APS360: GEOSCOUT - PHOTO GEOLOCATION WITH RESIDUAL NEURAL NETWORKS

James Li
Student# 1006730619
jyjames.li@mail.utoronto.ca

Nicholas Mutlak
Student# 1008339371
n.mutlak@mail.utoronto.ca

Mirai Shinjo
Student# 1007835210
mirai.shinjo@mail.utoronto.ca

Ananya Saigal
Student# 1008106354
ananya.saigal@mail.utoronto.ca

ABSTRACT

The team has created a Residual Neural Network trained to predict the closest city and precise country from a street view image. The ability to estimate the geolocation of a place from a particular image has several applications, particularly when GPS or IP address information is not available. The process and considerations in creating this AI model have been outlined in this document —-Total Pages: 9

1 INTRODUCTION

Google Street View images showcase a variety of visual features, from surroundings to city architecture, offering insights into the world and forming the basis of the game GeoGuessr, where players guess where they are based on these images. Identifying these features manually and using them to find the city and country of origin is difficult due to the large number of potential locations and subtle differences between places. There is a need for sophisticated deep-learning models that can improve the accuracy of geographic predictions when GPS and IP address information is unavailable. This project aims to develop a robust model for automatic location detection from Google Street View images, using deep learning techniques to predict the country and the nearest city the image is located in. The team will be focusing on North and South America, Middle and South East Asia, Europe, and Oceania because these countries have coverage in Google Street View imagery.

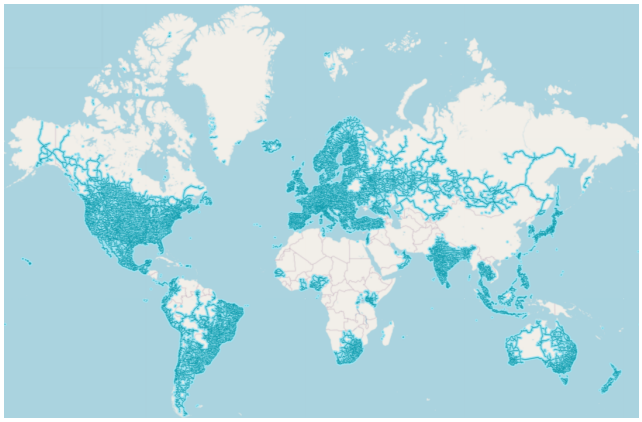

Figure 1: Map of Google Street View Coverage[1](SV-Map Contributors, 2024)

2 ILLUSTRATION

In our model, Google Street View images are paired with their corresponding coordinates to form our dataset. The coordinates are converted into S2 cell IDs (a type of area code) to form classes. Images are first transformed into a tensor and then passed through multiple residual blocks (convolutional layers with built-in skip connections) of a ResNet model. The model processes the data through these blocks until the final logits are produced, which are used to predict an S2 cell ID, along with the closest city and country the area code resides in. The following illustration shows the general architecture of our model.

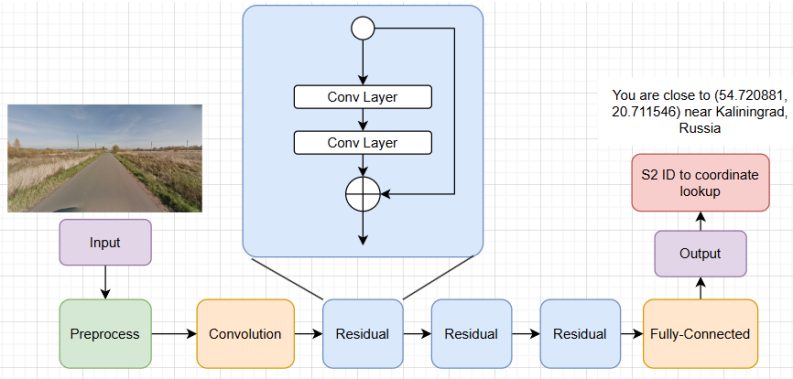

Figure 2: ResNet architecture of geolocation model, Image: GeoGuessr

3 BACKGROUND & RELATED WORK

Significant research has been done in geolocation using neural networks. Although the standard approach is to train a CNN using the ResNet architecture, by examining 5 different papers on this topic, several techniques can be uncovered that greatly enhance model performance.

The "Geolocation Estimation of Photos using a Hierarchical Model and Scene Classification" paper describes the method of using hierarchical grid sizes and employing the ResNet model as a baseline for feature extraction. The approach also incorporates a visual scene classifier that labels an image with multiple keywords, such as "urban" or "rural," based on its content (Müller-Budack et al., 2018).

Another notable method is "PIGEON: Predicting Image Geolocations," which uses polygonal grid shapes representing distinct geographic regions. Instead of predicting just one location, this model predicts a set of likely cells and refines the prediction using a clustering algorithm (Haas et al., 2024).

The "PlaNet" model uses a CNN based on the Inception architecture. The model produces a one-hot encoded vector corresponding to the geographic S2 cell containing the geotag of an image. One feature of this model is the use of adaptive grid sizes—larger cells for sparsely populated areas and smaller cells for denser regions (Weyand et al., 2016).

The "Interpretable Semantic Photo Geolocation" paper presents a method that takes coordinates and converts them into geographic regions as text. These region names are then classified into a hierarchy of smaller areas, providing a more precise geolocation prediction. The model uses both EfficientNet-B4 and ResNet50 architectures for feature extraction (Theiner et al., 2022).

Lastly, the "DeepGeo" model integrates elements from "Deep Residual Learning for Image Recognition" He et al. (2015). The first integration involves concatenating views from different cardinal directions of a Street View panorama. The second integration uses a shared CNN to generate a low-dimensional summary for each direction. A third integration strategy chooses the direction with the best geographic representation (Suresh et al., 2018).

[1] All coverage of Google Street View is covered by S2 Cells as well

4 Data Processing

The data processing component of this project is comprised of three phases. The first phase consists of processing a dataset on Kaggle, the second phase focuses on generating a custom dataset to address flaws in the Kaggle dataset, and the third phase further processes the custom dataset to work well for a classification problem.

4.1 Phase One

In phase one, we processed a dataset of 25k images containing both official and unofficial Street View images (refer to Figure 3) (Kaushal, 2023). The processing stage consists of removing duplicate locations and converting latitude and longitude pairs into S2 cells with cell level 3, splitting the world into 384 cells (S2Geometry). We performed a random horizontal flip and rotation between -10 and 10 degrees, resized them from 640 x 640 to 224 x 224 pixels, and finally converted the image into a tensor. Using a custom Dataset class, we populated training, validation, and test loaders with an 80:10:10 split.

Figure 3: Sample Images of 25k Dataset

Upon evaluation of the images within the processed 25k dataset, most images did not capture unique attributes of the geographic area as they were either fully obscured by vegetation, showed little of the surrounding landscape, or were heavily blurred. The misaligned and inconsistent images made the model unable to learn general geographic features, let alone country-specific infrastructure. Furthermore, the dataset was heavily skewed towards rural locations. An attempt was made to correct this by using the World Cities Dataset in conjunction with the Google Street View Static API to generate around 7k urban locations to offset the imbalance. However, even in the urban subset, the issue of a misaligned image composition persists (simplemaps, 2024).

4.2 Phase Two

In phase two, we focused on generating a custom dataset using MapGenerator (RollinHill). MapGenerator is a tool designed for GeoGuessr map creators, enabling them to generate JSON arrays containing the latitude, longitude, heading, and panorama ID of Street View locations. Its web interface allows users to define regions where locations should be generated via polygonal drawings. Additionally, the tool uses AI to determine the camera panning, specified as the heading field in each JSON object, such that the image generated is aligned with the road (refer to Figures 4 and 5).

