Documentation

Contents

1 The Problem

1.1 Introduction

We all know and love the famous online game *GeoGuessr*. You are dropped in the middle of nowhere in *Google Street View* and are tasked with determining your location as closely as possible. “Am I on the North or South Pole?”, “Is this the African or South American jungle” and “On which end of Russia am I?” are only some of the questions players are asking while playing. Fortunately, the problem we were tasked with is a little simpler (keyword: *a little*). Instead of playing GeoGuessr on a global scale, we play it only within Croatia. To do this, we were given a set of Google Street View images taken on Croatian roads along with their coordinates. The images come in quadruplets forming a non-continuous 360° view of the location, where each image was taken in a cardinal direction (north, south, west, east) from the perspective of the Street View car. Alongside the images themselves, we also received their locations in the form of latitude and longitude pairs for each set of four image. An example of the four images is displayed in Figure 1.

Figure 1: Example of four Google Street View images taken in a single location. Our model will train on these images.

1.2 The Task

As we previously stated, our task is to predict the coordinates of the images, specifically, the coordinates of the test images we will receive in the last week of the competition (kept of course in the most secret government bunkers, shut away from our prying eyes). It’s important to note that we will not receive the true coordinates for these test images. After providing predicted coordinates for each set of four images, the total error is measured using the **great-circle distance** between the predicted and true coordinates. The great-circle distance measures the distance between two points over a curved surface, e.g. the distance between two cities on the Earth’s curved surface, and it uses the *haversine formula* to calculate this (fortunately for you, we won’t go into detail about this). Total error is calculated as the mean of all great-circle distances between the true and predicted coordinates for all the images. It’s also possible to explain the error in the following way: the further a bird needs to fly from the predicted coordinates to the true coordinates, the larger the error. The total error will be used to determine how successful one method is compared to others.

On paper, the problem sounds fairly simple and is not unlike many other computer vision tasks. However, we are faced with the following problem: a country can look very similar over large swathes of land. Turns out, the grass is green and the sky is blue wherever you are in the world. For example, if we were randomly placed somewhere in the area of Slavonia and were told to say where we are located exactly, it might feel impossible to predict our exact (or even approximate) location (the unending flatness might give us a clue though). Unless we’ve already seen the landscape or we notice some obvious features of the location, such as a town sign or a famous landmark, there is little chance for us to correctly predict our whereabouts. There is a silver lining to this though. Croatia, although small, is very geologically and culturally diverse

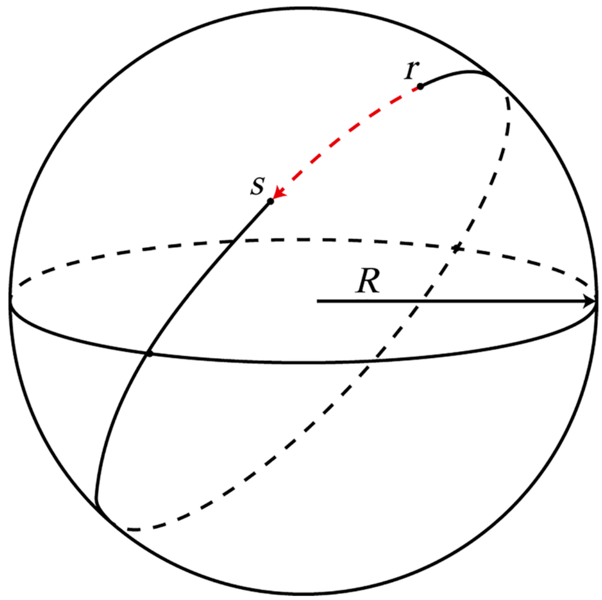

Figure 2: An image of a sphere where the dotted red line represents the great circle distance between two points on the sphere's surface. Notice that it is alarger distance than a direct straight line through the sphere.

(thank you centuries of non-independence). Due to this, 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 learn to spot these differences. That being said, it is nonetheless a difficult problem to solve and requires clever feature engineering and a careful neural network setup in order to work, which we will talk about in the coming chapters.

