GeoGuessr Final Report Document

Evan Glas, William Luqiu, Abenezer Addisu Gelaw, Eric Qi

Problem Statement

Project Background

The goal of our project is to develop a machine learning model that can accurately predict the coordinates and country of a given set of Google Street View images and to integrate this model into a playable GeoGuessr-like web app. Our project, when completed, will ideally offer an enjoyable way for users to both expand their geography knowledge and practice GeoGuessr against an AI opponent.

Geoguessr is a web-based game where players compete by guessing the locations of randomly chosen Google Maps Street View Panoramas. It was launched in 2013 by Anton Wallén and has recently gained popularity. Players receive scores based on how close their guesses are to the actual locations. We are aiming to use a computer AI to generate guesses for the game. We think this will be successful as there are distinctive features in the image relating to the general location such as road signs, geography, and languages used in storefronts.

Previous research into this area includes a CNN/LSTM based neural network [1] and a PyTorch-based CNN/RNN [2]. While these models are not directly comparable to our planned model (these models have different problem definitions), we are aiming to improve the accuracy of these previous research forays.

We decided to do this project as we are all avid Geoguessr players and would like to learn more about techniques to classify and segment images to detect features that would be relevant to spatial location.

Stakeholders

Henry Pfister (Professor)

Prof. Pfister will advise us and give general guidance on the direction of the project, as well as technical advice on how to implement and improve AI/ML aspects of the project. For example, Dr. Pfister's advice on using patches from the images was extremely helpful for our model development.

GeoGuessr Team: Billy, Eric, Abenezer, Evan

We anticipate that we will be the primary stakeholders of this project. As we will be graded on our work and we are likely to be the heaviest users of our completed game, we likely have the greatest stake in the project outcome. As we all enjoy playing GeoGuessr, our game may offer an alternative for us to the classic GeoGuessr game in which we are virtually not limited by playing time and can play at our own pace.

Division of Responsibilities

Billy and **Evan** will primarily work on the front end. Billy worked on the front-end/back-end integration with an inference API, and did initial work on the Google Maps integration with React. He set up the AWS infrastructure for the inference API. Evan took over all other aspects of the front-end development, setting up the initial React App, styling it, writing game logic, and making a usable front-end interface.

Aben was tasked with model development and deployment. Aben was researching ONNX and ways to make the model much more robust and accurate with pictures coming from top European cities; unfortunately this has been met with little success. Right now Aben is playing around with image scraping Bing to get a bit more data for the cities we are considering. Aben also experimented with ConvNeXT to see if the model would be able to become more accurate (it did not).

Eric was in charge of model development and data collection. He has scraped and recorded all the Google Street View images (10k+) that are to be used for training and testing. Using these images, He continually trained and tested a variety of models and approaches to determine which method was the most feasible for playing Geoguessr. The final model we settled on was a custom architecture utilizing a ResNet-18 model in conjunction with a feedforward “reliability network”, trained in a very specific manner.

Users

Geoguessr Players

Geoguessr players will be able to use a bot to help them improve their geography skills (e.g. maybe the bot could provide a location that is “most likely” given the input image, and the user could then use this suggestion to inform their own guess). Furthermore, the players can also practice against the bot as a mock opponent, giving them a sense of what the competition will be like. Beginning users are also expected to benefit from this as it will provide them with a frame of reference to situate their guesses. Advanced users may use the AI to spot patterns in images to better inform their guesses, especially if the AI was an interpretable one. Even without that, the AI may give confidence scores that will help Geoguessr players learn features that inform the decisions.

Geography Learners

Those interested in gaining knowledge of geography may use our game to gain exposure to the physical geography and feel of different locations across the world. As the game provides the actual location of the StreetView at the end of each round, the user may choose to further explore the area (such as through a Google search) after the round ends.

Risks

Google Street Maps API costs money and we have limited funds. We need to gather enough Street View images to produce a viable training and testing set, which may prove to be difficult. However, we used the generous free credits provided by Google's Street Maps to mitigate this issue. The transfer of such large amounts of image data, without hosting it on a server, may also be an issue. Thankfully, the Google Cloud Platform (GCP) allows you to use persistent disks to store your data, and we all have ample enough storage space in Google Drive for Google CoLab. Further, playing the game incurs a cost per API request. Each round likely generates at least three calls to the Google Maps API (for the StreetView, acquiring guess distances, and potentially manipulating the map). Our game may generate additional calls to the API when the user interacts with the StreetView, although we are not certain this is the case. We do not anticipate the cost from playing the game ourselves to pose a substantial risk (as our usage is likely far below our current Google Cloud Credit allocation), however, if we choose to scale the game, this would potentially become a risk.

Training the classifier may also pose a significant risk. As training will probably take a very long time, and we don't know how well the classifier will perform. If the classifier performs poorly, it may take significant time and effort trying to create a different classifier that will perform better. As it is currently, with our training and evaluation dataset, it takes around 6 hours to finetune the model, with significant computational resources required (either GCP or Google CoLab is used for training). Lackluster results require us both to rethink our approach and redo the training.

Another potential risk is poor time management, as this can easily sidetrack the project. Early in the semester, we established a tentative project plan which we could use to gauge our progress and timeliness throughout the semester. Further, open communication between the team (in the form of a group chat) should help to mitigate this risk by enabling us to hold each other accountable and ensuring we are up-to-date with each other's progress.

Finally, individuals may leave the project and withdraw from the class. In our case, Kevin withdrew from the project, but in accordance with this risk, the team reduced the scope of the project.

Assumptions

We are assuming that people will not be using the bot during tournaments or other competitive events. As this would be unethical and any players caught cheating would be immediately banned. We are also assuming that we are able to create a geoguessr knockoff so we don't have to interact directly with the game on the webpage. We don't have the source code for the website, so making a robot that can interact with the website is considerably more challenging and would require more engineering bandwidth.

Further, we are assuming that we are able to obtain enough data from every country in Europe so that the model we train is able to accurately predict the spatial location of an image. As we have learned, this is impossible for certain countries due to certain political and social restrictions. As such, we have adapted our solution to exclude these areas (for example, Bosnia and Herzegovina is a country that does not have any street view images available).

Finally, we are assuming the boundaries for what is considered Europe to be 36 to 70 degrees latitude and -10 to 50 degrees longitude. This way, our app would ideally concentrate on an area with a high degree of geographic diversity (Europe), while also avoiding distant portions of Russia farther east.

Modern Design Considerations

Societal Impact

Public Health, Safety, and Welfare

We hope that our project will offer users a fun game to play as well as an opportunity to extend their geographic knowledge. Our project offers a free means to play a GeoGuessr-like game at the user's own pace and without play-time limitations. In this sense, we hope our project would generate a positive impact on the welfare of all users. We do not anticipate our project posing significant risks to public health or safety, especially given the safety of preexisting GeoGuessr-like games.

