Geoguessr AI

Ashton Thomas*
University of Michigan
aethom@umich.edu

Claire O'Neill*
University of Michigan
cconeill@umich.edu

Emre Hayir*
University of Michigan
ebhayir@umich.edu

Sushrita Rakshit*
University of Michigan
sushrita@umich.edu

Abstract

This study explores the intersection of navigation, object detection, and geographic recognition in the realm of Computer Vision. Current Computer Vision has a large focus on refining techniques in recognizing both everyday objects and geographical landmarks. This research paper introduces Geoguessr AI, a model designed to identify locations within the United States from images, irrespective of lighting or seasonal variations. By leveraging a large dataset of labeled locations with many time and seasonal variations, the project explores the efficacy of extending pre-trained models, such as ResNet-50, through layer augmentation and activation functions.

1. Introduction

Navigation and object detection are prominent issues within the Computer Vision field that have recently emerged and are continually being refined. This includes arbitrary day-to-day objects and also landmarks and geographical locations. However, considering the issues associated with object recognition, we wanted to extend our understanding of geographic recognition. Particularly, if our model were to be fed in a large dataset of labeled locations, would it be able to recognize where in the United States it was located, regardless of seasonal and lighting presence within the images? This introduces our model Geoguessr AI, which stems from our group's love for looking at geographical architecture images and guessing where they came from.

For the sake of consistency and better predictions, we wanted to narrow the scope of our project to the United States, which would make data aggregation easier since the United States has a wide and diverse landscape. This project is important as it helps us understand how pre-trained models such as ResNet-50 function and how we can expand upon them by adding layers and activation functions on top of frozen and unfrozen layers to produce correct responses.

2. Background

2.1. Model Considerations

Training a model to produce approximate coordinates given a street-view image is a challenging machine-learning task. Previous approaches to this problem focused on identifying locations based on landmarks in images [5]. A promising model from 2016 called PlaNet tackled this problem by subdividing the surface of the earth into geographical cells [5]. This allowed researchers to treat the the geolocation task as a classification task. The resulting model out performed similar models such as IM2GPS and Skyline2GPS on geolocation tasks [6, 7]. From this research, we decided to begin to approach this problem as a classification task, where the US would be split into 150 boxes, specifically 15x10 evenly spaced grids. Our model would have to predict which box the image originated from.

It is important to recognize that a model's capabilities are limited to the data it is trained on. The PlaNet model pulled over 150 million images from the internet with minimal filtering. This led to improved performance on the geolocation task, however, we knew that we did not have the resources to pull millions of images [5]. For this reason, we decided to use a pre-trained model instead of training a model from scratch. After recent successes in transfer learning on a variety of tasks, we decided this approach would allow us to train a model with limited data. We were able to collect 61,000 images from Mapillary API that we could use to fine-tune the pre-trained model.

To tackle this problem, we had to choose a model we could fine-tune that has previous knowledge and large amounts of data we could extrapolate. When considering

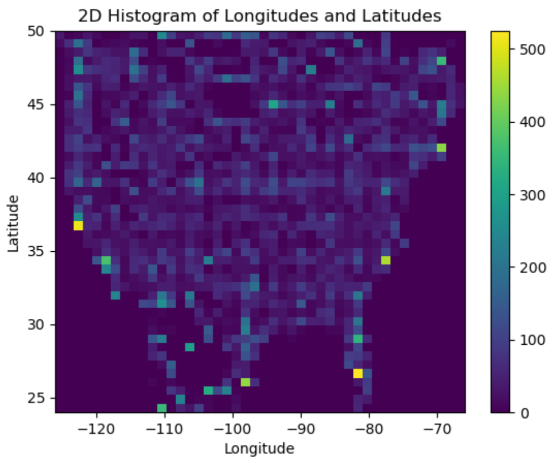

Figure 1. A graph displaying the dispersion of our data fetched from the Mapillary API.

models, we wanted to prioritize models that performed well on visual recognition tasks, this led us to VGGNet and ResNet. Initially, we considered using a VGGNet model because of its capabilities fitting complex data due to the model's depth [3]. However, we found the model to be problematic when attempting to work with its weights, as a result, we decided to try ResNet-50, a less complex model [4]. We found this model much easier to work with, while still performing well on its benchmark tasks [4]. These are all important background factors that were considered before refining our experiments and project methodology.