1.3 Computer Vision

Computer vision is an area of research that has arguably seen the most growth from the advent of deep learning, being right up there with meaningless buzzwords and failed startups. 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 analyze a large number of images, extract some potentially useful information from them, and use that information to classify those images into predefined classes or predict a target variable. It can be used in almost anything, from detecting traffic sings for self-driving cars to distinguishing fake works of art from real ones. The problem we were tasked with solving in this competition falls neatly into this category.

1.3.1 Convolutional Neural Networks

What makes computer vision distinct from other fields of deep learning is its usage of **convolutional neural networks** and its unmatched ability to infringe on privacy. Here, we will focus on the former. Because of the fact that images are 2-dimensional data structures, we can't just turn them into 1-dimensional vectors and feed them into our network for training without inducing major information loss. Due to this, CNNs use 2D *filters* to capture image information. A filter is a matrix of learnable weights that slides across images to detect patterns. Ideally, each filter in a CNN layer would specialize in learning different image features, such as lines and contours, or eyes of animals. By stacking multiple layers of these filters, each layer can learn a higher order of abstraction of the image data. For instance, the first layer might only learn to recognize simple lines. The second layer might combine these lines into shapes, while the third layer could finally combine these shapes into something recognizable, like a car. Such an architecture mimics how the human brain actually recognizes objects, that is to say, by combining smaller elements we see into larger ones. After we have stacked enough convolutional layers for recognizing objects, we finish off the network by feeding all the results into a fully connected layer for classification or regression and train it with any generic loss function.

A large advantage of CNNs compared to other network architectures is their interpretability. The filter weights can be visualized to depict what each filter detects in an image, while network weights depict how the filters are combined. This can help us in understanding how the network learns from images. (IMG: insert filter image)

2 Solution

2.1 Technology Stack

Before diving into the various components of our model, the technology stack we used will be described briefly.

- `python3.8` - the main programming language used for the project, everyone and their mom uses it so no explanation needed here
- `git` – the quintessential version control system

Python Packages

- `PyTorch` - an open source deep learning framework based on the Torch library used for applications such as computer vision and natural language processing. Although it is primarily developed by Facebook's AI Research lab, we can assure you that it does not collect any data from your computer
- `PyTorch Lightning` - a PyTorch framework which allowed us to skip a lot of boilerplate code and organize PyTorch code in a sensible and efficient way
- `black` - code formatter, so we're all on the same page (pun intended)
- `aiohttp` - Asynchronous HTTP Client/Server for asyncio and Python. Used for sending/receiving asynchronous requests when calling Google's Street View API (this was written by the API guy and I have no idea what it means)
- `Pandas` - the popular Python data analysis library. Used for loading, managing and decorating *.csv files
- `geopandas` - Pandas version used 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 management of multi-dimensional arrays. Does anything even run without this?
- `requests` - a HTTP library for Python. The goal of the project is to make HTTP requests simpler and more human-friendly
- `scikit-learn` - a free machine learning library for Python. It features various classification, regression and clustering algorithms
- `Shapely` - a BSD-licensed Python package for manipulation and analysis of planar geometric objects. It is based on the widely deployed GEOS (the engine of PostGIS) and JTS (from which GEOS is ported) libraries
- `tabulate` - easy and pretty Python tables
- `tensorboard` - library used for fetching and visualizing machine learning model training data in a browser
- `tqdm` - easy Python progress bars

2.2 Solution Architecture

Our model was composed of four main modules, as well as numerous helper functions. The helper functions were mainly used for tasks that were separate from the training process itself, such as transforming data into a format appropriate for training, generating classes, visualization, saving and loading, etc. Even though they probably do most of the work, we will here promptly ignore them and explain only the interesting parts of our solution, namely, the four main modules.

2.2.1 GeoguesserDataModule

The first module we will describe here is the `GeoguesserDataModule`. It inherits the `LightningDataModule` class and is used for simpler dataset loading. First, we load the dataset into memory and perform the necessary transformations on it to make it suitable for training. We then create the classes needed for classification together with their centroids, and add them to the dataset. We also perform some dataset statistics extraction, as well as some data standardization. Finally, we split the data into the train, validation and test dataset and create the dataset instances that will be described in the next paragraph. We also do some sanity checking to make sure everything is in order before continuing.

