proved to be the most accurate.

3.6 Custom Haversine Loss Function

The following model was trained on the **49,997** images with a batch size of **100**, **10** epochs, and a validation split of **0.2**.

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14714688
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 2)	50178

Total params: 14,764,866
Trainable params: 50,178
Non-trainable params: 14,714,688

Figure 13*: This chart shows layers, output shape, and parameters for the haversine loss function model.*

The training resulted in an average validation loss of **10,010** km. This means that on average, our guess was **10,010** km off from the actual. When compared to the **40,075** km circumference of the earth, this is a reasonable result. In addition, after the **10** epochs the model boasted a validation accuracy of **70.80%.**

This work occurred in the final week of the project and was slightly separate from the existing work. As a result, it would be worthwhile to continue experimenting with the custom loss function and regression model. Also, given the differences between regression and classification models, it is difficult to compare this model's proficiency with any of the classifiers.

4. DISCUSSION

4.1 Analysis of Incorrect Model Predictions

In analyzing the wrong predictions made by models, we noticed wrong decisions were still good predictions. The images below show the actual prediction from the model and the correct labels for the images. While the model was not correct, the prediction was very close to the correct answer when incorrectly predicting "United States" for "Canada."

Figure 14*: The model incorrectly predicts the "United States" for an image of a road, with green nature in the background, and a clear blue sky.*

Figure 15*: An image of a road, with green nature in the background, and a clear blue sky with "Canada," the image's correct label written above.*

See Figure A1 and Figure A2 in the appendix for more examples. These are mistakes a human would likely make as well. Understanding how the neural network makes predictions explains these small mistakes. With the softmax function, each class is given a probability for how likely that image corresponds to the class, forming a probability distribution. The incorrect prediction likely had high percentages for the correct answer, but not quite as high as the incorrect prediction itself. As the model is trained on more data, those correct but slightly lower probabilities are increased to the highest value, making the overall accuracy improve. This process is quite similar to how humans play Geoguessr, going back and forth in their head between the most likely countries, having an understood likelihood of each, before making the prediction.

4.2 Discussion on Image and Batch Size Tradeoff

In the beginning, one of the issues we continued to run into was running out of memory on

Rose-Hulman's shared GPU server. We decided to experiment with using smaller images to save memory and potentially increase accuracy.

However, after looking at the results from both VGG16 and EfficientNet, it is clear that the tradeoff is not worth it for the Geoguessr domain.

Networks trained on smaller dimensioned images provided less accurate results despite having larger batch sizes. These results make sense when you think about the level of detail needed to distinguish between details in the landscape. For example, most trees are green, but having enough detail for the shape of the tree could provide important information to the network if that tree is more common in certain countries.

4.3 Discussion on Experimental Results

From experimenting with different models, we found that the most accurate model tested was an EfficientNetB1 model, with **12** trainable convolutional layers, **3** fully connected dense layers with **150**, **120** and **124** nodes. The model was trained across **20** epochs, with a batch size of **75** and a validation split of **20%**. This model achieved a validation accuracy of **44.08**%.

One interesting thing to discuss is the performance of EfficientNetB1 compared to EfficientNetB2. For this domain, EfficientNetB1 was an additional **2.17**% more accurate than EfficientNetB2. This is an interesting result, as EfficientNet is advertised to provide higher accuracy with larger versions, however, the tests we ran showed that the use of EfficientNetB2 could decrease the accuracy of the final model.

5. CONCLUSIONS AND FUTURE WORK

After experimenting with a variety of architectures, hyperparameters and approaches, we were able to improve on the baseline by **19.05**% as seen in the appendix table A3. The best model tested for this domain was shown to be an EfficientNetB1 model on the highest resolution image (**224**x**224**). It was also shown that retraining the last 12 convolutional layers of the EfficientNetB1 architecture improved the accuracy of the network.

The combination of different models proved to not be particularly fruitful; however, using the predicted coordinates from the regression model as features in addition to the raw image for a classification model could be effective. In addition, in the future it could prove powerful to utilize the custom haversine loss function with sparse categorical cross entropy for a classification model. Applying weights to these loss functions would allow for control over how influential each loss function is on the final result. Ultimately, the custom haversine loss function does not place value on whether the predicted country is correct, it is only concerned with the proximity of the guess. Inclusion of sparse categorical cross entropy as well would shift some emphasis onto a correct country prediction.