Figure 4: Image with default panning (facing North)

Figure 5: Image of the same location aligned with road

Using MapGenerator, we drew 39 polygons corresponding to 39 level 3 S2 cells that were chosen based on coverage and geographic distinctness (refer to Figure 6). We then passed each coordinate and heading in the generated JSON file to the Google Street View Static API to download the image

of the given location. Each coordinate was also appended to a CSV file. However, within this raw dataset are a small number of images not belonging to any of the 39 chosen S2 cells, which are artifacts of imperfect polygon drawings in MapGenerator. To better organize and clean this dataset, we removed these extra images.

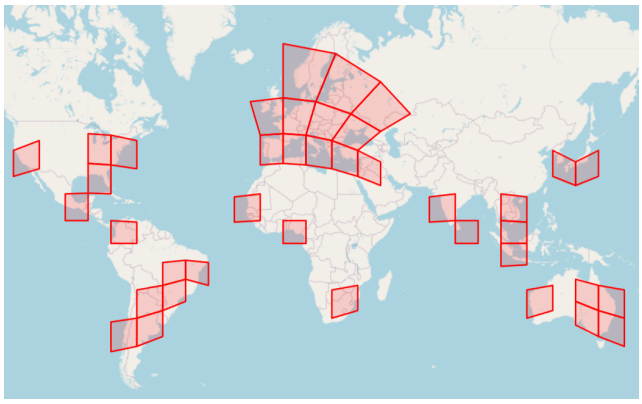

Figure 6: S2 cells of images in the custom dataset

4.3 Phase Three

The images from Phase 2 were placed into folders of their respective S2 Cell IDs, which will become the images' labels, with approximately 500 images for each of the 39 classes. The dataset was converted into data loaders with the same 80:10:10 split for training. However, we noticed for some classes, either their images were very similar to other classes, causing high precision and recall confusion, or had no distinct properties for our model to learn, resulting in low accuracy. We identified and pruned out 4 classes with the lowest F1 Score performance, resumed testing, and later pruned another 3 classes. The pruned classes total 5 from Europe, 1 from the U.S.A. and 1 from Mexico. Pruning to simplify our problem space stopped here because from this point, the F1 Scores of all the other classes from the model's peak performance were above 0.5 and removing them would not help the model in terms of class confusion or inability to learn distinct properties. The final dataset, dubbed Streets3, with 32 classes contains 16,750 custom-generated images in total.

5 Baseline Model

To have a reference point for both our baseline and primary models, it should be noted that with 32 S2 cell ID classes in the Streets3 dataset, randomly guessing the class of an image would yield an expected accuracy of $\frac{1}{32} = 3.125\%$.

Our baseline model is a simple CNN with 3 convolutional layers and 2 fully connected layers, similar to the LargeNet model from Lab 2. For each convolutional layer, we apply a ReLU activation function with a max pool operation of kernel size and stride of 2. We used the Streets3 dataset with 32 classes, which is reflected by our model's FC layer having a final output layer of a tensor of size 32 (refer to Appendix A). The model in total has 9,729,824 parameters comparable to the ResNet-18 architecture's size, which contains 11,689,512 parameters (refer to Figure 7).

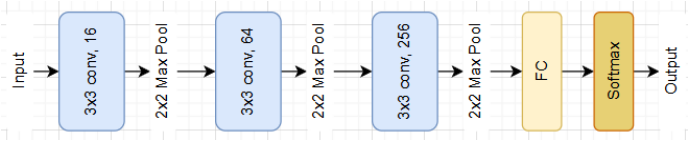

Figure 7: CNN Baseline Architecture

To train this model, CrossEntropyLoss and Adam Optimizer were used with a learning rate of 0.001 and batch size of 128. It should also be noted that after some experimentation, we realized that the model performed better on the validation set if there were no augmentations on the training data.

The best results achieved with this model after 15 epochs, were a training accuracy of 50.13%, validation accuracy of 37.07% and test accuracy of 37.19%. Refer to Figure 8 for the full training and validation curves. This shows that the simple use of machine learning with minimal tuning is 11.9 times better than random guessing.

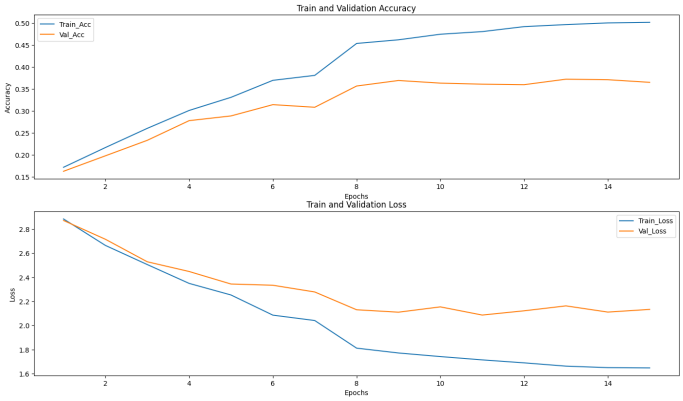

Figure 8: CNN Baseline Train vs Validation Curves

Upon examining the confusion matrix and classification report, the best F1 Score of all the classes was 0.73 for class 1062849512059437056, which corresponds to an area in North Western Africa off the coast of Senegal (refer to Appendix B). The model may be performing well by identifying the orange-coloured sands and flat landscape with occasional shrubbery (refer to Figure 9).

Figure 9: CNN Best Performing Class Examples

6 ARCHITECTURE

For image classification problems we decided to experiment with different Residual Network (ResNet) models to train residual blocks to achieve better gradient propagation, which enables training much deeper networks (Akbari, 2024). This approach aligns well with our project goals by providing greater flexibility and robustness. Previous research in this area has shown that ResNet models have outperformed VGG in terms of feature extraction (Salehalwer, 2023). The advantage of using ResNet is that it can alleviate the problem of vanishing gradients in a neural network.

The final architecture, shown in Figure 10, is a ResNet-18 model with 11,689,512 parameters. We also experimented with ResNet-34, incorporating dropout and freezing layers, however, they resulted in low validation, training accuracy, or overfitting. ResNet-18 demonstrated superior generalization on our dataset. Its smaller number of parameters reduces the risk of overfitting, making it less likely to memorize the training data and more likely to perform well on unseen data (Zhang, 2019).

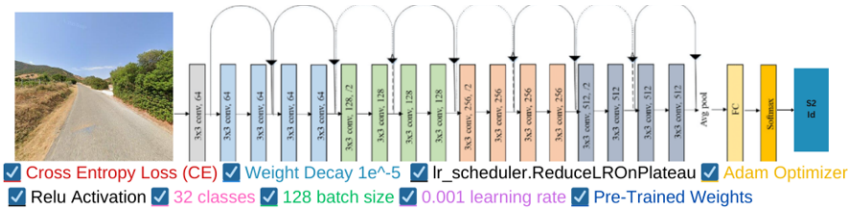

Figure 10: Architecture and hyper-parameters of `resnet-18`

Using a pre-trained ResNet model offered significant advantages, such as leveraging knowledge learned from larger and more diverse datasets. This not only saved time and computational resources but also provided a robust foundation for fine-tuning the model to suit our specific dataset.

Our implementation of ResNet-18 initially with 39 classes achieved its best results with 11 epochs, a batch size of 128, and a learning rate of 0.001. The model uses the Adam optimizer, which combines the benefits of momentum and adaptive learning rates for efficient training. To further refine the learning process, we employed a ReduceLROnPlateau scheduler, which dynamically reduces the learning rate when the model's performance on the validation set plateaus (PyTorch, 2023). The team used a weight decay of $1e^{-5}$, a regularization technique aimed at preventing overfitting, as part of the optimization process. By penalizing large weight values, weight decay helps to constrain the model's complexity and encourages it to focus on learning essential patterns.

