Documentation

Filip Wolf Leonardo Čuljak Borna Katović
Matej Ciglenečki

Contents

The problem **1**
Computer Vision 2

Solution **2**
Technology stack 2

Data and feature engineering **3**
Classification approach 3

Model **4**

Training **5**
Basics 5

The problem

We were given a set of Google Street View images taken in Croatia. This problem is inspired by the game called GeoGuessr. In GeoGuessr, you are given an random image of the Google Street View location for which you have to infer the coordinates. In this competition, we were given images that come in quadruples which form a non-continuous 360° view of the location. Non-continuous 360° view means that images can't be connected together to from a perfect 360° panorama. <TODO: insert image>. Each image represents a cardinal direction (north, south, west, east) from the Street View car. Alongside images themselves, we also received location in the form of latitude and longitude pair for each image quadruple.

Our task is to predict the coordinates of the images we will receive in the last week of the competition. Although obvious, it's important to note that we will not receive coordinates for those images. After we provide predicted coordinates for each image quadruple, the error will be measured using the Great-circle distance between predicted and true coordinates. The great-circle distance measures the distance between two points over a curved surface (e.g. distance between two cities where the curved surface is the Earth). Total error is calculated as the mean of all great-circle distances between true and predicted coordinates. It's also possible the explain the error in the following way: the further we have to drive the imaginary car from the predicted coordinate to the true coordinate, the larger the error. The total error will be used to determine how successful our method is compared to our competitors' methods.

On paper, the problem sounds fairly simple and is not unlike many other computer vision tasks. However, let's explore the following situation: a country can look very similar over large swathes of land. If we were randomly placed somewhere in the area of Gorski Kotar it might feel impossible to predict our exact (or even approximate) location in the area of Gorski Kotar. Unless we recognize the landscape we've already seen or we notice some obvious features of the location, such as a town sign or a famous statue. There is a silver lining to this though. Croatia, although small, is very geologically and culturally diverse. Mountains, houses, forests and even fields can look different depending on the region of the country, giving precedence to the idea that the model could catch these differences. That being said, it is still a difficult problem to solve and requires clever feature engineering and careful network setups in order to work, which we will talk about in the coming chapters.

Computer Vision

Computer vision is an area of research that has seen the most growth from the advent of deep learning. Over the past decade, it grew from a niche research area to one of the most widely applicable fields within machine learning. In computer vision, we use neural networks to somehow analyze a large number of images, extract some potentially useful information from them, and use that information to classify those images into predefined classes. The problem we were tasked with solving in this competition falls neatly into this category.

Solution

Technology stack

Before diving in various components of the solution, technology stack we used will be described briefly.

- `python3.8` - the main programing language used for the project
- `git` - version control system

Python packages: - PyTorch - The open source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab - PyTorch Lightning - PyTorch framework which is integral library that allowed us to skip the boilerplate and organize PyTorch code in a sensible and efficient way - `black` - code formatter - `aiohttp` - Asynchronous HTTP Client/Server for asyncio and Python. Used for sending/receiving asynchronous requests when calling Google's Street View API - `geopandas` - Pandas for geospatial data. Used to wrangle, manage and generate geospatial data - `imageio` - Write and read image files - isort - Sort *.py imports - matplotlib - Visualization with Python - NumPy - mathematical functions and managing multi-dimensional arrays. Used for everything - Pandas - Python Data Analysis Library. Used for loading, managing and decorating* .csv files - requests - scikit-learn - Shapely - tabulate - tensorboard - tqdm

Data and feature engineering

This problem can be approached from two distinct perspectives. We will call them **Classification approach** and **Regression approach**. In the Classification approach, we classify images to fixed set of regions of the Croatia (notice: here we lose the information about image's exact location) while in the Regression approach we try regressing the image coordinates to a continuous output from the model that will be restricted by the minimum and maximum possible coordinates (bounds of the Croatia).