Global Impact

At its current scale (a locally-hosted web application), we do not envision our project generating significant global impact.

Cultural Impact

There may be moral concerns among the Geoguessr community about using AI to cheat. There may be traditionalists and conservatives who believe that computer aids should not be used even for training, and thus all the information should be gleaned from the players themselves. Nevertheless, we do not anticipate our application presenting a significant cultural disruption given the AI's current sub-human performance.

Social Impact

We do not anticipate our project generating substantial social impact.

Environmental Impact

Our project consists of a web app and a framework to run the AI model. At its current scale, our system likely requires minimal energy input, and is likely otherwise environmentally neutral as all software is likely deployed on preexisting machines. In this sense, we do not feel like our project presents a significant environmental impact.

Economic Impact

Our project currently offers a free way to experience a GeoGuessr-like game, which can save our users money compared to other platforms which may charge based on usage/features. However, if we were to scale our game significantly, we would potentially incur costs from the Google Maps API which we would need to address in some way.

Ethical Considerations

Global

The ramifications of an AI model that is able to accurately predict geospatial location solely based on street view images, which look very similar to typical photos taken from a phone, could be very serious for political reasons. If images containing sensitive, even confidential information are leaked, they could be used for very dangerous purposes, especially if the model is able to give an accurate latitude and longitude. For example, take the Russo-Ukrainian war: if classified military images are leaked (maybe pictures of supplies and troops), and the opposing side has an extremely accurate AI model that has been trained to perform well even under adverse conditions (i.e. blurry images), they could use these leaked images to gain important intel on their enemies location. Of course, this assumes that such a perfect model exists (which none do), but the potential ramifications of having something even remotely close to that can be devastating, and as such, need to be treated with caution.

Economic

Building an AI to play Geoguesser could potentially disrupt the professional Geoguessr community. It may not be fun to watch humans play Geoguessr if AI can simply perform much better. However, this seems unlikely due to the advent of AI in games such as chess, and those are still watched worldwide. Our current implementation of the AI has also not yet approached human-level performance.

If we were to scale our Geoguessr-like game to support many players at the same time, in other words, producing a Geoguessr-like game to directly compete with the actual Geoguessr, then this would indeed be an ethical concern. Since we used Geoguessr as a source of inspiration, it would be an ethical problem if we were to monetize our platform – and would likely result in a lawsuit filed by the original Geoguessr company. Of course, we don't plan on actually scaling up our project to be used by hundreds, if not thousands of people, so this will not be a concern.

Environmental

We do not anticipate our project raising lasting ethical concerns about the environment.

Societal

If our model had better performance, it would likely raise privacy concerns from a large sector of the population. If such a model fell into the wrong hands, it could be used for all sorts of privacy violating things such as surveillance, stalking, etc. Privacy issues such as these are already at the forefront of people's minds thanks to the public outcry over how big tech companies such as Meta deal with private information. Thankfully, our project does not achieve such a high level of accuracy, and so it would not raise much ethical concerns about society's future (the opposite of ChatGPT).

Problem Solution

Description of Problem Solution

Our problem solution will be in the form of an interactive webpage that allows users to play Geoguessr with or against an AI model. The user will be presented with a Google Street View panorama and will be asked to identify the location of the panorama on a map. After selecting a location and submitting it as a guess, the user will be able to see how far off they were from the correct location and compare it to the prediction made by the AI model. You will receive points depending on how close your guess was to the actual location, and so will the bot. Each game will be composed of 5 rounds of guessing, and at the end, the user's total score will be tallied and compared to the model to determine who won the game. In this way, the AI model can serve as a helpful tool or as a competitive bot for the user. Hopefully, by the final report and demo, the AI model will be accurate enough to provide meaningful competition for the user.

List of Features

Frontend: Implemented

- Google Street View Explorer enables users to look around a given location
- Google Maps enables users to submit a guess
- Button to see AI guess before submitting user guess, ability to query the model for the guesses in a timely manner before user requests AI guess
- Ability to display user guess, AI guess, and actual location after a given round
- Ability to compute distances between each AI/actual and user/actual guesses
- Output the winner of each round
- Ability to display the country that corresponds to both AI/User guesses
- After a round is over, provide a button that enables the user to learn more about the actual location of the round, i.e. a google search with that city as well as a button that links to our project GitLab.
- A custom scoring system which considers both absolute guess distance and country correctness
- **Scoring system**: Instead of just using the guess distances as scoring metrics, we can bring in additional information (such as the country/country subdivision) which could assign varying

amounts of points to the guesses. This could likely make the game more interesting as the quality of the guesses are not solely determined by their distance from the true location (e.g., guesses in Russia may be spaced extremely far apart, but guessing within Russia when the true location is in Russia should still likely incur additional points).

Frontend: Potential Future Work

- **Custom Game Length**: Enable the user to choose the number of rounds they would like to play before the game ends rather than endless free play.
- **Better gaming experience**: In addition to the basic functionality currently implemented, it could make sense to enable power-ups, speed bonuses, or obstacles to make gameplay more dynamic and continually exciting. This feature would be especially important for the "GeoGuessr Player" category of users as they likely expect a refined gameplay experience.
- **Multiplayer gaming**: allow users to play against each other and/or the AI in a live setting.

Model features

- The model will be able to predict a country given the Google Street View Static image. We specifically feed it 4 images taken from 4 different headings, and average the resulting output to yield a country.
- The model API is able to take in a lat/lng of actual coordinates and return a lat/lng of predicted coordinates.
    - It does this by querying Google Street View for the static images, running those through the model, and outputting a resulting lat/lng.

Removed Originally Considered Features

- **Python application**: in our original vision document, we expressed a plan to build a Python application. We ultimately decided to build a JavaScript-based web application instead of a Python-based application given the ready accessibility of the Google Maps API in JavaScript and our past experience building web apps with React JS.
- **Static images**: we decided to present the user with a fully immersive Google Street View object rather than static images. At first, we had thought that it would make more sense to give the user static images so they see the same data as the AI. However, we chose to instead present the user with a full Street View in order to provide a more immersive UX. The AI also now receives a set of static Street View images collected around the actual location rather than the single static image we had planned to present to the user.
- **Random Data Collection:** After scraping an initial dataset of street view images, we quickly realized that randomly acquiring data this way was not going to be feasible. This is because the vast majority of images that were scraped were pictures in the countryside, and of trees, shrubs, grass, etc. These images hold almost no salient information that the model can learn from, and if trained on, will yield subpar predictions. As such, we instead pivoted to collecting images taken

from the vicinity of major cities, where there is a lot more information that the model can learn from.