We experimented with a custom loss function by combining CrossEntropyLoss with another loss function inspired by Haversine loss (refer to Figure 11). Haversine calculates the great-circle distance between two points on a sphere, making it suitable for geospatial applications (Prakhar7, 2022). Instead of relying on precise geographic coordinates, we tailored this approach to our dataset by calculating distances between the centroids of the actual and predicted S2 cells. Our hyperparameters remained the same, but for the custom loss function, we incorporated a normalized Haversine-inspired term with a weight called beta set to 1 (refer to Appendix C). This combination did not enhance the accuracy, which was 0.4 - 0.5% below training with CrossEntropyLoss alone. Increasing the beta term further caused the validation accuracy to drop, while decreasing the beta term increased the accuracy. As a result, we removed the haversine term from our loss function and continued to train with just CrossEntropyLoss.

At this point we swapped to using the Streets3 dataset with 32 classes combined with the hyperparameters of ResNet18 that worked best on the set of 39 classes. Refer to Appendix D for the exact architecture setup code.

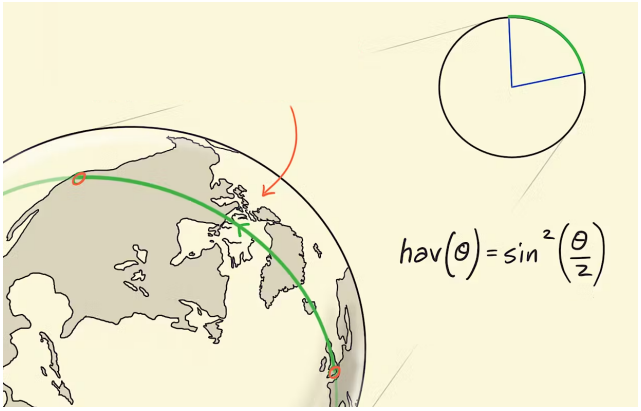

Figure 11: The Haversine formula (Prakhar7, 2022).

7 RESULTS

This section outlines the quantitative and qualitative results of our best models.

7.1 QUANTITATIVE

To evaluate the effectiveness of our model, we analyzed its performance on training and validation datasets. As highlighted in the Architecture section, each model demonstrated improvements in accuracy through iterative refinements. After tuning hyperparameters such as epochs and batch size using the 39-class dataset, we froze these hyperparameters across subsequent experiments. Table 1 provides a comparison of all the models discussed in the Architecture section, highlighting the number of classes, the type of loss function, and their respective training and validation accuracies.

Table 1: Comparison of `resnet-18` Model Performance

Number of Classes	Loss Type	Train Accuracy	Validation Accuracy
39	CE	100%	54.4%
35	CE + Custom_loss	100%	60.4%
35	CE	100%	60.9%
32	CE	100%	64.4%

Notably, the combination of CrossEntropyLoss and the custom Haversine-inspired loss function achieved competitive performance, however, CrossEntropyLoss ultimately performed better. While training accuracy remained at 100% in all models, validation accuracy improved as the number of noisy or ambiguous classes decreased.

Comparing the impacts of purging from 35 classes to the 32-class Streets3 dataset using CrossEntropyLoss, we observe that both models exhibit relatively low noise in their validation and training accuracy and loss curves, indicating that the models are learning stably and are less likely to overfit, improving generalization performance. The model trained on Streets3 achieved a lower training loss of 0.0071 and validation loss of 1.360, along with higher validation accuracy of 64.4% (refer to Figure 12). Specific training and validation curves for the 35-class models can be found in Appendix E.

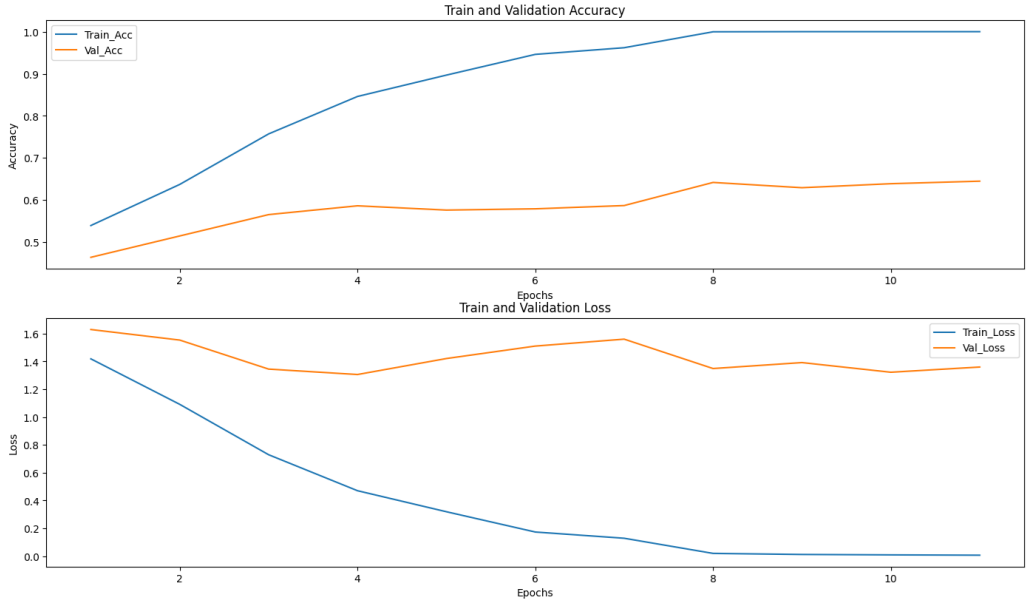

Figure 12: ResNet on Streets3 Train vs Validation Curves

7.2 Qualitative

Upon examining the confusion matrix and classification report, the highest F1 Score among all the classes was 0.87 for class 1062849512059437056, which corresponds to North Western Africa (refer to Appendix F). This is the same class where our baseline also performed well, and as such for the same reasons the model may be performing well due to its ability to identify features such as the orange-coloured sands along the roadside and occasional shrubbery (refer to Figure 13).

Figure 13: ResNet-18 Best Performing Class Examples

7.3 Model Evaluation On New Data

It is important to note that acquiring new unseen data in person would be unreasonable for the team, as it would require us to travel to each of the 32 regions around the globe and obtain a substantial number of street view images that are aligned in the same way as our Streets3 dataset. That said, because we generated our images ourselves, we reserved 1675 images (10%) of our total dataset in the test_loader which our model would never see during training. To ensure the data never got mixed up between training, validation, and the test_loader, we made sure to set consistent seeds for all sources of randomness.

After running our model on the test_loader, it yielded an accuracy of 63.1%. Compared to the baseline model's accuracy on the test_loader of 37.19%, the ResNet model performs 1.7 times better than the baseline and 20.2 times better than random guessing. Compared to other modern solutions that cover far more of the globe in scope, but train on millions of images in turn, the model stacks up in accuracy on the region level too (comparable to S2 cells). On a test set of 2k images, PIDGEOTTO, debuted in 2023 and trained on 9M images, has a region-level accuracy of 63.3%. CPLaNet, debuted in 2018 and trained on 30.3M images, has a region-level accuracy of 46.6%, and Im2GPS, debuted in 2017 and trained on 34M images, has a region-level accuracy of 47.7% (PapersWithCode, 2024).