2.2.2 GeoguesserDataset

To make training simpler, we define datasets for the train, validation and test part of the dataset separately. This is done by creating instances of the `GeoguesserDataset` module which inherits the standard `PyTorch Dataset` class. The most important operation that this module performs, aside from saving the dataset and its information, is define what happens when an instance of the dataset is retrieved. This includes fetching the data from both the image folders and the CSV file, loading the images, making them ready for input into the neural network, and transforming them with the specified transformations.

2.2.3 Model Definition

The neural network model that is going to perform the training is defined in the model module. The different models we defined here (classification and regression) all inherit the `LightningModule` class to make things as simple as possible. Each model fetches a pretrained network and changes its last layer to a layer more suitable for the model's task. We then define each model's `forward` function, which defines how our data is processed through all the layers of the model, as well as define and all of the training, validation and testing functions. Each function performs an iteration of training through its respective dataset and calculates error, and additionally in the case of validation and testing, the haversine distance function. We then log all the parameters of the model into a file for later fetching and usage.

2.2.4 Train Function

Finally, we join everything together in the `train` function. Here, we parse the arguments passed to the model, define the data transformations, initiate the `GeoguesserDataModule` module, initiate the logger, create the specified model, define our optimization algorithm and how the learning rate behaves, and finally, train the model. Optionally, we can also visualize our results after training.

3 Data and Feature Engineering

The problem we are facing can be approached from two different angles. Fearing encroaching on any originality, we will call them the **Classification approach** and the **Regression approach**. In the classification approach, we classify images into a fixed set of regions of Croatia in the form of a grid on the map (notice: we lose the information about the image's exact location here), 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 Croatia).

3.1 Data Representation

Because Croatia is a small country (though not as small as Slovenia) and its coordinates have a range of no more then a few degrees in both latitude and longitude, we needed to normalize them. But, before doing that, there was another transformation we needed to perform. The thing is, we can't use the haversine distance function during the training phase of the model due to its slowness (it contains a lot of trigonometric operations that don't cooperate nicely with GPU-s). But we also can't use regular coordinates even after transforming them because the Earth's surface is curved (even though some would want you to believe otherwise) and generic loss functions don't take this into account. Multiple ideas were tested to solve this, including cosine transforming the coordinates, as well as projecting them into Cartesian space. Finally, we ended up using a much more elegant solution with fewer steps: the *coordinate reference system (CRS)*. Here's how it works. It transforms every coordinate on Earth's curved surface into a different coordinate on a flat surface using a projection. It is expressed in meters and the error of the projection is minimal (no more than about 1 m). Maybe most importantly, a generic loss function is directly applicable to these coordinates, or at least, the standardized version of them. Splendid! Now we simply need to find the maximum and minimum coordinates of the dataset and use them to normalize the data into a 0 – 1 range. This is done to improve training stability and simplify the output. It is worth noting that we calculate these values only on the training part of the dataset, as using other parts of the dataset would essentially give the model access to their information, which wouldn't be fair at all. Also, functions like the haversine distance function take as input radians, so that was also a necessary transformation.

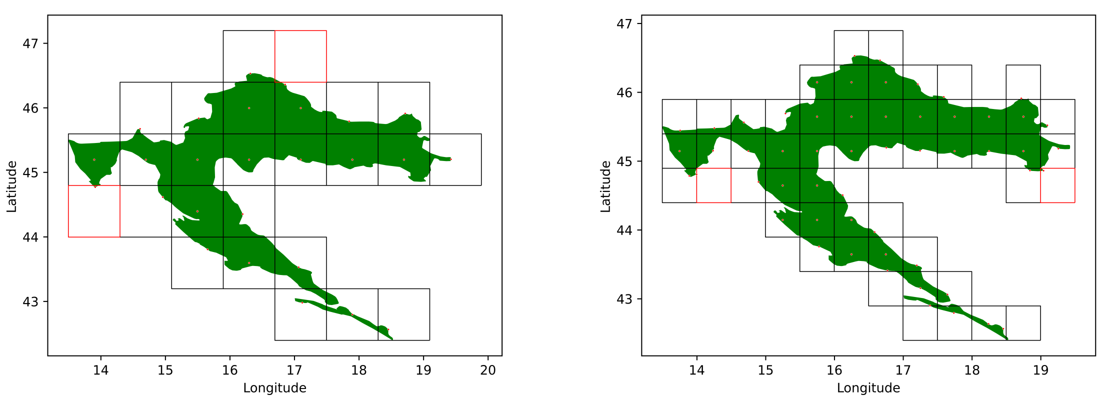

Figure 3: Examples of maps of Croatia divided into distinct regions that represent classes. Red dots represent the centroids of the regions, or otherwise, the point on land closest to the centroid if the centroid is at sea. The image on the right has a denser grid than the image on the left, meaning that it contains more classes.