Classification approach

The set of regions of the Croatia we mentioned above are in fact polygons on the map where each polygon corresponds to a single class. The idea is that instead of predicting the *exact* coordinates of the image, the model classifies the image to some region from the set of regions we created ourselves. Since we have to provide the predicted coordinate we declare it as the centroid of the region where the image was classified. Notice that image's true coordinates might be distant from the centroid of the region for which the image is classified.

How is this set of regions created? First, we create a grid inside of the bounds of Croatia. The grid contains many polygons (squares) which are located next to each other. Now, we have fixed number of polygons, however not every polygon should be taken into consideration, rather only the ones which intersect with Croatia's land. Therefore, we filter out the polygons which don't intersect Croatia's land. After we find all polygons that intersect with Croatia, we proceed to find centroids of those polygons. With the `geopandas` library this problem is reduced to a single expression: `polygon.centroid`. But before continue with the marathon, we should double-check our laces. Let's check the following example <TODO: image of centroid in the sea>.

Even though the centroid of the polygon is calculated correctly (we don't doubt `geopanas`) it doesn't make sense to declare that the image's coordinate is located somewhere at the sea (as we know for a fact that the dataset contains only the images taken on the land). We somehow have to modify our declaration method of the image's predicted coordinates. We introduce **clipped centroids**. Clipped centroid is the same as the original centroid unless the original centroid is not intersecting Croatia's land. In that case, we clip the centroid the the closest possible point on the land. Benefit of this method is that we are avoiding the unnecessary error which would often occur on the coast.

Notice that it's possible to specify the number of classes before creating the grid and thereby making it more or less dense. By choosing a dense grid, we can essentially simulate something closer to regression. As the number of classes increases (granularity decreases), more polygons and centroid values are available as potential classes. If we go to the extreme, a theoretical infinite number of classes which might feel similar to regression but it's not. This extreme has its problems; there simply aren't enough images per class for the model to effectively train. Another problem arises because of Croatia's pointy shape. Because of it, for some polygons, the intersection area with Croatia's land is only a few percentages of the polygon's total area . This means that the majority of the polygon's area ended up in the neighboring country with only a side of the polygon touching the territory of the Croatia. This isn't a major issue as we use clipped centroids, but there might be no images in the dataset that are located in that polygon. We did in fact end up in situations where some of the polygons don't contain any images. In that case, we discarded those polygons and pretended like they didn't belong to Croatia. Fortunately, this doesn't happen too often as the dataset is large and locations are uniformly distributed.

Model

There are numerous approaches that are considered state-of-the-art at the moment, all using completely different architectures. Because of readily available high performance models that were pretrained on large datasets being the norm, we also followed this route. Originally, we chose the EfficientNet architecture because of it showing both goof performance and having lower system requirements when compared to other approaches. However, after doing some experimenting, we ended up using a version of ResNet instead, as it simply proved more effective. Of course, we also had to modify these architectures in order to make them work with our dataset. There were two modifications we made. Firstly, we removed the last layer of the network and replaced it with our own classification layer in the form of a simple linear layer that had the appropriate number of classes for our problem. Secondly, because, as we mentioned before, every location contained 4 images, we modified our network to perform its forward operations for the four images separately and then concatenate the outputs before imputing them into the classification layer. We did this after doing some research on what was the best way to compute the outputs of separate, but statistically linked images.

Training

Basics

Due to using pretrained models, we first performed fine-tuning. ResNet was pretrained on ImageNet, a large image dataset with diverse objects. This gives the early layers of ResNet a collection of learned shapes and lines that are relatively similar to our own domain. In addition, our own dataset is much smaller than the number of ResNet parameters. If we were to train ResNet from scratch, we would quickly overfit. By using a model pretrained on a large generic dataset and then fine-tuning only the last layers, we preserve all the fine detail learned by all the early network layers and only overwrite the last layers where we essentially assemble these details into images. However, after performing fine-tuning for a sufficient time, we periodically unlock all the other network layers so that they can be trained along with the last few layers. By using a sufficiently small learning rate, we induce our dataset information into all the network layers without overwriting them.

Todo:

Datasets - split

Connecting datasets (data module): - hashing, mean, std, caculations

Training - - dataset

External sampling