The team also created an interactive tool that allows users to evaluate the performance of the model developed on unseen data from our test set (refer to Appendix G). The user can run the model on an image from the test set and receive the predicted location as well as the confirmed real location in 'City, Country' format. Additionally, an accompanying map displays the center of the predicted regions for visual clarity (if the model has predicted correctly, which it does for the majority of the time, the two areas on the map show).

8 Discussion

Our model performs exceptionally well given the complexity of the task with a relatively small dataset of 16,750 images split into 80% training, 10% validation and 10% reserved for our final evaluation with 32 unique classes. Unlike datasets containing handwriting or flowers, where specific objects are consistently present, our dataset contains landscapes with no guarantee of distinctive features.

An interesting and somewhat surprising finding was that the smallest ResNet architecture available in PyTorch, ResNet18, ended up being the best fit for our project. We initially considered using larger models like ResNet-50, especially since we anticipated having a dataset size of around 20k. However, ResNet-18 outperformed larger models in our case. This may be due to the nature of our data having no obvious features across many images that would benefit from a deeper network, and that it would likely not overfit in a smaller architecture and additionally our dataset, while substantial, is small in comparison to the vast number of images available from Google Street View. ResNet-18 architecture, has fewer parameters (11,689,512), and this may generalize better. We believe that with more training data, larger models such as ResNet-152 could achieve even better performance.

Throughout the project, we experimented with many different combinations of hyperparameters. We tried using different optimizers such as SGD and Adam, experimented with random dropouts, tried out different architectures of fully connected layers, tried out freezing some layers of the model instead of unlocking all the layers, and developed our own custom loss function by combining

Haversine formula/concept with CrossEntropyLoss to penalize predictions that are geographically further from the ground truth labels. Additionally, we tried adjusting batch sizes to as small as 16 to as large as 2048. We also applied data augmentation techniques. Despite these efforts, these adjustments had relatively small effects on the model's performance. However, what made the most significant impact was the quality and quantity of data. As we approached the end of the project, we refined our dataset by enforcing consistency in aspects like forcing the image to face the street straight-on and doubling the number of images. This led to the most substantial improvement in model performance by far.

9 Ethical Considerations

9.1 Privacy and Copyright

When working with visual data, privacy concerns are paramount, especially when the data involves images from public sources such as Google Street View (Andy Nagy, 2022). Google Street View images often contain private information, such as identifiable faces, license plates, or property details. It is crucial to ensure that all personal identifiers are blurred, as failure to do so could lead to violations of individuals' privacy rights (Klingler, 2023). This model could be misused to obtain the locations of individuals, raising privacy and security concerns.

9.2 Limitations to the Model

The model is limited by the diversity, quality, and quantity of the data it is trained on and may struggle with locations that are visually similar but geographically distinct, as it is inherently pattern-based and may not account for subtle contextual differences (InterProbe Information Technologies, 2024). A concern is the potential bias in the model due to the lack of geographic diversity in the training data. Google Street View and online dataset images are not equally distributed around the different regions of the world. More developed urban areas have greater coverage than rural or remote regions (Mapsted, 2023). This could result in an unreliable or unfair model that performs well in highly mapped areas but struggles in underrepresented regions.

10 Project Difficulty / Quality

GeoGuessr has rapidly evolved over the years from what once was just a fun novelty game to pass time, to now having a strong player base that spends a lot of time practicing and researching to make the best guesses possible in competitive settings. The average accuracy of a player is 10,556 out of a maximum of 25,000 points or 42% (**?**). Even at the highest level in the GeoGuessr World Cup, professional players may still guess more than 10k km away from the actual location, simply due to the arbitrariness of the given Street View images (refer to Appendix H).

Bridging the gap between GeoGuessr and deep learning prediction presents numerous challenges and complex aspects. Many places around the world share similar visual features and elements. These commonalities mean that the model developed must be able to distinguish between locations based on small and subtle differences. Another significant challenge is that a single location can look very different based on various conditions. For instance, differences in the time of day, seasons, and weather can give a completely contrasting appearance of a place, which adds difficulty in training the model developed for generalization with high accuracy. Furthermore, data collection is challenging because assembling a comprehensive dataset that includes images from various locations with an equal distribution for each class is difficult. Even if the distribution is balanced, the data within a class might be skewed towards certain features or conditions that are unwanted (biased), while others may have many different variants (not biased). This imbalance can negatively impact the model's ability to learn representative features for each location. Given these challenges, our project is more difficult than one might initially expect. Despite these obstacles, our model performs well at 64.4% accuracy on unseen data with a baseline performance of 37.2%.

REFERENCES

Kevin Akbari. Understanding residual networks (resnet) in deep learning. *Medium*, 2024. URL https://akbarikevin.medium.com/understanding-residual-networks-resnet-in-deep-learning-ee069215afa3.

Simon Du Perron Andy Nagy. Key takeaways for businesses when using location tracking technologies. *BLG*, 2022.

GeoGuessr. Best moments of the geoguessr world cup '24, 2024. URL https://www.youtube.com/watch?v=iQiBwgcodk0.

Lukas Haas, Michal Skreta, Silas Alberti, and Chelsea Finn. Pigeon: Predicting image geolocations, 2024. URL https://arxiv.org/abs/2307.05845.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

InterProbe Information Technologies. Ai and geopositioning: The future of location prediction. *Medium*, 18, 2024.

Ayush Kaushal. Streetview image dataset, 2023. URL https://www.kaggle.com/datasets/ayuseless/streetview-image-dataset. Accessed: 2024-10-13.

Nico Klingler. Face blur for data privacy in deep learning. *viso.ai*, 2023.

Mapsted. Location-based tech & artificial intelligence: A match made in heaven, 2023.

Eric Müller-Budack, Kader Pustu-Iren, and Ralph Ewerth. Geolocation estimation of photos using a hierarchical model and scene classification. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (eds.), *Computer Vision – ECCV 2018*, pp. 575–592, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01258-8.

PapersWithCode. Im2gps benchmark (photo geolocation estimation), 2024. URL https://paperswithcode.com/sota/photo-geolocation-estimation-on-im2gps. Accessed: 2024-11-27.

Prakhar7. Haversine formula to find distance between two points on a sphere. *geeksforgeeks*, 2022.

PyTorch. Reducelronplateau - pytorch 2.5 documentation. *PyTorch*, 2023.

RollinHill. Mapgenerator. URL https://map-generator.vercel.app/.

S2Geometry. S2 cell statistics. URL http://s2geometry.io/resources/s2cell_statistics. Accessed: 2024-10-13.

Salehalwer. Geoguessr-inspired exploration of cnns: Predicting street view image locations. *Medium*, 2023.

simplemaps. World cities database, 2024. URL https://simplemaps.com/data/world-cities. Accessed: 2024-10-28.

Sudharshan Suresh, Nathaniel Chodosh, and Montiel Abello. Deepgeo: Photo localization with deep neural network, 2018. URL https://arxiv.org/abs/1810.03077.

SV-Map Contributors. Sv-map: Interactive street view map, 2024. URL https://sv-map.netlify.app. Accessed: 2024-10-03.

Jonas Theiner, Eric Müller-Budack, and Ralph Ewerth. Interpretable semantic photo geolocation. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 750–760, January 2022.

Tobias Weyand, Ilya Kostrikov, and James Philbin. *PlaNet - Photo Geolocation with Convolutional Neural Networks*, pp. 37–55. Springer International Publishing, 2016. ISBN 9783319464848. doi: 10.1007/978-3-319-46484-8_3. URL http://dx.doi.org/10.1007/978-3-319-46484-8_3.

Lisa Zhang. Preventing overfitting, 2019. URL https://www.cs.toronto.edu/~lczhang/360/lec/w05/lec08.pdf.