3.2 Classification Approach

The set of regions of Croatia we mentioned above can be represented in the form of square polygons on a map. Each polygon corresponds to a single class and each polygon also has a centroid that represents the coordinates assigned to the class. The idea is that, instead of predicting the *exact* coordinates of an image, the model classifies the images into regions from the previously described set of regions. Since now we don't have specific coordinates we predict for each set of four images, we instead declare the predicted coordinates to be the centroid of the region where the image was classified and calculate our error in regards to that centroid. Notice that the image's true coordinates might be relatively distant from the centroid of the region into which the image was classified. This error shrinks as the number of specified regions (classes) grows. An image of Croatia divided into square regions can be seen in Figure 4.

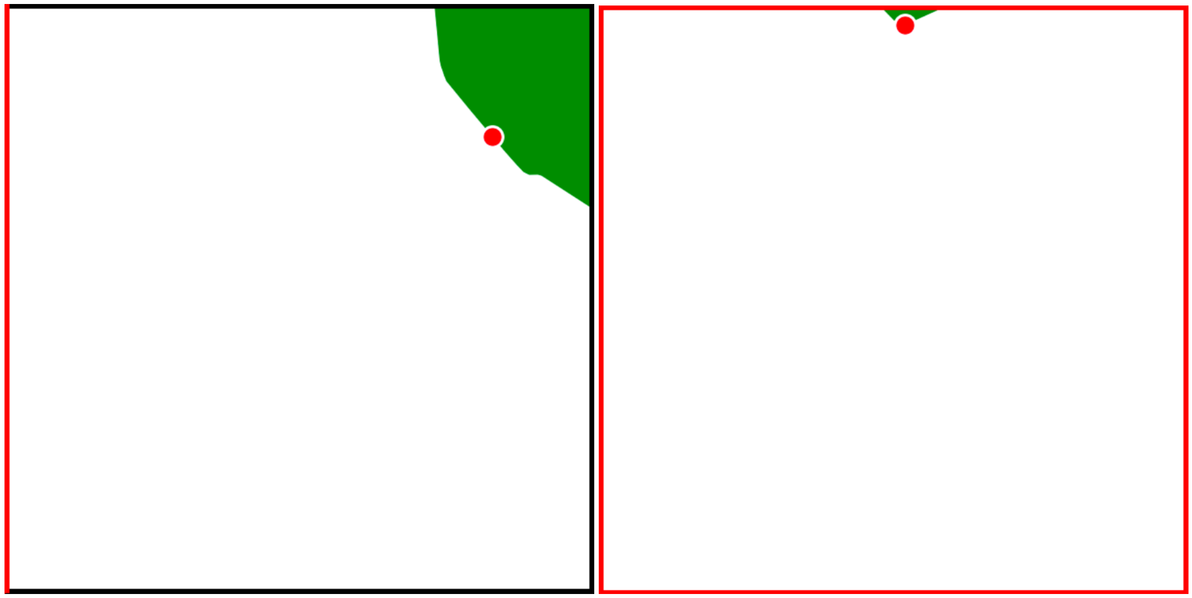

Figure 4: The image on the left depicts a class whose centroid isn't located on land, and has thus been relocated to the closest point on land . The image on the right depicts a class that doesn't contain any images in it due to being mostly made up of sea. This class will be ignored during prediction.