2.2. Final Model Selection - Pretrain Task

ResNet-50 worked amazingly since it can perform well with few layers. ResNet-50 is trained with trillions of data points and hundreds to thousands of hours of computing power, which would help since we lacked resources. We did additional research on transfer learning and found that if we freeze layers within ResNet-50, we could keep its robust recognition skills while adding our data to the model. We also explored ResNet-50's complete architecture and found code demonstrating how to freeze and unfreeze different layers within the pre-trained model. We employed all this information in our methodology and experimentation sections. Full visual information can be seen within Figure 2.

Each layer within ResNet-50 was responsible for different specialties within the input images. For example, the intermediary layers have 3x3 convolutional layers and the final output layer is a 1,000 way fully-connected layer utilizing the SoftMax activation function [2]. By reading through in-depth about the ResNet-50 architecture, we found that the final few layers were key in the model's objective, which was object recognition [2]. Going forth in the methodology

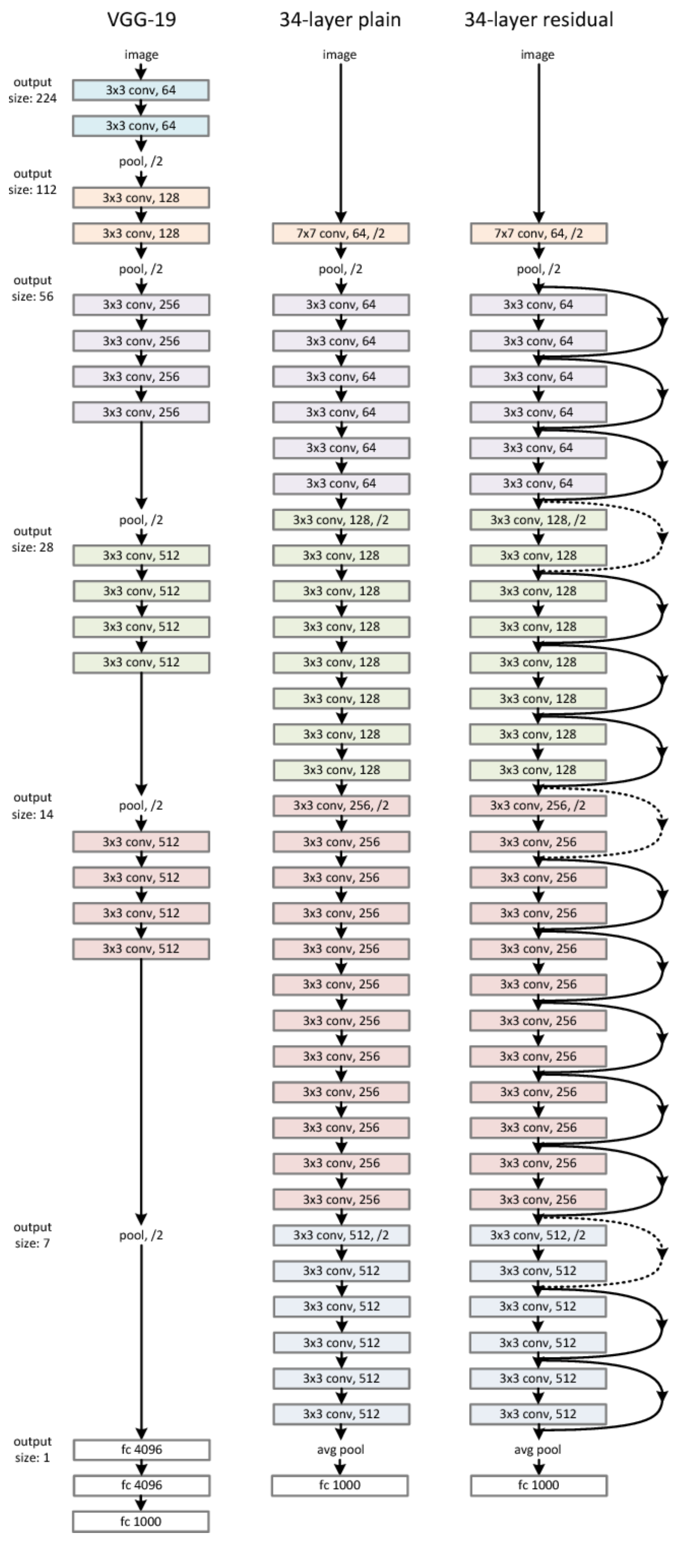

Figure 2. Detailed Architecture for ResNet-50's convolution layers, fully-connected layers, and activation functions. [1]

section, we would explore the differences in freezing all the layers and unfreezing the final few layers in order to make it geographically compatible instead of just for object detection.