- **Regression only output.** We initially had our model trained to predict a latitude and a longitude (so a regression output), but we quickly learned that the model had learned to only predict the mean location of all the images, instead of being able to discern each location independently. We now have the problem defined as both a classification problem (predict the country the image came from) to combat the problem being too complex for the model.

User Interaction Scenarios

The learner: our platform is intended to provide a fun environment to increase one's geographical knowledge. By presenting the user with a different location each round, the user will likely slowly gain familiarity with the various countries/regions of Europe as they play the game. Since Google Street View semi-randomly places the user in a location (whatever is closest to the requested coordinates), the user will likely find themselves in intriguing locations which draw further curiosity. Upon seeing the true location for a given round, the user may then be inclined to further research the given area, or even look around a bit further in actual Google Street View. Further, the user may leverage the AI to gain geographic intuition. The AI will ideally pick up on nuanced details of the location images. We give the user the option to see the AI guess before the end of the round as a "hint" to inform their guess. For instance, the AI may pick up on Cyrillic text which the user did not immediately see and guess Eastern Europe. The user may then draw from their own observations, in combination with the AI, to submit their answer. In this sense, we believe our project offers an opportunity for the user to gain a glimpse of Europe (or whatever set of locations we provide) without leaving their home.

The gamer: GeoGuessr has grown rapidly in popularity since its creation. Our platform offers a rapid way to practice the GeoGuessr game given our current endless free-play setup. Further, our game encourages a sense of competition not present in actual GeoGuessr by providing an AI opponent. Now, the user can hone their GeoGuessr skills without having to rely on game lobbies or waiting for other human guesses. Players may now play at their own pace, rapidly progressing through different countries and locations only limited by the speed of the AI's processing power.

The cost-savvy GeoGuessr player: this currently provides a free implementation of the GeoGuessr game without a time limit. So long as our users remain below a certain threshold, our platform can provide a zero-cost way for the user to experience virtually all the GeoGuessr-like gaming they please.

Key Design Decisions and Final Design

Frontend:

The UX of our web app is a central piece of our project. Since this is fundamentally a game, our platform would not attract users if it did not present a comfortable and enjoyable experience.

Accordingly, we invested a significant amount of time into the UI design process. This, in itself, may be considered a key design decision, as we chose to divert attention to the front end of the game that could have otherwise possibly been devoted to model development. Nevertheless, we feel this division of our attention was worthwhile given the vital nature of the UX.

Within the UX itself, we made a number of key design decisions across the planning and implementation phases. Foremost, we chose to use the CSS framework Tailwind. Although no group members had substantial experience using the framework before this semester, we ultimately chose to implement this within our project given the potential for more pleasing styling as well as a more readable code base (which would ideally enable collaboration). Unlike many other CSS frameworks, Tailwind is a class-based framework in which styling is added via pre-set classes on the HTML components themselves. This way, developers would not have to keep track of styling classes in separate CSS files and they can directly see the classes applied to each element in the HTML code.

In terms of the UI itself, we made a multitude of design decisions. For one, we chose to maximize the size of the Street View Panorama. As the Street View will likely be the user's center of attention for most of the gameplay, it would make sense for the Street View to take on a central presence in our interface. In our first design, the StreetView took up the top half of the screen. Although this is still a significant amount of screen area, we felt we could improve upon this to provide a more immersive experience. In our second iteration, the StreetView now occupies the entire screen, with other components superimposed on the StreetView. Now, the map and other buttons have been placed directly on top of the StreetView in the bottom left corner of the panorama. A tradeoff of this decision is that the map/buttons do block some of the StreetView, however, this portion of the image would likely not be valuable to the user anyway as the most salient features of the panorama likely reside closer to the center of the image.

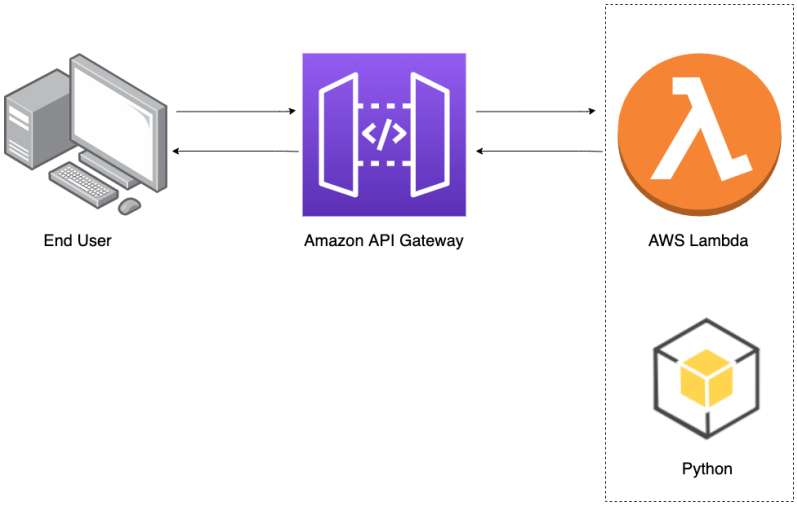
We hoped to ensure a comfortable user experience through several design decisions. For one, we limited the number of buttons/features presented to the user at a given time. In doing so, we intentionally kept the UI as basic as possible so that the user is not bombarded with features. The user is also highly constrained in their actions, as they may only click the map, move around, submit their guess, or see the AI guess on the opening page. At the end of the round, the map simply occupies the center of the screen, allowing the user to easily see the distance of their guess vs. the AI guess, as well as the outcome of the round. The user can then immediately jump back into the next round as the StreetView changes immediately (even before they hit play again). The entire UI also uses familiar components that most users have likely seen before (StreetView, GoogleMaps, Buttons). A tradeoff of the current minimalist design is that it may become boring for experienced users. Further, the minimal UI also currently limits the potential to display additional information to the user which could possibly inhibit the experience of those already comfortable with the app. For instance, those seeking to practice GeoGuessr could potentially benefit if the UI presented average guess time, historic guess accuracy, etc.

API

We wrestled with either uploading images first and then calling the model with pre-uploaded images, or downloading street view images as part of our backend API. For the prior, it would remain more true to the game as the model would see the exact same images as the user. However, during model training, we used static street view images, and the panorama is not of the same size, nor the same point of view as the static images. Additionally, uploading the images to S3 through a React interface was slow during our testing, and introduced additional bottlenecks as we would have to wait for the upload to complete in order to call the AI model. This also introduced additional costs for the cloud computing costs, and bandwidth for the user as well. Eventually, we decided to just download the images and call the model with a singular backend function. We used four separate images from four separate points of view to get a more accurate prediction and to encompass the different points of view.