3.2.1 Class Creation

How is this grid-like set of regions created? First, we create a generic grid that is located fully inside the bounds of Croatia. It contains numerous polygons (squares) which are adjacent to each other. Although we are working with a fixed number of polygons, not every polygon is created equal. This is because, unfortunately, some of them aren't located in Croatia at all, as they don't really intersect Croatian territory. Therefore, they shouldn't be taken into consideration further on and are filtered out. After this is done, we proceed to the task of finding the centroids of these polygons. Using the `geopandas` library, this problem can be reduced to a single simple expression: `polygon.centroid`. Great! Now we have a list of classes for our model. But before we continue to the next section, we should double-check what we did so far.

3.2.2 Problems That Arise

Let us observe the following example. Even though the centroids of the polygons were calculated correctly (they're in the center of the squares), some of them decided to go sailing and ended up in the middle of the sea. This doesn't make sense for our prediction, as we know for a fact that the dataset contains images only taken on land. This has to change. Therefore, we introduce *clipped centroids*. Clipped centroids are a modification of regular centroids that fix the previously stated issue by clipping the undesirable centroid to the closest possible point on land. By doing this, we reduce the error rate of the model by moving seaborne centroids closer to the image's true coordinates, which are on land.

We have previously mentioned that it's possible to specify the number of classes we desire before creating the grid and thereby make it more or less dense. By choosing a dense grid, we can essentially simulate something akin to regression. This is because, as the number of classes increases and the size of each class decreases, more polygons and centroid values are available as potential classes. A smaller class means that the theoretical maximum distance between a class' centroid and the true image coordinates is also smaller, and therefore has the potential of decreasing the total prediction error. Note that, at the end of the day, this is what matters, not the accuracy of our class predictions, because we calculate the final error by measuring the distance between an image's coordinates (here the class centroid) and its true coordinates. Even if we classify all images correctly, we will still have a potentially large average haversine distance because we never actually predict the true image coordinates, only the class centroids. If we take this to the extreme, which is an infinite number of classes, we can come quite close to regression, but there is a caveat. In classification models, each class needs a certain number of images to effectively train. If this number is too low, the model simply can't extract enough meaningful information from the images of a class to learn the features of that class.

Another problem arises because of Croatia's unique shape, only matched by that of Chile and The Gambia. For some polygons, the intersection area with Croatia's territory is only a few tiny spots, meaning that the majority of the polygon's area ends up in a neighboring country. If this was anywhere before 1991., this wouldn't be a problem, but it isn't. The usage of clipped centroids we previously defined somewhat alleviates this issue, but another problem arises because there simply might not exist any images in the dataset that are located in that tiny area of the polygon, that is, within Croatia. And in fact, we did end up in such a situation. We solved this by discarding these polygons and pretending like they didn't exist in the list of classes we could classify the images into. Fortunately, this doesn't happen too often as the dataset is fairly large and the image's locations are uniformly distributed.

3.2.3 Loss Function

Lastly, as with most classification approaches, we use cross entropy loss on the predicted and true classes. The true classes are represented with a one-hot encoded vector, while the predicted classes are represented with a vector of probabilities for the likelihood of each class. To obtain the coordinates from these probabilities, we extract the centroid information for each of the classes and multiply it by the predicted probabilities. We add everything app and end up with an average of all the predictions. This way, our final predicted image coordinates are not necessarily within our predicted class, but as our model becomes more sure in its predictions, so do these averaged coordinates come closer to the class. We have noticed that this averaging approach improves performance.

3.3 Regression Approach

This approach is a bit more obvious. Each set of images has its target coordinate, and we task our model with predicting these coordinates. Therefore, the size of the output of the model is not determined by the number of classes, but is simply 2, accounting for the latitude and longitude (or to be precise, the CRS standardized versions of latitude and longitude we explained earlier). We directly compare these output coordinates to the true image coordinates and calculate the error using the haversine formula. Unlike in the classification approach, the loss function we use for the model can also tell us an accurate state of our predictions, as we are not bound by an artificial limit like having classes. That being said, in practice, we noticed that this approach often performs worse than the classification approach and is also slower. It appears that the presence of classes does help the model in training somewhat.

3.3.1 Loss Function

For the loss function, we use mean squared error loss and we predict the coordinates directly, so no transformations are necessary unlike in classification. It is worth noting that these two approaches, classification and regression, can only be compared using the haversine distance metric, as their loss functions work in very different ways and output vastly different values. Therefore, they can't be compared during training, but only during validation and testing, because we calculate the haversine distance value only then.