3. Methodology

3.1. Experiment 1

We decided to use a pre-trained model and perform transfer learning and fine-tuning. The first portion of this process was to find adequate data for our learning objective. We scraped a free map-based API called Mapillary. We had set the data scraper to grab images evenly distributed from each of 150 different regions within the United States, which we defined by going from -66 to -126 degrees longitude and 24 to 50 degrees latitude. Mapillary would attempt to grab these images evenly and return an image, coordinate pair. We aggregated 61,538 images with coordinates, which we used to create training and evaluation sets. We used 80% of this data for training and 20% for evaluating our model.

For this experimental stage, we chose a learning rate of 10e-5 with a batch size of 1 and a regularizer of 10e-3. The loss function was our custom Haversine distance loss and we set the number of epochs to 3.

Initially, we intended to train our model by creating "boxes" for each coordinate to fall into. The output from the model would be a vector sized 150x1 where each box *n* represents the normalized probability of the image belonging to such box n. To determine the final point our model would output we'd multiply the probability of each box by the coordinates of the center of the box, this output would be latitude/longitude of the weighted average of the boxes. To measure our accuracy we created a custom loss function that would penalize the model based on the Haversine distance between our predicted point and the correct point [8]. Particularly, if the model predicted boxes far from the ground truth point, the penalty would be applied as the distance between the weighted center of the boxes (the prediction) and the actual location. For reference, the Haversine distance between two points (ϕ_1, λ_1) and (ϕ_2, λ_2) on the Earth's surface, where ϕ represents latitude and λ represents longitude, is given by:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

where:

$$\begin{aligned} \Delta\phi &= \phi_2 - \phi_1 \\ \Delta\lambda &= \lambda_2 - \lambda_1 \\ r &= \text{radius of the Earth} \end{aligned}$$

This approach seemed promising, however, it led to many challenges. We were troubled with how we might perform batching. We could only enable our model to train with a batch size of 1. This was inconvenient as we wouldn't be able to generalize properly. When we plotted the loss for our training with batch size 1, we saw a drastically oscillating loss graph with no consistent trend, of which was also not monotonically decreasing, indicating that convergence criteria was not being met and the model was not learning adequately. This was the first method we attempted, which failed. To rectify this, we decided to alter our approach.

3.2. Experiment 2

After the above experiment, we decided to make a model that would only give two outputs: the latitude and the longitude. We would also use the loss function Mean Squared Error to allow for generalization in batching instead of complicating our Haversine distance loss. The inputs for training were the latitude and longitude pairs from the images and the output was the model's prediction of the image (latitude and longitude values). For this experimental stage, we chose a learning rate of 10e-5 with a batch size of 50 and a regularizer of 10e-3. We decided to keep ResNet's base knowledge by freezing all the layers of the pre-trained model and conducting a one-shot performance. However, despite training with our data, we saw a perfectly linear representation of the model, going from Seattle in the northwest to Florida in the Southeast. For reference, this result can be seen in Figure 3.

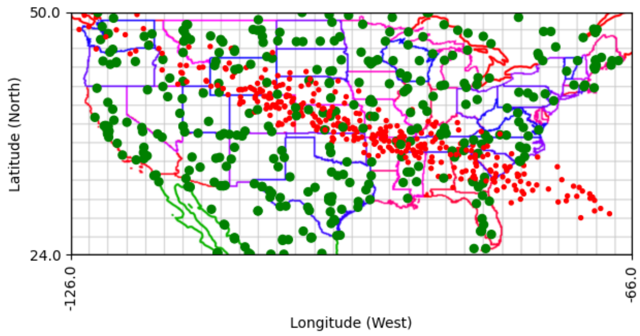

Figure 3. The output of the experiment and methodology in 3.2. As can be seen, there is a very linear, somewhat noisy representation of predictions that are distributed along the Northwest to Southeast diagonal of the United States.