We decided to use serverless functions for the inference API. This was done in AWS (Amazon Web Services) Lambda platform. There were a few things we considered. Either we could use serverless functions, have a continually running server (either in AWS EC2, or locally), or use a purpose-built ML tool such as AWS Sagemaker. Ultimately, having a continually running server was cost-prohibitive and didn't seem to be worth the decrease in latency that would provide over the other options. We would also have to configure EC2 manually, or on our own server. We didn't want the model on the client, as that would increase the bulk of the app, and we wanted to be able to write the backend code in Python, rather than interfacing with the JavaScript of the front end. Additionally, we decided against using Sagemaker as we felt that having serverless functions gave us more flexibility in how we would call the model. Eventually, due to the flexibility and lack of need to set up our own hardware, we went with serverless functions. The easy integration with REST APIs was also a plus.

Additionally, we call the model with four separate fields of views (north, south, east, and west), and take the highest probability country out of those. All in all, the final implementation that we decided on was an AWS Lambda Function that is accessed through an AWS API Gateway instance. The API Gateway takes a POST request that takes the latitude and longitude. From there, it calls the Google StreetView Static API in the four views and gets those images. It then feeds those images through the model and gets probability vectors for each country. It takes the highest probability per average country and gets a random city in that country and returns the latitude and longitude of that city with some uniform noise. The AWS Lambda Function is run through a container that was created using Docker and deployed using AWS Elastic Container Registry.



API Design [3]

Using AWS CloudWatch, we are able to determine latency for the total end to end API through API Gateway. We report mean, P95, and P99 latency figures. We note that the latency is still lower than the 30 seconds limit through API gateway.

Metric	Measurement
Average	2.547 seconds
P95	7.060 seconds
P99	10.129 seconds

We also looked at Lambda metrics, and the cost per use.

Metric	Measurement
Memory Used	510 MB
Latency Average	5.147 seconds
Cost Per Use	$0.174/1000 requests

Model design

One of the first design choices we made was choosing to train on the Google Cloud Platform and Google CoLab. This is because, for fine-tuning and training in deep learning, it is almost a necessity to have a GPU's computational power, so that we may accelerate the speed of learning. Using

cloud-provided GPU units serves our purposes (seeing as how none of us have the standalone hardware necessary). While this does mean we have to set up either the colab notebook or the GCP virtual machine, it is well worth it if it makes training faster.

As for our model architecture, since this is a computer vision task, we chose to use a Convolutional Neural Network (CNN), a tried and true method. We started off with some simple Residual Networks (ResNets) of varying sizes (18, 34, and 50 layers). However, these did not perform very well and after reading some research on the incorporation of transformer-like architecture into CNN models, we also investigated Visual Transformers (ViT) and ConvNeXT models. In the end, these did not pan out very well either and we pivoted to a custom architecture that produces a classification output and a reliability score, and is trained on patches of the images (see the appendix for more information).

Furthermore, since we are limited in both time and resources, we are going to be using pre-trained CNN models and then fine-tuning them to fit our dataset. This is very common in deep learning and allows us to use an already fundamentally sound baseline model (in this case, the CNNs are pre-trained on the ImageNet dataset).

We also spent a lot of time deciding on the choice of a loss function/output prediction for training/inference, as this will be crucial for our model to learn properly. The initial design choice was to have a simple Mean Squared Error loss for a simple regression output that computes a latitude and a longitude. However, we quickly realized that this was subpar (essentially just picking the mean. Curiously enough, it only performed extremely poorly for longitude). We experimented with changing the loss function to specifically suit Geoguessr (by using the point-based system the game relies on) but that was still subpar. After consulting with Dr. Pfister, we changed the problem definition to classification, and so used Cross Entropy Loss, and began focusing on predicting solely the countries of the images. This improved our model performance (which hovers at around 50% accuracy), but it is still not able to accurately predict the country of every single image. This is likely due to a bias in the training set, as we have constructed it from a list of Europe's most populous cities. The countries with more people and more cities will have the data skewed in their favor.

Our next design step integrated both regression and classification with the use of ViT's and ConvNeXT's. The idea here is that we want the model to predict both the country (as this is important for scoring in the game), as well as the latitude and longitude. This would supposedly reinforce a broad and narrow definition of the problem, as the model should hopefully be able to broadly choose the correct country, and then focus on and select an accurate geospatial location. As such, we combined both a classification-based loss (a weighted Cross Entropy Loss to combat the data imbalance) and a custom regression-based loss. After much training and testing, the results of this endeavor were not very promising (with test time accuracies of only 8-9%), and so we abandoned this approach for a custom architecture.

The final model design choice was suggested to us by Dr. Pfister, and involved the incorporation of a "reliability network" and chunking full sized images down into "patches" to go along with a typical

classification problem formulation. The “reliability network” was trained to weight the correctness of the patches – in other words, the reliability network would assign more weight to patches that have meaningful information in them (such as signs, words, etc.) and to reduce the weight of patches that are not helpful. Besides these two additions, the model was trained in the standard classification practice, using a ResNet-18 network as a feature extractor. Since our training was slightly different than in previous iterations, our inference works differently as well. Instead of using a single image to produce a single result, we use an aggregate schema for inference. First, we take 10 random patches from the image and use that to produce 10 classifications logits – one for each patch. We then multiply each of those logits by the weight output by the reliability network (which produces one weight for each patch). Finally, we sum up all 10 classification logits and use that to produce our prediction. This methodology greatly improves training time, but makes inference slower – a trade-off we were more than willing to take. Overall this custom method did improve our performance, but not enough for the AI model to be considered reliable. Specifically, the inference time accuracy is approximately 22% – which is much better than our initial endeavors, which peaked at around 8-9% accuracy.