APPENDIX A: CONVOLUTION NEURAL NETWORK BASELINE MODEL

Custom CNN model developed to serve as the baseline model for the primary model:

```
class CNN_Model(nn.Module):
    def __init__(self,kernel_steps=[3,3,3],name="CNN_Model"):
        # changed model for dynamic kernel size for later training

        super(CNN_Model, self).__init__()
        self.name = name
        # no padding needed since black borders exist from rotation
    augmentation
        self.conv1 = nn.Conv2d(3, 16, kernel_steps[0], 2)
        self.conv2 = nn.Conv2d(16, 64,  kernel_steps[1], 2)
        self.conv3 = nn.Conv2d(64, 256, kernel_steps[2], 2)
        self.pool = nn.MaxPool2d(2, 2)

        # for dynamic kernel sizing, compute output size for FC layer
        self.x1 = math.floor(((224 - kernel_steps[0])/2 + 1)/2)
        self.x2 = math.floor(((self.x1 - kernel_steps[1])/2 + 1)/2)
        self.x3 = math.floor(((self.x2 - kernel_steps[2])/2 + 1)/2)

        self.fc1 = nn.Linear(256 * self.x3*self.x3, 4096)
        self.fc2 = nn.Linear(4096, 32)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 256 * self.x3 * self.x3)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

APPENDIX B: CNN CONFUSION MATRIX AND CLASSIFICATION REPORT

Confusion matrix of baseline model:

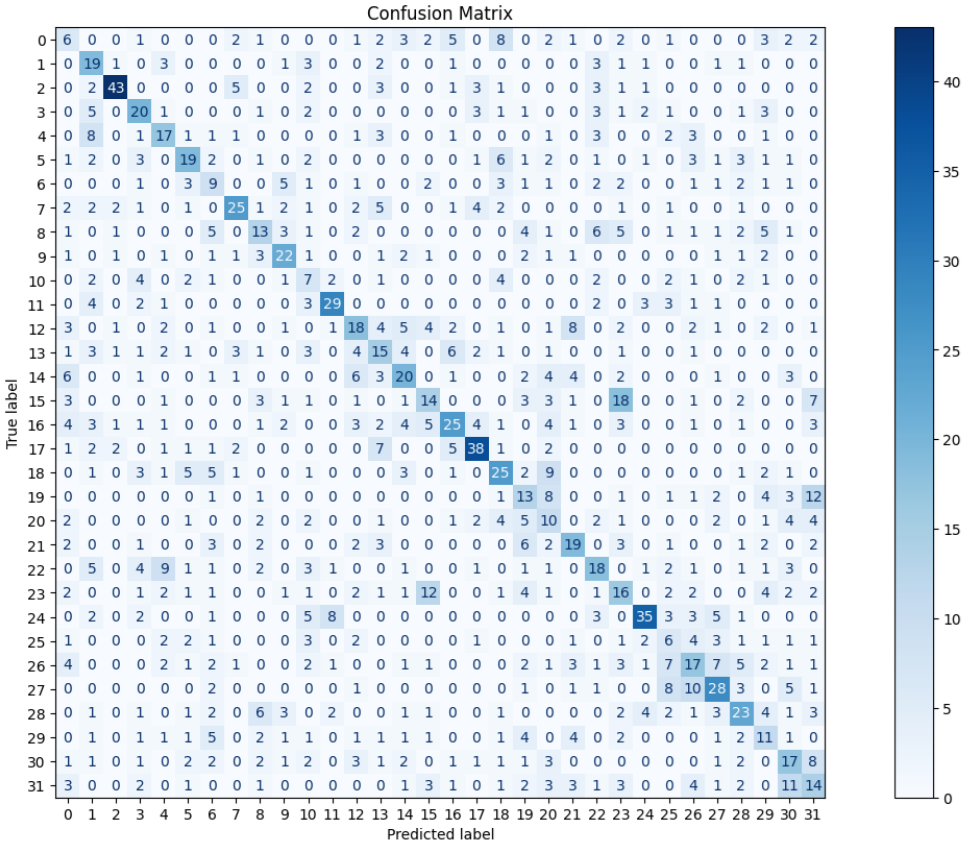

Figure 14: CNN Baseline Confusion Matrix

Classification report of baseline model:

	precision	recall	f1-score	support
10250192751895248896 (0)	0.14	0.16	0.15	37
10610480722084888576 (1)	0.30	0.42	0.35	45
1062849512059437056 (2)	0.81	0.74	0.77	58
10682538316122816512 (3)	0.38	0.41	0.40	49
10718567113141780480 (4)	0.34	0.33	0.33	49
10790624707179708416 (5)	0.48	0.40	0.44	52
10826653504198672384 (6)	0.19	0.27	0.22	33
11709359031163289 60 (7)	0.60	0.50	0.54	50
14951950762870046 72 (8)	0.35	0.27	0.30	56
15312238733059686 40 (9)	0.50	0.51	0.51	43
22157710166662840 32 (10)	0.15	0.24	0.18	29
31525197391593472 00 (11)	0.68	0.55	0.61	55
33326637242541670 40 (12)	0.34	0.32	0.33	53
35488365063679508 48 (13)	0.29	0.30	0.29	54
35848653033869148 16 (14)	0.38	0.37	0.37	52
38370668825196625 92 (15)	0.30	0.25	0.27	57
42333836497282662 40 (16)	0.50	0.35	0.41	75
43054412437661941 76 (17)	0.63	0.61	0.62	62

```
4701758010974797824 (18)       0.42      0.42      0.42        64
5062045981164437504 (19)       0.24      0.23      0.23        56
5098074778183401472 (20)       0.16      0.29      0.21        35
5206161169240293376 (21)       0.43      0.42      0.42        48
 522417556774977536 (22)       0.37      0.30      0.33        63
6935543426150563840 (23)       0.24      0.27      0.25        63
7656119366529843200 (24)       0.69      0.54      0.60        65
7692148163548807168 (25)       0.12      0.21      0.15        24
7728176960567771136 (26)       0.24      0.26      0.25        54
7764205757586735104 (27)       0.48      0.47      0.47        62
9277415232383221760 (28)       0.45      0.35      0.39        74
 954763121002545152 (29)       0.22      0.24      0.23        46
9853875984686645248 (30)       0.38      0.33      0.35        67
9925933578724573184 (31)       0.20      0.27      0.23        45

               accuracy                           0.37      1675
              macro avg       0.37      0.36      0.36      1675
           weighted avg       0.40      0.37      0.38      1675
```

APPENDIX C: GEOSCOUT CUSTOM LOSS FUNCTION

Custom Loss Function that combines Haversine concepts and Cross Entropy Loss:

```
def calc_s2_distances(s2_cell_ids):
  num_classes = len(s2_cell_ids) # should be 37
  distance_matrix = np.zeros((num_classes, num_classes))
  lat_lngs = []
  for s2_cell_id in s2_cell_ids:
    cell = CellId(int(s2_cell_id)).to_lat_lng()
    lat_lngs.append(cell)
  for i in range(num_classes):
    for j in range(num_classes):
      if i != j:
        distance_matrix[i][j] = lat_lngs[i].get_distance(lat_lngs[j]).
    degrees * 111.32
  return distance_matrix

def geoscout_loss(outputs, targets, s2_distances, alpha=1, beta=0.1):
  cross_entropy = nn.CrossEntropyLoss()
  cross_entropy_loss = cross_entropy(outputs, targets)
  batch_size = outputs.size(0)
  predicted_classes = outputs.argmax(dim=1)
  distances = s2_distances[torch.arange(batch_size), predicted_classes]
  distances = (distances - distances.min()) / (distances.max() -
    distances.min() + 1e-8)
  incorrect_predictions = (predicted_classes != targets).float()
  distance_penalty = (distances * incorrect_predictions).mean()
  loss = alpha * cross_entropy_loss + beta * distance_penalty
  return loss
```

APPENDIX D: RESIDUAL NEURAL NETWORK TRAINING PROCESS

Code developed to train ResNet:

```
def setup_resnet():
  # pretrained resnet18 model from torchvision.models
  # https://pytorch.org/vision/main/models/generated/torchvision.models.resnet34.html
  # start with pretrained=True
  model = models.resnet18(pretrained=True)

  # connect resnet34 and fc layer
  # 384 cells when using s2 level=3
  model.fc = nn.Linear(model.fc.in_features, 37)

  # model.fc = nn.Sequential(
  #   nn.Linear(model.fc.in_features, 256),
  #   nn.ReLU(),
  #   nn.Dropout(0.5),
  #   nn.Linear(256, 37)
  # )

  for param in model.parameters():
    param.requires_grad = False

  layers = list(model.children())

  for layer in layers[-2:]:
    for param in layer.parameters():
      param.requires_grad = True

  for param in model.fc.parameters():
    param.requires_grad = True

  # loss func
  criterion = nn.CrossEntropyLoss()

  # Adam optimizer; may need to try different lr
  # or do we want to use SDG?????
  # check more on model.fc.parameters() vs model.parameters()
  # optimizer = optim.Adam(model.parameters(), lr=0.0001)
  # using weight decay to penalize large weights
  optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=1e-5)

  # only use a cuda device and never use a cpu device
  if torch.cuda.is_available():
    device = torch.device('cuda')
  else:
    sys.exit(1)

  torch.cuda.empty_cache()

  # move to cuda device's memory
  model.to(device)

  return model, criterion, optimizer, device

def train(epoch, model, criterion, optimizer, device, train_loader):
  total_loss    = 0
  total_size    = 0
  total_correct = 0
  total_samples = 0

  # set the model to traning mode
```

```
  model.train()

  # clear GPU cache
  torch.cuda.empty_cache()

  for batch_id, (data, gtl) in enumerate(train_loader):

    # move to cuda device's memory
    data, gtl = data.to(device), gtl.to(device)

    # clear old gradients from last iteration
    optimizer.zero_grad()

    # forward pass
    output = model(data)

    # invoke the loss func
    loss = criterion(output, gtl)

    # calculate gradients
    loss.backward()

    # update params based on computed gradients
    optimizer.step()

    # loss.item() returnsthe scalar value of the loss tensor; data.size
    (0) returns number of samples in the current batch
    # accumulate loss
    total_loss += loss.item() * data.size(0)

    # get predicted class indices
    _, predicted = torch.max(output, 1)

    # total correct prediction counter
    total_correct += (predicted == gtl).sum().item()

    # total sample counter
    total_samples += data.size(0)

    total_size += data.size(0)

  avg_loss = total_loss / total_size
  # avg_loss = evaluate(model, loader, criterion, max_distance=20000)
  accuracy = total_correct / total_samples

  # print(f"Epoch {epoch}\nLoss: {avg_loss:.4f}\nAccuracy: {accuracy:.4f
    }")
  return avg_loss, accuracy

def validate(model, criterion, device, val_loader):
  model.eval()

  total_loss    = 0
  total_correct = 0
  total_samples = 0

  pl = []
  tl = []

  for i, (data, gtl) in enumerate(val_loader):

    # disable gradient calculation
    with torch.no_grad():

      # move to cuda device's memory
```

```
      data, gtl = data.to(device), gtl.to(device)

      # forward pass
      output = model(data)

      # invoke the loss func
      loss = criterion(output, gtl)

      total_loss += loss.item() * data.size(0)

      # get predicted class indices
      _, predicted = torch.max(output, 1)

      # total correct prediction counter
      total_correct += (predicted == gtl).sum().item()

      # total sample counter
      total_samples += data.size(0)

      # # get predicted class indices
      # _, predicted = torch.max(output, 1)

      # append pl and tl to lists using cpu
      pl.extend(predicted.cpu().numpy())
      tl.extend(gtl.cpu().numpy())

  # print(classification_report(tl, pl))

  avg_loss = total_loss / total_samples
  accuracy = total_correct / total_samples

  return avg_loss, accuracy

def test(model, criterion, device, test_loader):
  model.eval()

  total_loss    = 0
  total_correct = 0
  total_samples = 0

  pl = []
  tl = []

  for i, (data, gtl) in enumerate(test_loader):

    # disable gradient calculation
    with torch.no_grad():

      # move to cuda device's memory
      data, gtl = data.to(device), gtl.to(device)

      # forward pass
      output = model(data)

      # invoke the loss func
      loss = criterion(output, gtl)

      total_loss += loss.item() * data.size(0)

      # get predicted class indices
      _, predicted = torch.max(output, 1)

      # total correct prediction counter
      total_correct += (predicted == gtl).sum().item()

```

```
        # total sample counter
        total_samples += data.size(0)

        # append pl and tl to lists using cpu
        pl.extend(predicted.cpu().numpy())
        tl.extend(gtl.cpu().numpy())

    avg_loss = total_loss / total_samples
    accuracy = total_correct / total_samples

    return avg_loss, accuracy

def main():
    num_of_epoch = 128
    batch_size = 32
    test_length = 10209
    train_loader, val_loader, test_loader = get_data_loader_2(batch_size,
      test_length)
    model, criterion, optimizer, device = setup_resnet()
    # scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=12, gamma
      =0.1) # might wanna adjust params here
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min',
       factor=0.1, patience=5)
    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []

    for epoch in range(num_of_epoch):
        train_loss, train_accuracy = train(epoch, model, criterion, optimizer
      , device, train_loader)
        val_loss, val_accuracy = validate(model, criterion, device,
      val_loader)
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)
        print(f"Epoch {epoch + 1}, Train Loss: {train_loss:.4f}, Train
      Accuracy: {train_accuracy:.4f}, Validation Loss: {val_loss:.4f},
      Validation Accuracy: {val_accuracy:.4f}")
        # scheduler.step() # dynamically adjust lr
        scheduler.step(val_loss)
    torch.save(model.state_dict(), "GeoScout.pth")
    plt.figure(figsize=(10, 5))
    plt.plot(range(1, num_of_epoch + 1), train_losses, label="Training Loss
      ")
    plt.plot(range(1, num_of_epoch + 1), val_losses, label="Validation Loss
      ")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training and Validation Loss")
    plt.legend()
    plt.show()