We figured this would be due to the linear predictions from the fully connected layer's output. As a result, we decided to add new activation functions and fully connected layers outside of ResNet's final fully connected layers. We added 3 new fully connected layers with intermediate ReLU activations, including after the final fully-conected layer. From this, we saw a significant boost in performance. We saw drastic variability in predictions afterward and there was still slight linearity. This makes sense as the model picked up on the United States shape and saw that there was more dense data collection towards corners of the country. This improvement can be seen in Figure 4. There was a concerning issue, however, where the model was unlikely to predict the Northeast United States and also consistently predicted the Atlantic Ocean when there were no training points located in the region. Our next step in methodology was to speculate why this was the case and to fix this discrepancy.

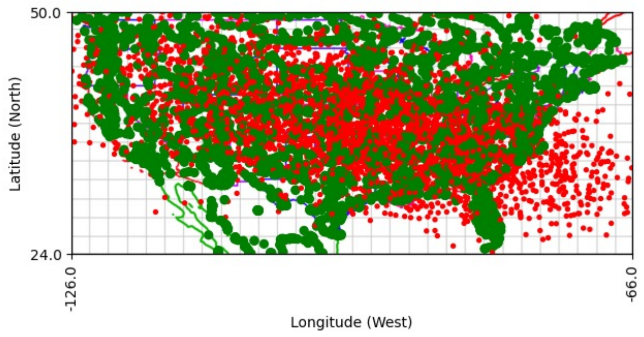

Figure 4. The output of experiment and methodology in 3.2. As can be seen, the model is still somewhat linear but definitely an improvement compared to Figure 3.

3.3. Experiment 3

After performing the above experiments and noticing that there were significant predictions within the Atlantic Ocean, we decided to modify the pre-trained model itself. The last few convolutional layers within ResNet-50 were what we decided to target, as these layers were responsible for the object detection. We deduced that since ResNet-50 was designed for object detection, it was likely picking up on cars and buildings within the data, not the geographic location itself. As a result, we decided to stray away from freezing all layers of ResNet-50 and instead unfreeze the last convolutional layer.

In addition to unfreezing the last convolutional layer, we decided it would be best to tune additional hyperparameters. For learning rate, we chose 10e-5 with a batch size of 50 and a regulizer/weight decay of 10e-5. We decided to keep the loss function as Mean Squared Error loss and increased the epoch size from 3 to 7. After making these changes we saw significant improvements with a smoother loss and less linear predictions trailing into the Atlantic Ocean. We were still facing issues where we would never predict Florida, however, this was a large step into meaningful predictions for Geoguessr AI.

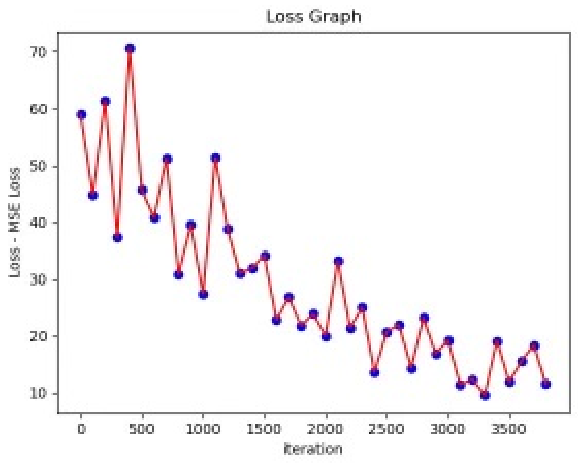

Figure 5. The training loss associated with Experiment 3.3. We obtained a reliable, monotonically decreasing loss curve, indicating a more stable learning process unlike in experiment 3.2.

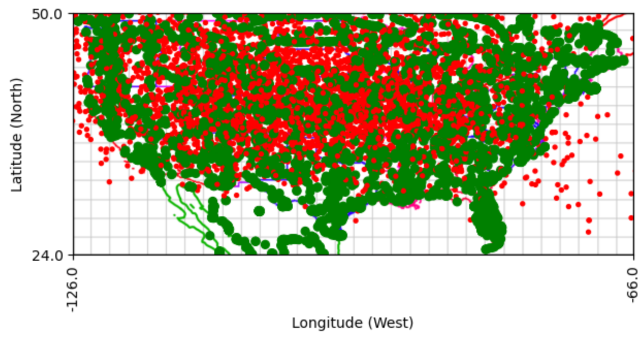

Figure 6. Predictions associated with experiment 3.3. There is less predictions spontaneously scattered within the Atlantic Ocean. However, we see the model skips predictions in Florida and also overpredicts in Northern South Dakota (where we are missing data).