Final Demonstration

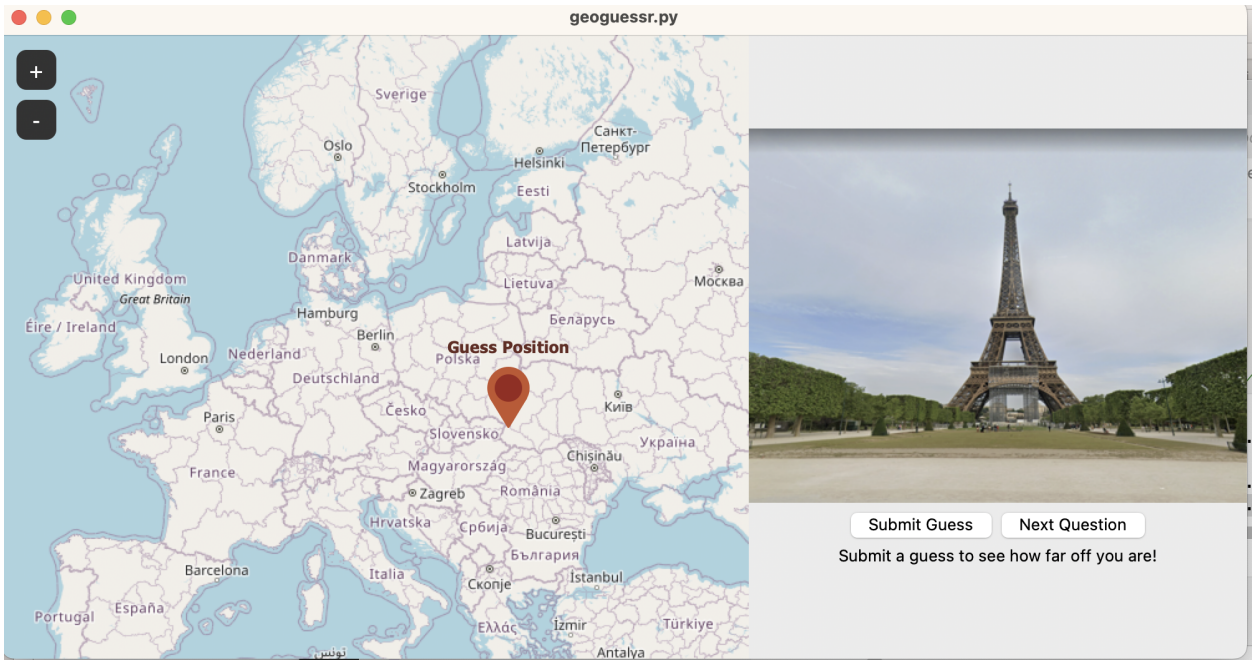
For the final demo we will play a game of Geoguessr live for the class to see. We will showcase the finalized website, complete with the finished game and the best ML model, to play a game of Geoguessr with the class. For every round, we’ll take guesses from the audience and then see how they fare compared to the prediction from the finalized ML model. Hopefully, they will be able to match up to our model. As we play the game, we will walk over some final implementation details about the website, the model, and the backend. Once we finish the game, we will leave room for questions from fellow students and from Dr. Pfister.

Appendix

Project Iteration

Front end

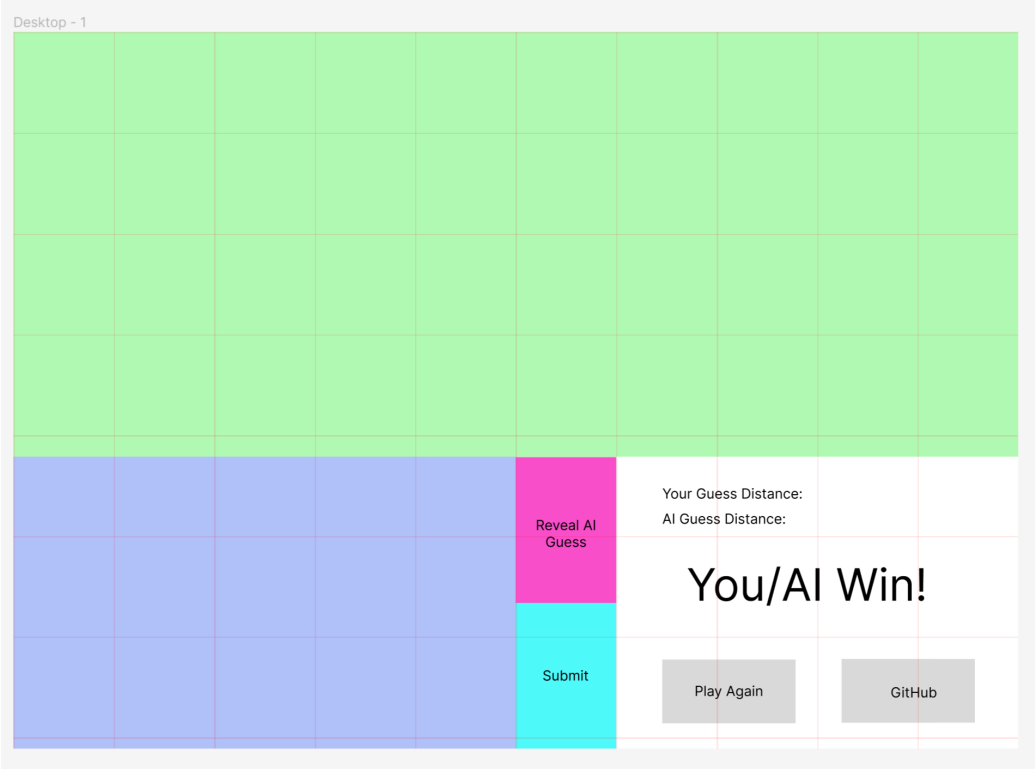
We first started by creating a low-fidelity prototype in Python. We used Tkinter for this as the front-end framework. This was done to first demonstrate the functionality of the Google Maps Street View API. We first used the static API for this that simply fetched single images and we guessed the location based on that. This had the additional benefit of allowing us to gain familiarity with the API which we would then use for data scraping. A screenshot of that is shown below. The user was able to submit a guess based on the right hand side image which was pulled from Google Street View, and submit a guess on the OSM (Open Street Map) based map on the left hand side.



From there, we decided to do more research into the Google Street Map API. We discovered that there was no good Python library that supported the dynamic street view API, and if we used Python we would have to use the static library. We decided against this as we wanted our front end to mimic the actual GeoGuessr front end as much as possible, and we wanted a dynamically controllable street view. Building a web app would also potentially allow us to deploy the game to the internet at a later point in time, enhancing its accessibility to a broader audience. Thus, we pivoted and decided to use JavaScript for the final project. We settled on React in JS as some team members had prior experience with this, and there were numerous pre-built packages that we could leverage for faster development.

Before we implemented the front end, we worked on figuring out what package to use to call the Google Street View API. First, we experimented with natively using the API provided by Google. However, this did not interface nicely with react off-the-shelf. We then experimented with several libraries that provided React Components that effectively served as wrappers for the Google Maps API, including react-google-street view and react-google-maps. We ultimately decided to use react-google-maps due to its widespread use which led us to believe it would be better supported and maintained. Looking back, it may have been possible to call the Google Maps API directly using external React Components which loaded the Google Maps JS script given an API key as input. This may have allowed for a greater degree of flexibility in our end-product as well as a cleaner implementation, however, the react-google-maps fulfilled our needs in most cases.

We then designed the first iteration of our project through the Wireframing software Figma. In the first design below, we attempted to construct an interface which met the requirements of a minimal viable product of our game. The green box would serve the role of the Street View through which the user could navigate. The blue box on the bottom right would serve the role of the Map which the user could use to submit their guess and see the AI's guess. After the end of the round, the user would be able to see the distance of their/the AI's guess as well as the round winner in the white box on the bottom right. The user would then be able to play again by clicking the play again button, which would reset the Street View and map on the lower left. At this stage, all of the components in the UI would be static to expedite the prototyping process.