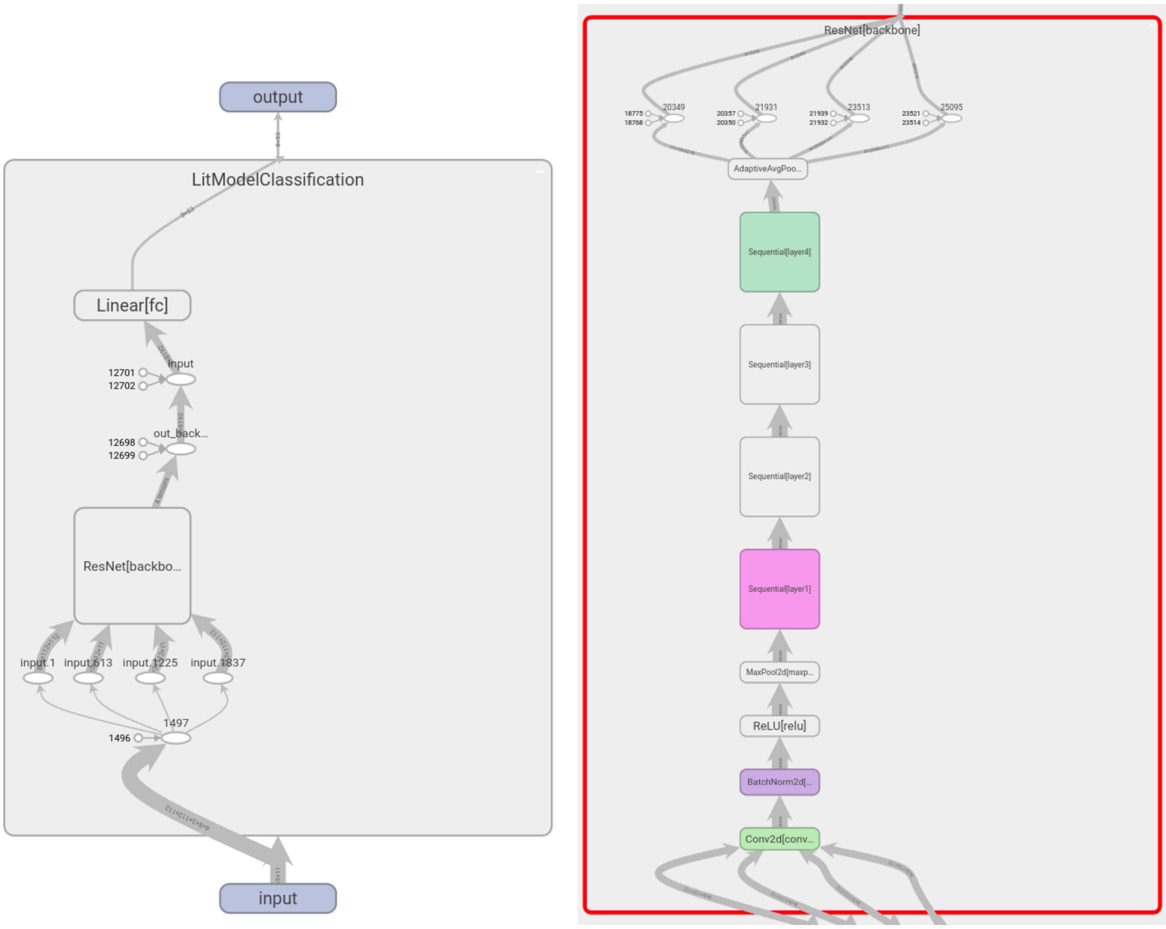

Figure 5: On the image on the left, we can see an overview of the model we used for training. Inputs are fed into the ResNeXt layer with four images being processed in parallel. The output is then fed into a linear layer which can either classify the results into distinct classes or predict coordinates directly as output. On the image on the right, a closeup of the ResNeXt backbone is shown. We can see that it is composed of an early convolution layer, followed by multiple sequential layers who all contain their own convolutions and transformations.