4. Results

4.1. Final Model Section and Architecture

Our final model was the model from the section titled Experiment 3 (section 3.3). It consisted of transfer learning from ResNet-50, unfreezing the fifth and final convolutional layer prior to the output, and adding a fully connected layer. This fully connected layer had 1028 nodes as input from ResNet-50 and 2 nodes as output, which were our latitude and longitude predictions.

4.2. Training Statistics

The best Geoguessr AI model from experiment 3 had consistent loss plots that monotonically decreased and converged through Mean Squared Loss. In addition, we plotted predictions through the entire final training process to see examples of our performance you can see this in Figures 7 and 8.

4.3. Evaluation

We evaluated our final Geoguessr AI model and it performed rather well on the evaluation set, which was the remaining untouched 20% of our set. We saw an even scatter between our ground truth data and the model's prediction.

In addition, our second evaluation tactic consisted of shuffling our evaluation set and grabbing the first image and label at the top of our dataset. We would feed the model the input, request a prediction, and plot the ground truth and prediction on the same map. We saw some astoundingly close-to-accurate results! These are shown and defined in detail within Figures 7 and 8.

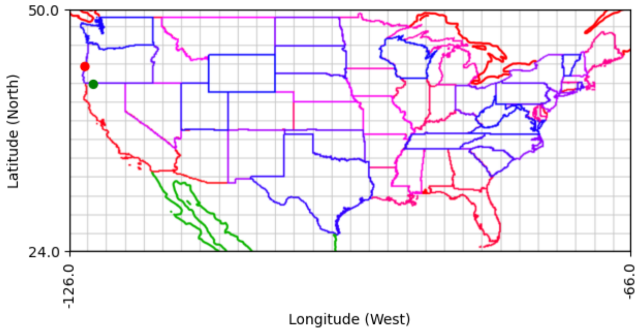

Figure 7. Singular prediction associated with singular ground truth in our evaluation set. As can be seen, model is very close at guessing! For clarity, the green dot represents the ground truth while the red dot represents model prediction.

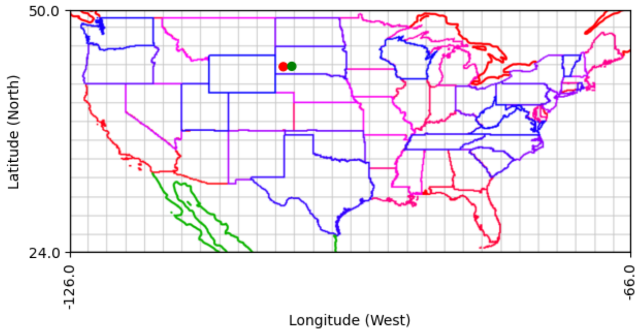

Figure 8. Singular prediction associated with singular ground truth in our evaluation set. As can be seen, model is very close at guessing! The green dot represents the ground truth while the red dot represents model prediction.

As we ran the single plotting across different shuffles of our evaluation set, we saw that the model was extraordinarily good at predicting latitude, with longitude giving some variance. We suspect this is the case because latitude is further associated with foliage and desert/seasonal variations than is longitude. The model likely picks up on these regional features and is able to accurately grasp proper coordinates. In addition, careful shuffling of our evaluation and intentional separation between the training and evaluation sets guaranteed that our model was not memorizing noise associated with the images but was rather learning proper image representations.

5. Conclusion

In short, we spent many weeks with different methods creating an effective model via supervised learning that would be able to guess where an image came from when given an image as input. While we saw significant improvements in performance, we would have liked further time to test new ideas. Some of these ideas include trying different base models as a pre-trained base. We originally picked ResNet-50 due to its strong performance on image classification tasks and its relatively manageable computational demands. However, we noticed many data points were taken during different times of the day and seasons, leading our model to pick up on causal differences. As a result, it would have been beneficial to apply grayscale and other augmentations to our image and feed that input to the pre-trained model. This was not possible with ResNet-50 since the model was pre-trained for the classification of color images.

The conclusion we drew from this project is that guessing geographic locations can be a difficult task, especially since most of our data comes from seasonally varying sources and also since it comes from different times of day. We want a rigorous model that can properly handle this but since geographic locations can vary heavily, we would need millions of more data points to do so. We have realized how integral data is and the difficulties in creating custom losses that could accurately monitor the performance of our model. We hope to forward this knowledge in future machine-learning research projects, industrial research, and capstone courses.