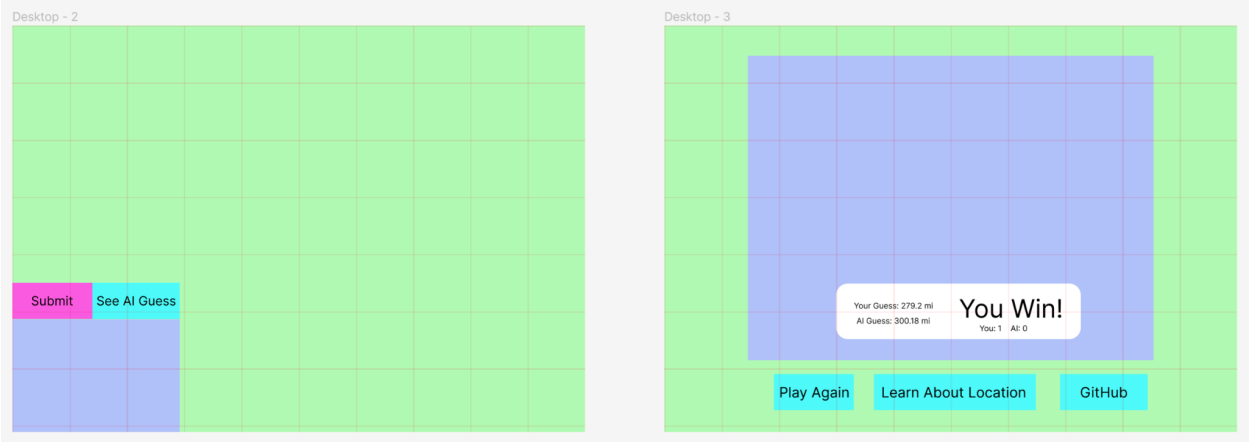


We then took our ideas to code with a React prototype. This prototype had the panoramic street view on the top, and on the bottom somewhere to put the guesses, submit the guess, and see the AI guess. The app drew from a fixed set of locations based on highly populated cities in the United States. Because the back end AI model was not fully implemented at this point, we simply predicted random locations for this iteration. We also allowed users to see the actual location. A screenshot of this first prototype in React is shown below.

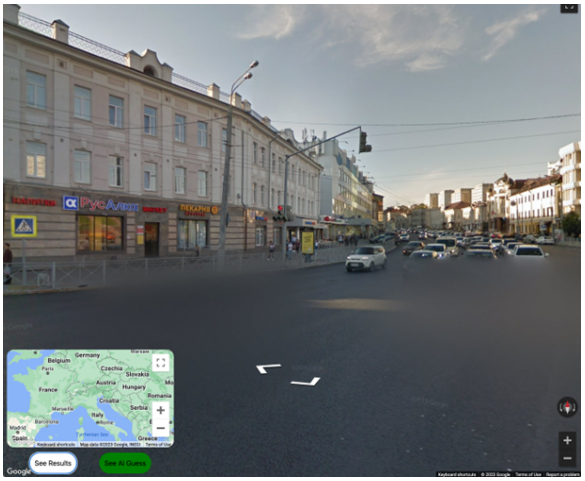
We felt the UI would serve a critical role in our project given the necessity of a strong user experience in a game environment. As such, in order to deliver the most visually pleasing UI possible, we chose to use the Tailwind CSS framework (despite having little to no experience). Tailwind CSS offers a set of pre-built styles which may be implemented via specific class names directly in HTML (for instance, the class “p-1” would create a padding of 1 px around a given element). Taking this approach to styling our web app removed the need for separate .css files as all styling took place in line. Further, the in-line styling may have better enabled collaboration, as all styling applied to a given component would be immediately visible from within the HTML (collaborators would not need to search through separate CSS files). Finally, the built-in Tailwind styling classes allowed for rapid changes and the general rapid implementation of styling decisions. Using Tailwind did, however, come at the expense of an initial learning curve and some initial hassle with setup.

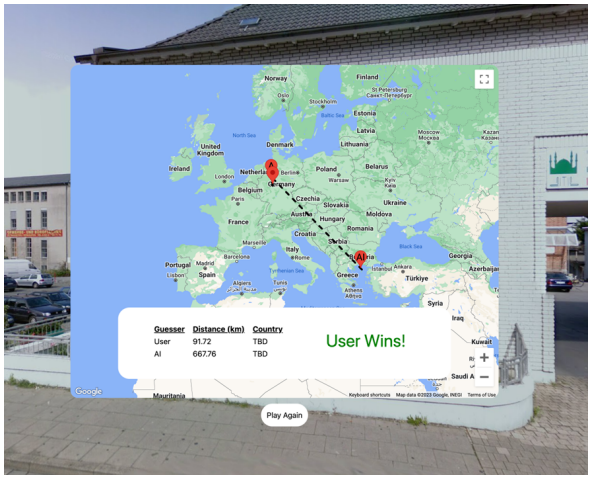
After this, we wanted to make additional front end UI changes. First, we integrated our current iteration of the AI model to the frontend. Initially, we configured the AI to connect to the backend Lambda endpoint. We will provide descriptions of further iterations of the backend later in this document. After implementing the AI for the first time, we noticed that the API calls were taking a long time compared to

our first “AI” which simply selected random coordinates. In order to avoid forcing the user to repeatedly wait for the AI guess, we requested and cached the location of the next AI guess immediately after the conclusion of a round. This way, the next AI guess would ideally be available whenever the user begins a round (assuming they progress at a reasonable speed). We also made a series of UI updates to enhance the playing experience of our game. We outlined our new ideas in the following Figma design below:



This iteration exhibited several improvements from the first design. First, the Street View now occupied the entirety of the user’s window, with additional components overlaid on top of the Street View. This way, the user could leverage their entire screen to explore a given location, offering a more immersive gaming experience compared to the half-screen approach from the initial prototype. Further, we split the UI into two “phases” corresponding to live gameplay and a round results page, respectively. As shown above, we felt this decision presented a better use of screen space. Now, the results were only shown at the conclusion of a round, rather than continuously occupying screen space. The map was set to move from the bottom left corner to the center of the screen upon a round’s conclusion to offer a clearer view of the user/AI guesses as well as the true location. We also leveraged the Google Maps API to place labeled markers on the map (rather than colored circles) and to draw dotted lines from the guesses to the actual location.