Furthermore, for future experimentation, in an effort to expand the dataset, photos not exclusively from google street view could be used. These photos could then be labeled according to their geotags. This would not only increase the dataset size, but also broaden the model's use cases outside of just applications to the Geoguessr web game.

Finally, a major change that could be pursued is the use of manual feature extraction. Training the model to make predictions in a similar process to a human could prove fruitful. This would require the training of models that would detect and classify road signs, road markings, car makes, and more. Ultimately these would result in manual features that could be fed into the final model.

6. KEY CHALLENGES AND LESSONS LEARNED

We had sufficient data for the scope of this project; however, as with any deep learning project, more data would have been better and resulted in a stronger final model. In addition, the intention of the project was to create a regression model as the stretch goal; however, since the data source only had the country marked as a feature we were restricted to a classifier.

In addition, there were more challenges with the custom loss function. Firstly, working with TensorFlow Tensors was a challenge at times because they were just references and therefore did not contain concrete values. The initial vision for the

custom loss function was to still train a classifier but to define the loss as the haversine distance between the centroid of the predicted country and actual country of the image. Due to the Tensors, there were many problems with the dimensionality and not being able to use any of the values to index the array storing country centroid data. This led to the final approach of training a regression model which took country centroids as features and would predict a longitude and latitude for the image. In addition, a key challenge was organizing file structures and having clearly defined and documented saved models.

During the beginning of the project, we faced technical difficulties when training the EfficientNet models. To start, we ran out of GPU memory when trying to train the EfficientNet model. These issues came from using Rose-Hulman's shared GPU server *Gebru* too close to the milestone deadlines. As the submission deadline approached, the server would become more and more used. We learned to be more aware of the current GPU usage on the shared Rose-Hulman servers, and to set reasonable expectations for the batch size during training.

Challenges also arose when saving and utilizing checkpoints during EfficientNet training. Additionally, the EfficientNet model was unable to be saved because a Keras object called an "EagerTensor" was not able to be serialized. The model's training, prediction and scoring worked as expected. We researched the issue and found that there were others who experienced the same challenges as us, but there were no solutions left from the collaborators. We attempted to save the model to multiple different formats including JSON, keras and h5. However, none of these formats fixed the issue, and one of the project advisors, Dr. Boutell was also stumped on this challenge.

7. REFERENCES

[1] GeoGuessr. Available: https://www.geoguessr.com/. [Accessed: 11/10/2023].

[2] Ubitquitin, "Geolocation GeoGuessr Images 50k," Kaggle Dataset. Available: https://www.kaggle.com/datasets/ubitquitin/geolocation-geoguessr-images-50k. [Accessed: 11/10/2023].

[3] Komsitr, "Country-Centroid," GitHub repository. Available: https://github.com/komsitr/country-centroid. [Accessed: 11/10/2023].

[4] Keras. "EfficientNet," Keras API documentation. Available: https://keras.io/api/applications/efficientnet/. [Accessed: 11/10/2023].

[5] "Haversine formula," Wikipedia. Available: https://en.wikipedia.org/wiki/Haversine_formula. [Accessed: 11/10/2023]

8. APPENDIX

Figure A1: The model made predictions on these images. These are all incorrect predictions.

Figure A2: These are the corresponding images to Figure A1 but with the correct label above.

Table A3: Country Prediction Models and Accuracies in order of implementation

Model Name	**Image Size**	**Batch Size**	**Accuracy (correctness)**
Simple Bias Classifier	N/A	N/A	25.03%
Preliminary VGG16 (0.2 validation split)	224x224	100	33.90%
Optimized VGG16 (0.8 validation split)	224x224	100	34.95%
Optimized VGG16 (0.8 validation split)	128x128	500	27.94%
Preliminary EfficientNetB0	224x224	32	39.28%
Preliminary EfficientNetB1	224x224	40	40.83%
Preliminary EfficientNetB2	224x224	40	38.66%
Optimized EfficientNetB1	224x224	75	44.08%
Meta Random Forest Classifier	224x224	N/A	36.55%
Meta Neural Network	224x224	75	42.23%

Table A4: Continent Prediction Models and Accuracies in order of implementation

Model Name	**Image Size**	**Batch Size**	**Accuracy (correctness)**
Simple Bias Classifier	N/A	N/A	36.20%
Pretrain Classifier	224x224	100	54.44%
Post-Processed Classifier	224x224	100	52.80%