4 Model

4.1 Pretrained networks

Because deep learning progresses at such a breakneck pace, there are numerous approaches that are considered state-of-the-art at the moment, all using vastly different architectures. As readily available high performance models that were pretrained on large datasets are the norm and us being extremely lazy, we chose to use one of these models instead of creating our own from scratch. Originally, we chose the EfficientNet architecture due to it showing both good performance and having lower system requirements when compared to other approaches. However, after doing some experimenting, we ended up using a version of ResNet called ResNeXt instead, as it simply proved more effective.

4.2 ResNeXt

ResNeXt is a highly modular architecture that revolves around repeating powerful building blocks that aggregate sets of transformations. It first processes the images with a simple convolution layer. It then feeds this into a sequential layer composed of multiple bottleneck layers. A bottleneck layer has the function of reducing the data dimensionality, thereby forcing the network to learn the data representations instead of just memorizing them. Each bottleneck layer is again composed of multiple convolution layers for information extraction. After the sequential layer is done, the network is fed into another sequential layer. This process is repeated multiple times. Finally, after all the sequential layers have processed their outputs, the data is fed into a fully connected layer for classification or regression. Of course, all of the mentioned layers also use numerous normalization techniques, such as batch normalization. This process is visualized in Figure 6. Due to this modularity of the layers, ResNeXt includes few hyperparameters that have to be tuned for the architecture to be effective. Aside from the described architecture, we had to do some modifications to the network ourselves in order to make them work with our dataset. There were two modifications we made.

4.3 Modifications

Firstly, we removed the last layer of the network and replaced it with our own classification or regression layer in the form of a simple linear layer that had the appropriate output dimension for our problem. Secondly, because, as we mentioned before, every location contained four images, we modified our network to perform its forward operations for the four images in tandem and then concatenate the outputs before imputing them into the last layer. We did this after doing some research on what was the best way to compute the outputs of separate, but statistically linked images.

5 Training

5.1 Basics

Due to using pretrained models, we first performed what is called fine-tuning. During fine-tuning, we leave the weights of the early layers of a network unchanged and only train the weights of the last few layers. ResNeXt was pretrained on ImageNet, a large image dataset with a diverse assortment of objects. This gives the early layers of ResNeXt 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 ResNeXt parameters. If we were to train ResNeXt 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 other network layers so that they can be trained along with the last few layers. During this phase, by using a sufficiently small learning rate, we induce our dataset information into all the network layers without overwriting the previously trained model.

5.2 Dataset

As previously mentioned, the dataset is divided into two components. The first one is a collection of folders each containing four images taken in the same location. Each folder is named with a unique identifier. The second component is a CSV table where each row contains one of the previously mentioned folder IDs and the corresponding image's latitudes and longitudes expressed in degrees. We divided the data by hand into a training, validation and testing subfolder in a 80%/10%/10% split (for some models, it was 90%/5%/5%). This way, the data could be easily loaded during the training process from the appropriate folder depending on if we are in the training, validation or testing phase. Due to the datasets (and image's) large size, we were unable to load the whole dataset directly into memory (we only had a *measly* 12 GB or RAM available on our GPU). We wanted to do this because it had the potential of greatly decreasing training times, but we were forced to figure out other methods of speeding up training, like using only parts of the dataset during experimentation and employing smarter learning rate schedulers.

Before the training process itself could begin, we needed to first process the data to make it more suitable for training. This entailed modifying the CSV file to include data such as class affiliation, class centroid information and the image's and centroids latitude and longitude information in CRS format. Aside from this, image information such as the minimum and maximum value of each channel had to be also stored for fetching during the training process, and later, inference.

For the competition, we were provided with a dataset of around 64 000 images (16 000 separate locations of four images each), but we added our own separately procured images from Google Street View numbering around 68 000. In other words, we doubled the available data for training. In common deep learning wisdom, having more training data is the best form of regularization, this being our main guideline for gathering additional images.

5.3 The Model Training Phase

5.3.1 Training

The algorithm was composed of 3 steps. In the first step, the algorithm goes through the training dataset in batches of a specified size (usually 8, more if memory constraints allowed). The training step had to be as quick and efficient as possible because time- and processing-wise, it entailed the vast majority of the learning phase. Due to this, the haversine distance of the true and predicted image coordinates wasn't calculated here. This not only let us skip the time consuming process of calculating the haversine distance itself, but also avoided the costly transformations of the data necessary for converting it to a format suitable for input into the haversine distance functions.

