# CS109: The Probability behind Geoguessr

Hinson Chan

## Introduction

I play a lot of this game called GeoGuessr. The premise is simple; given any streetview image on Google Maps, deduce where in the world that image was taken. This is also one of the games where the difference between a novice who has never played before and a professional is huge; even after playing thousands of games myself, I still struggle to differentiate between regions of most countries and even similar-looking countries.

For most professionals, on the other hand, the game is actually a game of mental probability. A single image will have many features to look for, scattered unpredictably in places most people wouldn't know to look.

## Basic Probability

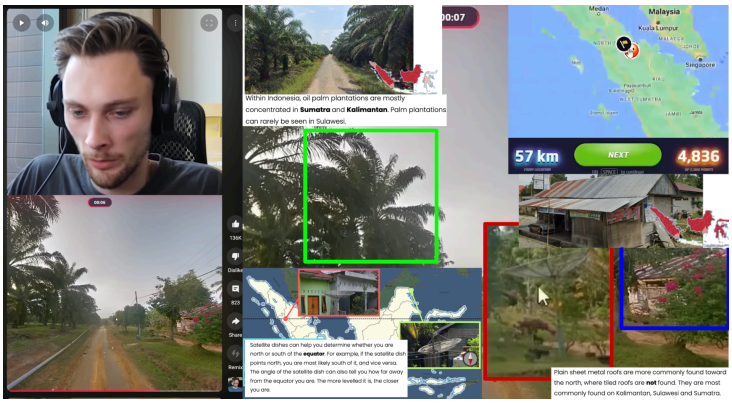



Figure 1: Distributions of features in Indonesia narrow down North Sumatra.

Here, a professional, in less than 10 seconds, combines his knowledge of palm tree plantations mostly being in Sumatra and Kalimantan, with the satellite dish pointing to a point near the equator, along with the tin roofs to narrow it down to north Sumatra. In probability terms, if the professional had noticed all of these things and was trying to pick a country, they would make the following decision:

$$P(\text{Indonesia} \mid \text{image}) = \frac{P(\text{palm oil} \mid \text{Indonesia}) \times P(\text{sat dish} \mid \text{Indonesia}) \times P(\text{sheet roofs} \mid \text{Indonesia})\ldots}{\sum_{\text{country}}^{\text{all countries}} P(\text{palm oil} \mid \text{country}) \times P(\text{sat dish} \mid \text{country}) \times P(\text{sheet roofs} \mid \text{country})\ldots}$$

In other words:

$$P(\text{guess\_location} \mid \text{image}) = \frac{\prod_{\text{feature}}^{\text{all features}} P(\text{feature} \mid \text{guess\_location})}{\sum_{\text{location}}^{\text{all locations}} \prod_{\text{feature}}^{\text{all features}} P(\text{feature} \mid \text{location})}$$

At first glance, a player will probably arrive at a few main location candidates in a split second, before slowly going through the features with these candidates in mind. Most players have the distributions of everything from camera generations, car types, crop farms, and more memorized! This might insane, that every player makes this complex probability calculation on each map; but this is actually how players play the game. Approximating, or "hedging" (real term!) their guess allows them to maximize the probability that their guess is as close to the target as possible.

## Can I build a GeoGuessr AI?

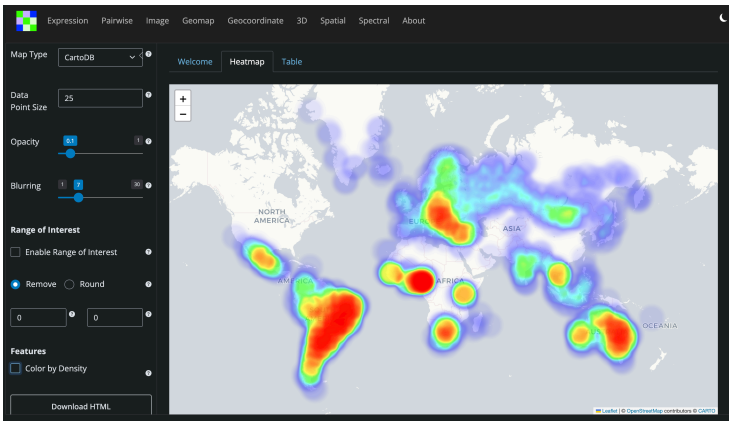
Yes, of course! In fact, the best GeoGuessr models today are already much better than players, and in fact rarely rely on the same specific features that humans do. This is what makes modelling GeoGuessr so interesting! It is more optimal for humans to remember specific facts about places, which can either result in a hedge between several probable locations, or a precise 'pinpoint' (memorizing the exact road or city with certainty). However, such methods often cause humans to 'overfit' to the Google Streetview images, and cannot generalize easily to geolocation of other places. Models, on the other hand, are great at crunching through an absurd number of datapoints and genuinely building a continuous latent space across the Earth;

model guesses can be tuned to precision and are extremely consistent, but lack the sponteneity that humans do to pick out specific locations.