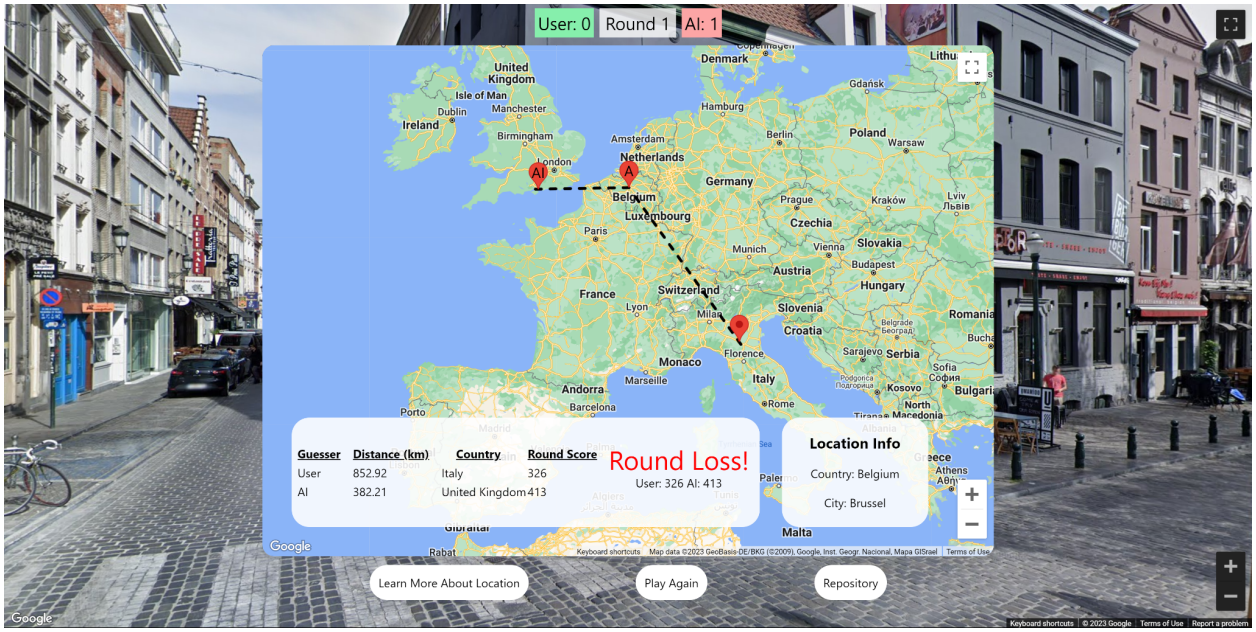


Before the final presentation, we added several more features to the UI. For one, we added rounds and a custom scoring system to build a better sense of a competitive environment. Each round, the user/AI score is computed as follows:

$$Score = 500 * \left[\mathbb{1}_{\text{country correct}} + \exp\left(-\frac{d}{2000}\right)\right]$$

In the above expression, d is the guess distance in kilometers and $\mathbb{1}_{\text{country correct}}$ is an indicator variable for whether the guess had the correct country. We used this scoring system instead of a purely distance based system as it emphasizes the player's ability to discern geographical features of individual countries. Since many countries in Europe are close together yet clearly visually different, this score equation helps avoid assigning excessive distance-based points due to luck. Now, the player's round score derives an equal weight from their guess distance (with the distance score exponentially decreasing with guess distance) and country correctness.

We displayed the running score of the game (in terms of round wins) at the top of the Street View window. We also added a "Learn More About Location" button that when clicked, attempts to open a new tab with a Google Search of the actual location city. In order to determine the actual location city and country from the location coordinates, we used the react-geocode module which implements the Google Maps Geocoding API. We display the actual location country (and city when available) in a separate "Location Info" panel on the results page. In addition, we added a button to see the GitLab repository.



Data Scraping:

The first problem we had to tackle after defining the project scope was creating a dataset for our machine learning model to train and test on. Since we decided on using a deep learning approach, the creation of a neural network that performs reasonably well as a classifier is dependent on the existence of an abundance of high quality training data. As such, a significant amount of time and effort was put into the development of this dataset. Here is an overview of the process by which we arrived at the final dataset (the one used to train and evaluate the neural network used in our final demonstration).

Since we are making a Geoguessr game, which relies on Google Street View images, the very first thing we tried was to just randomly scrape static street view images using the Google Maps API. This is the most straightforward approach because Street View exposes an API to get images from a specified location, i.e. we can collect training images along with their locations. We decided on using only static images because stitching together panoramic images (which is what the Dynamic Street View offers) yields a distorted image – likely due to the camera lens the interactive image is taken with. With this in mind, we scraped together a couple thousand static street view images pulled from random European coordinates (latitude and longitude) to serve as our first dataset. When we tried using this data, however, we quickly learned that it was essentially **useless**. The reason being is that these images held no useful information – more often than not they were pictures of trees, shrubs, random dirt roads: the scenery was so ubiquitous it was impossible to discern a location based off of them. Because of this, we quickly abandoned the idea of scraping the data using random coordinates.

After discussion with Dr. Pfister, we then investigated previously created datasets to see if any of them would fit our application. In particular, we looked into a dataset called *World-Wide Scale Geotagged Image Dataset for Automatic Image Annotation and Reverse Geotagging* published by Université Lyon.

This is a dataset of 14 million geotagged images crawled from Flickr (a popular image sharing website) – more than enough for our application. The main problem is that, although the images are geotagged, the pictures themselves are not limited to places and landscapes. The images can be of anything: books, people, food, etc. as long as they have geotags they are included in the dataset. Needless to say, those pictures are not useful to our problem, and if we want to use this dataset, we need to filter out these types of images. For filtering, we tried to manually pick a few representative example pictures and use them to train a simple feedforward neural network to classify images in the dataset that were "useful" or "not". This did not work. Firstly, it is very hard to pick good examples that could work for all 14 million images, and the problem of matching each picture with the possible example pictures would be very time consuming. Secondly, even if the classifier did manage to filter out the useful images (which would be just landscape images), the quality of these images ranges too wildly to be useful for our purposes – these images are often blurry, taken with dark lighting, etc. Because of these reasons, we decided to abandon the idea of using a previously constructed dataset.

Some other datasets we looked at were the YFCC100M Flickr Dataset (which was disregarded for similar reasons as above), and the Kaggle Geoguessr Dataset (this was ignored because the images were not geotagged and they contain artifacts from the Geoguessr game).

Our final approach, which is the one we ultimately used for training and testing, was to conduct semi-randomized image scraping using the Google Maps API. Instead of just feeding in random European coordinates, we instead randomly select a city from a CSV file of the top 500 most populated European Cities and add random noise to fuzz the coordinates. By doing this, we are sampling from areas with higher population density, which tend to yield images that have more useful things like buildings, signs, etc. Even if we sample the same city, the random noise will make it so that the image we scrape is from a different part of the city. This approach netted us with much better quality images than our first attempt.