5.3.2 Validation

After we exhausted the training dataset (normally called an epoch), we validate our progress on the validation part of the dataset. This is done to prevent overfitting, which we will explain in more detail. As the model learns on the training dataset and if the model is of a sufficient size, i.e. it has more learnable parameters than there are images in the data, it will essentially memorize all the information in the training dataset by heart and learn how to predict it perfectly. However, because it has never seen the data in the validation dataset before, it will predict it correctly only if it has a good generalization capacity, that is to say, it can apply its knowledge on previously unseen data. Aside from this, the validation dataset allows us to finally calculate the haversine distance for the predictions, which is the most accurate (and only true) measure of our current progress, as well as the final performance of the model. It is worth noting that during the validation process, learning is disabled, as the model's trainable parameters are "frozen" here. Another thing to note is that the validation phase also functions as a "checkpoint" for our model. The best performing iterations of the model on the validation dataset are saved for later to be used on the testing and inference datasets.

5.3.3 Testing

Finally, after all the training epochs are over, meaning we went over both the training and validation dataset numerous times, we can test our model on the testing dataset. Just like the validation dataset, the test dataset has never been seen by our model before and is used as a final performance assessment. The best model found in the validation phase is used here. What makes the testing dataset special is that, during validation, we often change and test different values for hyperparameters. This is called hyperparameter tuning. During this process, some of the information from the validation dataset *leaks* into the model, because we are essentially training the hyperparameters on the validation dataset. Due to this, at the very end, another test is performed on a separate, never before seen test dataset to create a truly unbiased performance assessment of our model.

5.4 Hyperparameters

5.4.1 Image and Batch Size

We set the image size to either 224x224, 112x112 or 56x56 pixels, depending on the model. A smaller image size allows for larger batches and faster training, but potentially contains less information in the images due to resolution loss. Batch size itself has the purpose of regulating how fast and accurate our training is. You see, perfect backpropagation (an algorithm which we are sure you know very well and we won't explain for a millionth time) updates model weights only after going through all images in the entire training dataset. This usually results in the most accurate possible gradients for optimization, but has the downside of being very slow and often ending up in local optima. On the other end of the spectrum, training on only one image per batch is very fast and can easily escape local optima, but is also rather inaccurate and converges quite slowly due to imprecise gradients, especially in the later stages of training. The best of both worlds can be achieved by training on multiple images per batch, ideally as many as possible (actually, there is an upper limit to the number of batches one should use, but it is often infeasibly large). Memory and GPU limitations commonly don't allow for this, so we are forced to use the largest batch size we can (8, to be precise, for an image size of 112x112 pixels). Lowering the image resolution can give us more wiggle room in choosing batch sizes, but setting batch size too a too large value runs the risk of quickly filling up the GPU's memory and crashing the training process midway through.

5.4.2 Learning Rate

Aside from image size and batch size, another important parameter to be adjusted is learning rate. A learning rate with a large value can converge the model to a solution quickly, but lacks fine tuning capabilities necessary for that extra edge in performance during later phases of training. On the other hand, a small learning rate can make the training process very slow and can cause it to often end up in local minima (in other words, learning rate has the opposite behavior of batch size). Ideally, we should have a large learning rate in the early stages of training, while periodically reducing it as the training moves on. The application of momentum can also help in this. Momentum is a value that is proportionally added to the learning rate during training if it is going particularly well. An optimizing algorithm that includes all of the aforementioned mechanisms is Adam. We chose it due to its good balance between precision and speed during training, as well as its built in momentum function.

5.5 Inference

Inference is performed after the entire training phase of our model is over. It is the process of testing our best performing model on never before seen images and can be described as the training phase in reverse: instead of seeing an image's coordinates and training our model on them, we now have to look only at the image itself and predict it's coordinates. The true image coordinates are hidden from us and are compared against our answers without us overseeing any part of the process. This is how our model will finally be tested and compared against other models in the end to assess its final performance. More about how this is done can be found in our Technical Documentation (TM).

To do:

Datasets

- split

Connecting datasets (data module):

- hashing, mean, std, calculations

Training - - dataset

External sampling