    plt.figure(figsize=(10, 5))
    plt.plot(range(1, num_of_epoch + 1), train_accuracies, label="Training
      Accuracy")
    plt.plot(range(1, num_of_epoch + 1), val_accuracies, label="Validation
      Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("Training and Validation Accuracy")
    plt.legend()
    plt.show()
```

APPENDIX E: COMPARISON OF OUR DIFFERENT MODELS

Graphs for Models Purged to 35 Classes Using Only CrossEntropyLoss and CrossEntropy + Haversine Loss

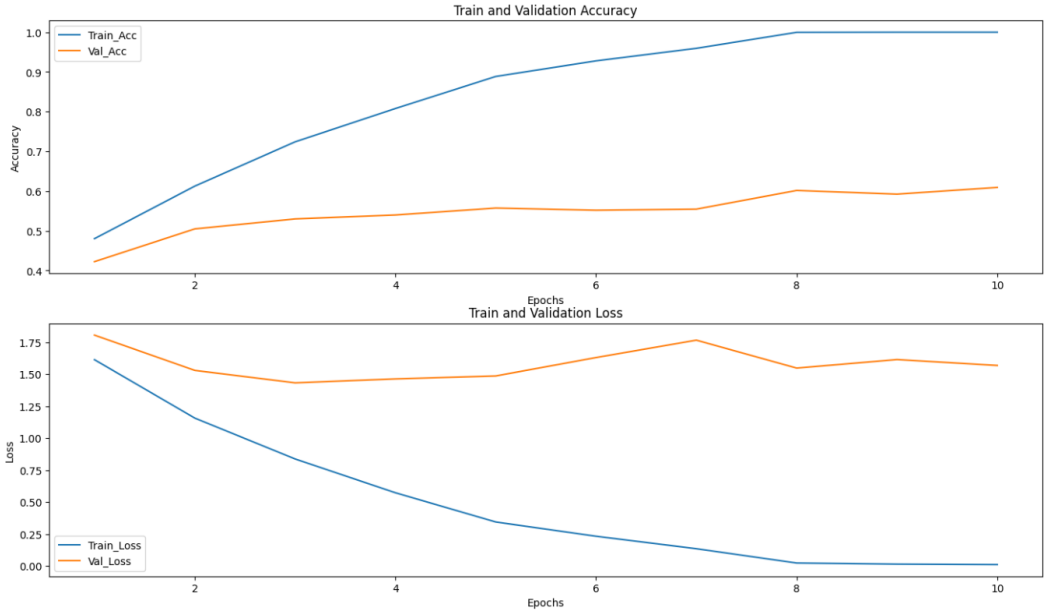

Figure 15: Purged to 35 Classes Using Cross Entropy Loss

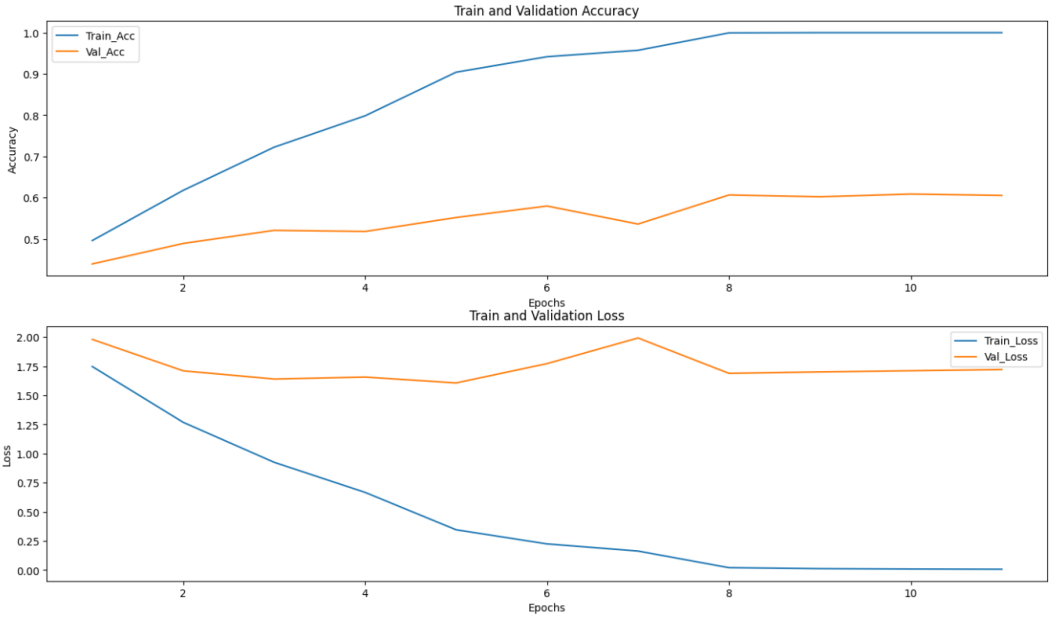

Figure 16: Purged to 35 Classes Using Cross Entropy + Haversine Loss

APPENDIX F: BEST RESNET18 ON 32 CLASSES - CONFUSION MATRIX AND CLASSIFICATION REPORT

Confusion matrix of best ResNet model:

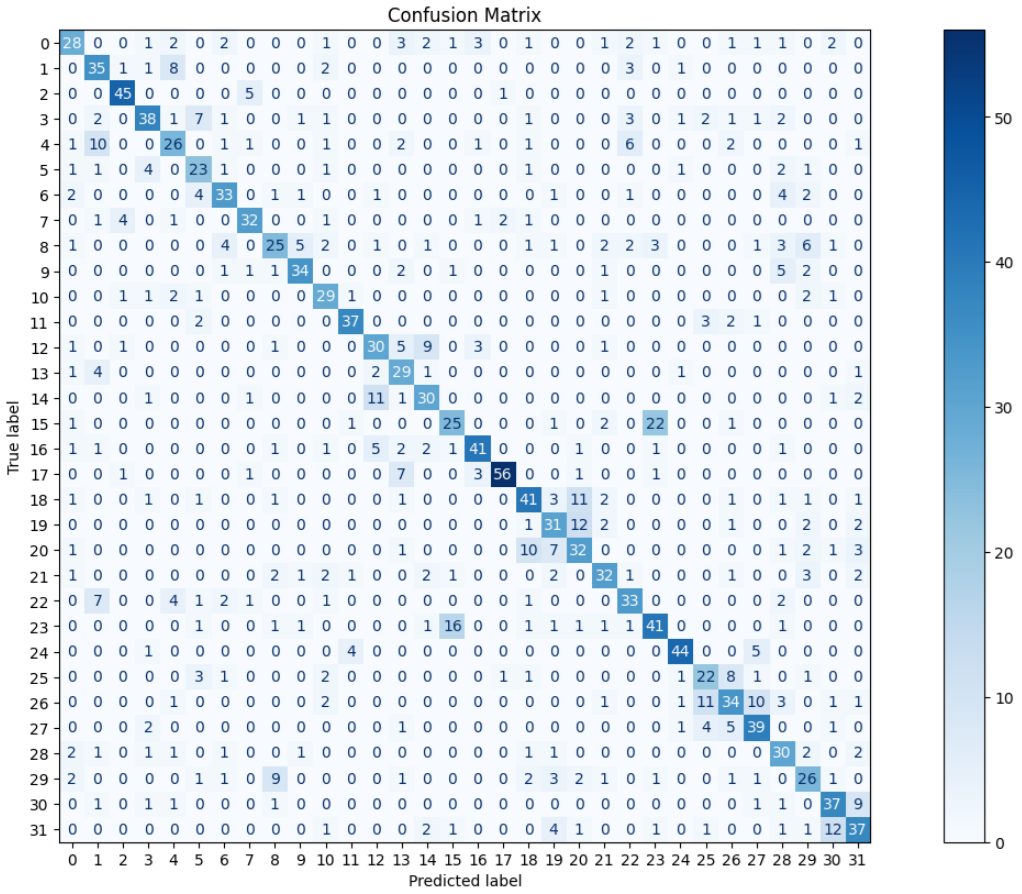

Figure 17: ResNet Confusion Matrix

Classification report of best ResNet Model

	precision	recall	f1-score	support
10250192751895248896 (0)	0.59	0.49	0.54	53
10610480722084888576 (1)	0.56	0.69	0.61	51
1062849512059437056 (2)	0.85	0.88	0.87	51
10682538316122816512 (3)	0.73	0.62	0.67	61
10718567113141780480 (4)	0.55	0.48	0.51	54
10790624707179708416 (5)	0.52	0.64	0.57	36
10826653504198672384 (6)	0.69	0.69	0.69	48
1170935903116328960 (7)	0.76	0.73	0.74	44
1495195076287004672 (8)	0.58	0.43	0.50	58
1531223873305968640 (9)	0.77	0.69	0.73	49
2215771016666284032 (10)	0.62	0.74	0.67	39
3152519739159347200 (11)	0.82	0.82	0.82	44
3332663724254167040 (12)	0.62	0.61	0.61	51
3548836506367950848 (13)	0.51	0.74	0.60	38
3584865303386914816 (14)	0.60	0.65	0.62	46