Later in our model development cycle, we realized that the semi-randomized scraping approach yielded biased data (because countries with more cities that are heavily populated will have more images) and so we equalized the data so that each country had a similar amount of images.

Model Development:

Our naive approach was to first use a pre-trained model and fine-tune it to see if it could directly predict the latitude and longitude when given an image. This would provide a quick and easy way to see how difficult the problem formulation would be. As such, we used a pre-trained Residual Network (ResNet) model from the PyTorch Torchvision Hub and fine-tuned it to produce a regression output. The result of this initial approach was a model that simply predicted the mean latitude and mean longitude every single time (give or take some noise, also, curiously enough, the predicted latitude had much more variance than the longitude). In other words, regardless of the input image, the model would predict approximately the same location every single time – an indication that the problem is too complex for

the model to solve. This held true even as we changed the loss function to a custom regression loss based off of the Geoguessr scoring system and when we changed the base pre-trained models (we tried VGG, DenseNet, deeper ResNet's etc). Because of these reasons, we decided to try a different approach.

Our next approach was to instead formulate the problem as classification instead of regression. Initially, we tried fine-tuning the model to predict the 500 most populous cities listed above from an image, but quickly realized after some discussion with Dr. Pfister that this would be an enormously difficult task, and that we should broaden our scope a bit. So we settled on predicting just the countries given the street view images. Reformulating the problem as such increased our accuracy up to ~50%, but after closer inspection, we realized that this was a skewed statistic. Our data was heavily imbalance to favor the countries with more populous cities (like Russia, Great Britain, Spain, etc.) and so the model was actually only predicting these few countries every single time – it didn't really learn how to discern countries from the images, just what was best to minimize the loss function. After equalizing the dataset, the model performed quite poorly.

After seeing the pure CNN's perform so poorly, we tried seeing if utilizing state of the art transformer-influenced architectures would help improve our performance. Vision Transformers (ViT's) and ConvNeXT were the models we settled on. In this scenario we tried fine tuning pre-trained models to produce both a regression and classification output, the idea was to see if this would help reinforce a broad and narrow definition of the problem. The model would broadly choose the correct country, and then use that information to focus on selecting a correct geospatial location (that would hopefully be inside the country's borders). To do this we combined a classification based Cross Entropy Loss with a custom regression based loss and fine tuned the model on our final dataset. Unfortunately, this approach also did not work, and yielded subpar results. We suspect that these subpar results arise from our lack of training data in correlation with the problem difficulty – we don't have enough data to train a model of this architecture for a problem this hard. These transformer influenced architectures need millions and millions of images to achieve decent performance, which is something that we simply don't have the capacity to obtain.

After further consultation with Dr. Pfister, we decided to implement a custom architecture to tackle this problem, rather than using models off-the-shelf. The main focus of this approach was the utilization of "patches" of the image, i.e. smaller chunks of the image, as well as a "reliability network". The reliability network was trained together with the base classification model in an on-and-off schema (wetrain the classification model for one epoch, freeze those layers, and then train the reliability network for one epoch, freeze those layers, and repeat). The base model used for feature extraction was a ResNet-18 model. For more details, see the Key Design Choices section.

Model API:

As mentioned in previous sections our model API design was a dilemma between using dedicated ML cloud tools, serverless functions, and a hosted server. We decided on serverless functions using AWS Lambda. The model went through a few iterations, and thus the API went through a few iterations. Initially, we just fed a single streetview image and returned the result from that. Later, we took four separate images from the streetview, one for north, south, east, and west, and fed those into the model and took the average coordinate.

After the model started predicting a country as a classification task, we took the highest combined probability country with the four separate countries and returned that country and a randomly selected city from that country. As we don't return the latitude/longitude in the new model, it is impossible for us to return any specific city so we randomly select one. We initially just used the final prediction, not the probability vector, which gave us a better prediction.

Team Contributions

Evan

My work revolved around the front-end of the project. I entered this semester with some experience with React JS, however, this semester offered an opportunity to broaden my general web-development skills within a project that integrated an AI. I began the semester by learning about figma and developing a basic mockup of our project. I learned how to integrate Tailwind CSS with our project and made most styling decisions across the frontend. I designed and built most of the front-end infrastructure, including each of the major components (the guessing map, the results page, interactive components, etc.), the gameplay logic, and general game flow. Further, I incorporated built-in React features in our design, including hooks which allowed for the UI to immediately display changes in game state. Over the course of this semester, I spent substantial time learning about new web development concepts and tools, including Figma, Tailwind CSS, API requests (and associated asynchronous function calls in JS), React Hooks, the Google Maps API, and additional external React components.

Billy

I worked on the initial python prototype and did some initial research into the Streetview API. After that, my main task was working on the inference API and connecting that with the front end. I set up the AWS infrastructure and the Docker containers needed to make that work. This includes the ECR (elastic container registry), API Gateway, AWS Lambda, and logging associated with this. I also set up the Rest API and debugged that, along with the react-geocode package. Finally, I made some minor front end improvements as needed, mainly by fixing small bugs that I found, and tested the front end as well. I also did some work into seeing if S3 uploads were feasible before determining that they were not.

Aben

I was first focused on potentially handling the data storage and backend model deployment configuration. To that end, I was researching into converting between Tensorflow based and Pytorch based models using protocols like ONNX. In the end, we ended up using a colab based Tensorflow model development, so exchanges were not necessary. I looked into potential ways of generating more data for training multiple times as it became apparent that a reasonably performing model would need a massive corpse of training data. As such, I tried to first write scraping scripts on google images to get more data for a given city. This ended up being quite useful since google randomizes their images page returns to prevent any significant scraping. I further went on to bing images, but it turned out the images returned were much less helpful and off quality. Besides that I looked into making massive image datasets (such as the Flickr dataset) we have discovered into useful inputs for training. When it became apparent that we just don't enough blanched data, I looked into solutions like prior-normalized classification (Prof. Pfister's suggestion), Fake data generation, and Balanced data generation. I discovered none of them provided much merit, at least within our computing resources. To sum it up, my contributions ended up mostly trying proposed solutions out and discovering they don't quite work.

Eric

I worked a lot on researching the usage of Google Maps API, specifically Static Street View for both the initial python prototype, as well as the construction of our dataset. After constructing the dataset, I set up the training and evaluation of the neural networks using PyTorch on Google Cloud Platform (which includes configuring the VM for deep learning), before later migrating to Google CoLab (which I have a subscription to) due to some CUDA issues. Research into other datasets was conducted by Aben and I, but my main focus was on the training and testing of models – as well as brainstorming potential ways of improvement by conducting in depth investigations of different model architectures.