For this challenge, I built a simple Multi-Layer Perceptron to attempt to play this game. While I probably would've had a better model if I had the flexibility of using a library like torch, I chose to write all my layers entirely from scratch. That definitely made the challenge much tougher, and restricted the extent of the architectures that I had access to, but I was surprised that even just a tiny model with only a few million parameters could do pretty well!

My video will go over more of the process, but in principle, I wanted to approach the game a little differently than just a single guess. When playing, the goal is to pick the point as close to the real target as possible, but instead, I modeled the 'target' as a 2D gaussian distribution with an adjustable sigma, or 'spread' value. My hope was that with this architecture, the model would pick up on similar-looking countries, or even allow for more interpretability with a model's guess.

I trained on a small, 10,000 image dataset (from Kaggle), which had this distribution:




One key thing to note is that not every country in GeoGuessr exists on Streetview, and certain places just have a lot denser coverage. The uneveness of this dataset (and several datasets I tried) meant the resulting guesses inadvertently ended up looking a lot like the dataset distribution.

Several key modifications to the standard MLP had to be made. For starters, I couldn't just start with several fully-connected Dense layers; that would make the model way too large. Even just 640 x 640 x 3 was already 1.2 million neurons...even a small layer after that would result in a model of hundreds of millions of parameters! Instead, I created a new strategy of 'patching' the image into 1600 patches, each 16 x 16 pixels large. Then, each patch would get 800 weights in the first layer, and their dot product would be a single neuron output. This meant I could represent over a million numbers with only 1600 neurons!

Secondly, I wanted to ensure that I could punish the model adequately for guessing too often, or too far. I started with KL divergence, which was a good start. In the equation,

$$
\begin{aligned}
D_{KL}(p\|q) &= \int p(x)\log\frac{p(x)}{q(x)}\mathrm{dx} \\
&= \int p(x)\log p(x)\mathrm{dx} - \int p(x)\log q(x)\mathrm{dx} \\
&= C - \int p(x)\log q(x)\mathrm{dx}
\end{aligned}
$$

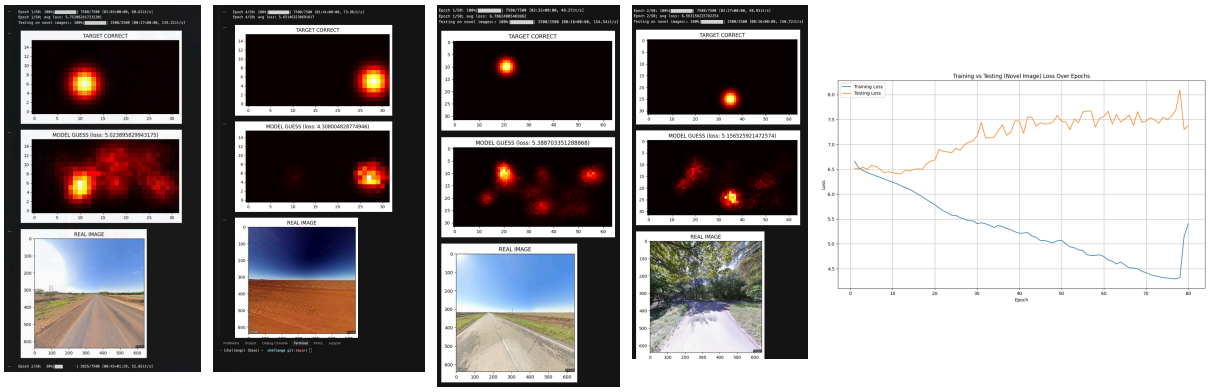
this would effectively be cross-entropy, since the entropy of our dataset is constant with respect to training. However, I suspected that since the target $p$ was often just empty black space, if the model just guessed indiscriminately a blob that just matched the input distribution of the dataset, it would only be punished minimally for guessing in places that the target distribution was zero (black). I attempted several other things, like a forwards KL + backwards KL:

$$\text{loss} = -\int p(x)\log q(x)\text{dx} - \int q(x)\log p(x)\text{dx}$$

as well as including mean squared error:

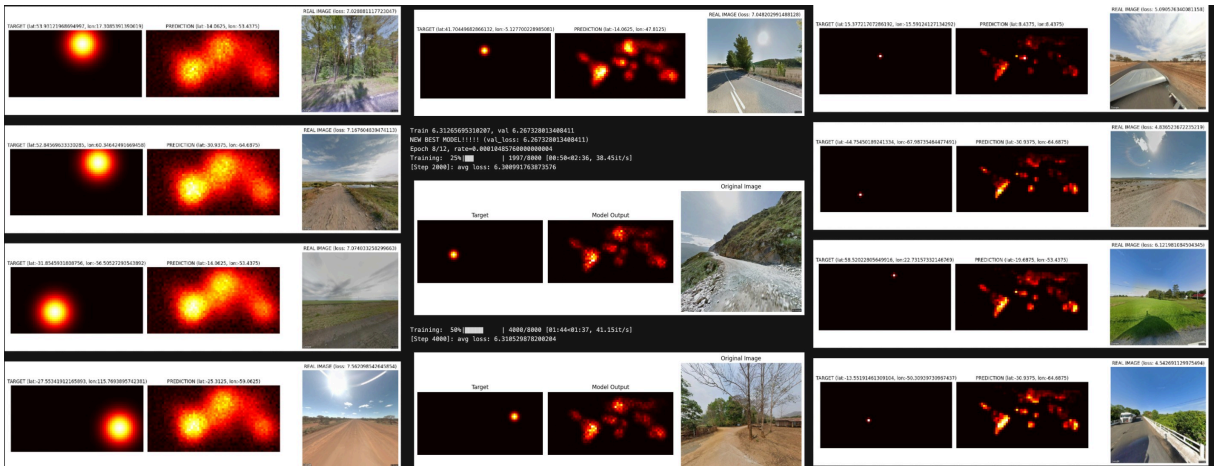
$$\text{loss} += \int (q(x) - p(x))^2$$

Unfortunately, the model didn't change very much even after trying different loss techniques. Regardless, here are some of the results!




The images actually look promising; my model seems to be pretty good at determining Brazil and Australia. Brazil usually has a distinctive red soil, short vegetation, and asphalt road, while Australia is even more distinctive. It even gets the tough South Africa guess at the end! However, the loss graphs show that the model learns its training dataset very well, getting to a point where it almost memorizes the trainset, while the test dataset drops for a bit, then slowly increases as the model overfits to the training set.

At this point, I add in L2 regularization and dropout, attempting to force the model to stop overfitting to my test dataset. Unfortunately, while the train/test loss now looked better, the model just ended up guessing the entire distribution:




Here, I vary my value of sigma that produces the target (how spread out the Gaussian target is) and it's clear that the tighter the Gaussian point, the more the model adheres to the shape of the Earth. Sadly, it doesn't seem to stop it from guessing the target distribution either.

## Conclusion

So, could I really consider that a success? In the state that my model is at, it's definitely nowhere near usable by a player. Ironically, the model that overfit on it's training set produces "nicer" looking guesses, but definitely can't for predictions at all. If I were to approach the problem with no constraint of hand-writing my layers, I probably would've experimented a lot more with vision transformers and CNNs, or even having it directly predict a top N set of points that could form the distribution instead. Regardless, given the difficulty of

this project, I'm surprised that my model was even able to produce a coherent result. And if you've read this far, I'm sure you're looking for what the best models look like. I'll link you to these ones here:

- https://lukashaas.github.io/PIGEON-CVPR24/ Created by Stanford students, and one of the earliest examples of a GeoGuessr model that beats humans!
- https://huggingface.co/geolocal/StreetCLIP An approach that leverages image embeddings and vision transformers, but still specific to street view.
- https://nicolas-dufour.github.io/plonk My personal favorite paper (of all time), which beautifully approaches the problem from a physics standpoint, and is trained on all sorts of images, not just streetview! This was the paper that inspired me to approach the GeoGuessr challenge from a distributional approach.

Project page: https://github.com/flatypus/geonets