```
3837066882519662592 (15)       0.57      0.47      0.51        55
4233383649728266240 (16)       0.81      0.72      0.76        58
4305441243766194176 (17)       0.93      0.80      0.86        70
4701758010974797824 (18)       0.64      0.64      0.64        64
5062045981164437504 (19)       0.56      0.62      0.59        50
5098074778183401472 (20)       0.54      0.54      0.54        61
5206161169240293376 (21)       0.66      0.62      0.64        50
 522417556774977536 (22)       0.65      0.65      0.65        52
6935543426150563840 (23)       0.58      0.60      0.59        68
7656119366529843200 (24)       0.86      0.79      0.82        56
7692148163548807168 (25)       0.51      0.54      0.52        41
7728176960567771136 (26)       0.59      0.52      0.55        65
7764205757586735104 (27)       0.62      0.75      0.68        51
9277415232383221760 (28)       0.53      0.72      0.61        43
 954763121002545152 (29)       0.53      0.49      0.51        55
9853875984686645248 (30)       0.64      0.74      0.69        50
9925933578724573184 (31)       0.62      0.60      0.61        63

               accuracy                           0.64      1675
              macro avg       0.64      0.65      0.64      1675
           weighted avg       0.65      0.64      0.64      1675
```

APPENDIX G: INTERACTIVE DEMO TOOL

An interactive demo tool to test the model on unseen data.

```
def prep_demo():
  batch_size = 32
  test_length = 16750
  _, _, demo_loader = get_data_loader_2(batch_size, test_length)
  return demo_loader

demo_loader =prep_demo()

def plot_lat_long(ax, gtl_lat, gtl_lng, pl_lat, pl_lng):
  zoom_margin = 50  # Degrees of margin for zooming
  min_lat = min(gtl_lat, pl_lat) - zoom_margin
  max_lat = max(gtl_lat, pl_lat) + zoom_margin
  min_lng = min(gtl_lng, pl_lng) - zoom_margin
  max_lng = max(gtl_lng, pl_lng) + zoom_margin
  extent = [min_lng, max_lng, min_lat, max_lat]

  ax.stock_img()
  ax.coastlines()

  ax.set_extent(extent, crs=ccrs.PlateCarree())

  ax.plot(gtl_lng, gtl_lat, marker='o', color='green', markersize=5,
    transform=ccrs.PlateCarree(), label='Ground Truth')

  ax.plot(pl_lng, pl_lat, marker='x', color='red', markersize=5,
    transform=ccrs.PlateCarree(), label='Predicted')

  ax.legend()

def geo_scout(model, demo_loader):
  model.eval()
  random_batch = random.choice(list(demo_loader))
  inputs, gtls = random_batch
  device = torch.device('cuda' if torch.cuda.is_available() else sys.exit
    (1))
  inputs = inputs.to(device)
  gtls = gtls.to(device)

  with torch.no_grad():
    pls = model(inputs)
    pls = nn.functional.softmax(pls, dim=1)
    predicted_labels = torch.argmax(pls, dim=1)

  inputs = inputs.cpu()
  gtls = gtls.cpu()
  predicted_labels = predicted_labels.cpu()
  geolocator = Nominatim(user_agent="geo_scout_app")
  random_indices = random.sample(range(inputs.size(0)), 1)  # Process
    only one image
  fig = plt.figure(figsize=(10, 10))

  for i, idx in enumerate(random_indices):
    image = inputs[idx]
    gtl = gtls[idx].item()
    pl = predicted_labels[idx].item()

    print(f"Ground Truth Label Index: {gtl}, Predicted Label Index: {pl}"
    )
    print(f"Model output probabilities: {pls[idx]}")

    gtl_class_name = class_idx_to_name[gtl]
```

```
    pl_class_name = class_idx_to_name[pl]

    try:
      gtl_s2_id = int(gtl_class_name)
      pl_s2_id = int(pl_class_name)
    except ValueError:
      print(f"Cannot convert class names '{gtl_class_name}' or '{pl_class_name}' to integers.")
      continue

    try:
      gtl_cell_id = CellId(gtl_s2_id)
      gtl_latlng = gtl_cell_id.to_lat_lng()
      gtl_lat = gtl_latlng.lat().degrees
      gtl_lng = gtl_latlng.lng().degrees

      pl_cell_id = CellId(pl_s2_id)
      pl_latlng = pl_cell_id.to_lat_lng()
      pl_lat = pl_latlng.lat().degrees
      pl_lng = pl_latlng.lng().degrees
    except Exception as e:
      print(f"Error in converting S2 Cell ID to lat/lng: {e}")
      continue

    gtl_lat_lon_str = f"{gtl_lat}, {gtl_lng}"
    pl_lat_lon_str = f"{pl_lat}, {pl_lng}"

    time.sleep(1.5)
    try:
      gtl_location = geolocator.reverse(gtl_lat_lon_str, exactly_one=True, language='en-US', timeout=100)
    except Exception as e:
      print(f"Error during reverse geocoding for ground truth location: {e}")
      gtl_location = None

    time.sleep(1.5)
    try:
      pl_location = geolocator.reverse(pl_lat_lon_str, exactly_one=True, language='en-US', timeout=100)
    except Exception as e:
      print(f"Error during reverse geocoding for predicted location: {e}")
      pl_location = None

    if gtl_location:
      gtl_location = gtl_location.address
    else:
      gtl_location = f"S2 Cell ID {gtl_s2_id}"

    if pl_location:
      pl_location = pl_location.address
    else:
      pl_location = f"S2 Cell ID {pl_s2_id}"

    ax_img = fig.add_subplot(1, 2, 1)
    ax_img.imshow(image.permute(1, 2, 0))
    ax_img.set_title(f"Ground Truth Location: {gtl_location}\nPredicted Location: {pl_location}", fontsize=12, wrap=True)
    ax_img.axis('off')

    ax_map = fig.add_subplot(1, 2, 2, projection=ccrs.PlateCarree())
    plot_lat_long(ax_map, gtl_lat, gtl_lng, pl_lat, pl_lng)
    ax_map.set_title('Location Map')

```

```
  plt.tight_layout()
  plt.show()


def demo():
  model = models.resnet18()
  model.fc = nn.Linear(model.fc.in_features, 32)
  model.load_state_dict(torch.load("/content/drive/MyDrive/APS360/
    GeoScout.pth", weights_only=True))
  model.eval()
  if torch.cuda.is_available():
    model = model.cuda()
  else:
    sys.exit(1)
  geo_scout(model, demo_loader)
```

APPENDIX H: GEOGUESSR WORLD CUP

The following image shows a no moving, panning, or zooming (NMPZ) game of MK against Kratsoo in the 2024 GeoGuessr World Cup. The players are shown a still image and are tasked to guess its location on the map. In this particular round, both players make guesses far from the target location and receive 0 points. MK eventually wins the game and will go on to lose to Blinky in the finals. This anecdote is evidence of the difficulty of geolocating an image, as even the best players who have dedicated years to the game can make blunders.

Figure 18: 2024 World GeoGuessr World Cup Recap (GeoGuessr, 2024)