References

[1] GeeksforGeeks. (n.d.). Residual Networks (ResNet) in Deep Learning. Retrieved from `https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/`

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. (2015). Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*. Retrieved from `https://arxiv.org/pdf/1512.03385`

[3] T.T. Lemani and T.L. van Zyl. (2023). Comparing Male Nyala and Male Kudu Classification using Transfer Learning with ResNet-50 and VGG-16. *arXiv preprint arXiv:2311.05981*. Retrieved from `https://arxiv.org/pdf/2311.05981`

[4] VISO AI. (n.d.). VGG – Very Deep Convolutional Networks. Retrieved from `https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/`

[5] Tobias Weyand, Ilya Kostrikov, and James Philbin. (2016). PlaNet - Photo Geolocation with Convolutional Neural Networks. Retrieved from `https://arxiv.org/pdf/1602.05314/`

[6] James Hays and Alexei Efros. (n.d.). IM2GPS: estimating geographic information from a single image. Retrieved from `http://graphics.cs.cmu.edu/projects/im2gps/`

[7] Srikumar Ramalingam, Sofien Bouaziz, Peter Sturm, Matthew Brand, and others. (n.d.). SKYLINE2GPS: Localization in Urban Canyons using Omni-Skylines. Retrieved from `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5649105`

[8] scikit-learn. (n.d.). *haversine_distances*. Retrieved from `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.haversine_distances.html`

6. Appendix

6.1. Setup

1. Download the Repo via the Github link by using the git clone method: Github

   https://github.com/aethom00/Geoguessr442

2. Download the Checkpoints and the Image Data via the download link and create a folder within the project directory named Predictions_new: Data

   https://drive.google.com/drive/folders/1POZeCffw Gw3xwal5pob0dSFtzUHK9-Gc?usp=sharing

3. **Disclaimer***: You may not be able to run this code unless you have access to a CUDA GPU

6.2. Overview of Options

First you will see a driver file named driver.py. This is how you will operate our model. The driver features two commands:

- - - train: Will allow you to train the model with a necessary parameter (integer) which represents the number of epochs you would like to train the model with.
- - - evaluate: Will allow you to evaluate the model with the options:
  - The number of images you'd like to evaluate (integer)
  - Whether you would like all the points to display on different graphs/maps (boolean, True or False inputs)

6.3. Executing the Code

Running any code below should output each estimation's Haversine distance (distance accounting for Earth's curvature) between the estimated point and the correct point in the form...

```
Haversine Distance of iteration i: n km
```

where i is the ith iteration and where n is the number of kilometers between the estimated point and the correct point cut off at the hundredths place (e.g. 10.96 km).

1. You may run a line such as this to train your model with 7 epochs:

   ```
   python3 driver.py --train 7
   ```

   Your result should be something like this which displays the green points (actual locations) vs the red points (the model's predictions).

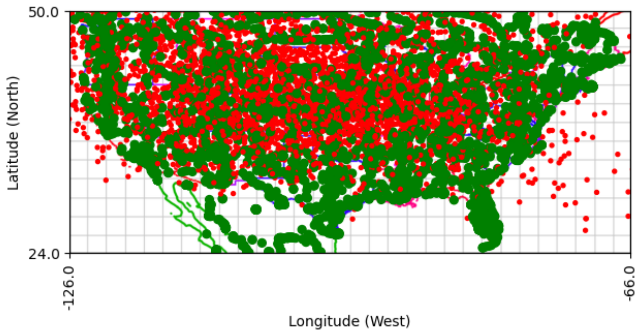

Figure 9. An example of the train command line option

2. You may run a line such as this to generate a single point estimation:

```
python3 driver.py --evaluate 1 False
```

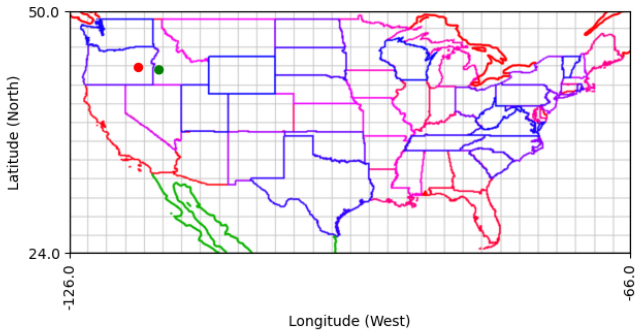

Figure 10. An example of the evaluate command line option