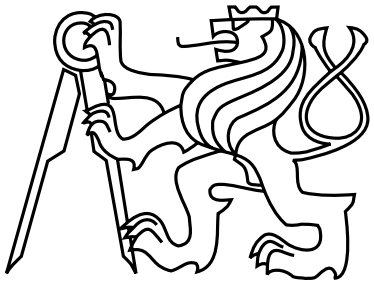
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Masters's Thesis

Visual image search and geolocation

Andrii Zakharchenko

Supervisor: Georgios Tolias, Ph.D.

Study Programme: Open Informatics

Field of Study: Data Science

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zakharchenko** Jméno: **Andrii** Osobní číslo: **453318**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Datové vědy**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vizuální vyhledávání obrazů a geolokaliace

Název diplomové práce anglicky:

Visual image search and geolocation

Pokyny pro vypracování:

Given a large collection of geo-tagged images, the goal is to build an image-based localization system which will predict the location at which outdoor images are taken. The task will be handled as image retrieval where a large collection of images needs to be indexed. Deep learning models will be used for the visual representation of images. Location estimation is relies on a first step of image retrieval and the location estimation uses the top-ranked and their geo-tags to predict the location.
The student is expected to:
1. Collect a new dataset that is appopriate for training and evaluation for the considered task. Datasets used in the literature are not fully available as they rely on re-downloading images from Flickr and many are missing.
2. Implement the method in the Deep IM2GPS (ICCV) paper that relies on retrieval and kernel density estimation.
3. Perfrom an evaluation of the implemented approach on the newly collected dataset.
4. Seek improvements of the location prediction approach. Candidate directions are to design models that rely on the result provided by the ICCV approach and improve by locally refining the result or learn an improved image representation that is better suited for the particular prediction method.

Seznam doporučené literatury:

Vo Jacobs Hays, ICCV 2017, Revisiting IM2GPS in the Deep Learning Era
Radenovic Tolias Chum, PAMI 2019, Fine-tuning CNN Image Retrieval with No Human Annotation

Jméno a pracoviště vedoucí(ho) diplomové práce:

Georgios Tolias, Ph.D., skupina vizuálního rozpoznávání FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.02.2021** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2022**

Georgios Tolias, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

O Captain! my Captain! our fearful trip is done,
The ship has weather'd every rack

Walt Whitman

Aknowledgements

I want to express sincere gratitude to my thesis supervisor Georgios Tolias, Ph.D. for his help and moral support during work on this thesis.

I would also like to thank my family for their continuous support, unconditional love and for giving me the opportunity to pursue my study.

Finally, I also want to take this opportunity to say thank you to my girlfriend, who helped me to get through the most stressful times.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Law no. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague .....

Abstract

Image geolocalization, inferring the geographic location of an image, is a challenging computer vision problem with many potential applications. In this work, we explore the problem of predicting the geolocation of an input image based on a content-based image retrieval approach. We firstly collect a large dataset of geotagged photos that were taken in the Czech Republic. We extract descriptors from the collected photos using a convolutional neural network. Using obtained descriptors we build an image geolocalisation system that finds similar images in the database and predicts the location of input image based on the coordinates of similar images and kernel density estimation. We propose a neural network that is designed to improve obtained descriptors specifically for the task of image geolocalisation with kernel density estimation.

Keywords: content-based image retrieval, image geolocalisation, kernel density estimation, neural network

Abstrakt

Geolokalizace obrazů je náročným problémem počítačového vidění s mnoha potenciálními aplikacemi. V této práci zkoumáme problém predikce geolokace vstupního obrázku na základě přístupu založeného na vizuálním vyhledávaní obrazů. Nejprve sbíráme velký datový soubor geotagovaných fotografií, které byly pořízeny v České republice. Ze shromážděných fotografií extrahujeme deskriptory pomocí konvoluční neuronové sítě. Pomocí získaných deskriptorů budujeme geolokalizační systém obrázků, který najde podobné obrázky v databázi a předpovídá lokace vstupního obrázku na základě souřadnic podobných obrázků a jádrového odhadu hustoty. Konečně navrhujeme neuronovou síť pro vylepšení získaných deskriptorů speciálně pro úlohu geolokalizace obrazu s jádrovým odhadem hustoty.

Klíčová slova: vyhledávání obrázků založené na obsahu, geolokalizace obrázků, jádrový odhad hustoty, neuronová síť

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and goal

This thesis explores the problem of predicting the geolocation of an image based on an image retrieval technique. This means that given an input image that we don't know anything about, except its content, we want to predict the geographical coordinates of a place where this image was taken. Having a large collection of images annotated with GPS coordinates, we can find the most similar images to our input image. Based on those similar images and their GPS coordinates, we can predict the location of the input image.

The problem of inferring coordinates from an input image is a challenging and interesting problem that has various applications, such as determining where the photo was taken for forensic analysis, usage in a search engine, also knowing the coordinates of an image we can easily enrich it with additional metadata such as climate, population density, average income, etc.

The goal of this work is to

1. Collect a large dataset of geotagged images.
2. Build an image localization system that relies on a content-based image retrieval pipeline. The system should predict coordinates at which a given input image was taken.
3. Seek improvements to the geolocalisation system by learning an improved image representation.

We will approach our goal as follows, at first we will use an image hosting platform called Flickr to download geotagged images. We will build the image geolocalisation system based on kernel density estimation and other methods. To be able to compare photos, we will obtain image representations using a convolutional neural network by Radenovic *et al.* For the final task, we will propose an artificial neural network that we will train to improve image representations for the task of geolocalisation prediction with kernel density estimation.

1.2 Thesis structure

The rest of this thesis is structured in the following way:

- In chapter 2 we describe the background theory for this work, in particular, we describe the content-based image retrieval and its application for image geolocation with an overview of existing work in these fields.
- In chapter 3 we describe our data collection process and analyze collected data.
- In chapter 4 we will describe in detail the models we use for image geolocation and show how we performed parameter selection for these methods, and explain how different parameters influence performance. Lastly, we will evaluate models on the test set.
- In chapter 5 we will propose an approach based on a neural network to improve the results of models described in chapter 4.
- Finally, in chapter 6 we will summarize this work.

2 Theoretical background and related work

Since the concept of content-based image retrieval is crucial to the thesis in this chapter we will first provide a theoretical overview of the method and after that, we will describe how it can be used for the task of image geolocation.

2.1 Content-based image retrieval

The goal of content-based image retrieval (CBIR) is to find similar images in the reference database given some input image (also referred to as *query image*) based on the content of images. CBIR approach is opposite to text-based image retrieval, which relies on the textual description of images or some other metadata.

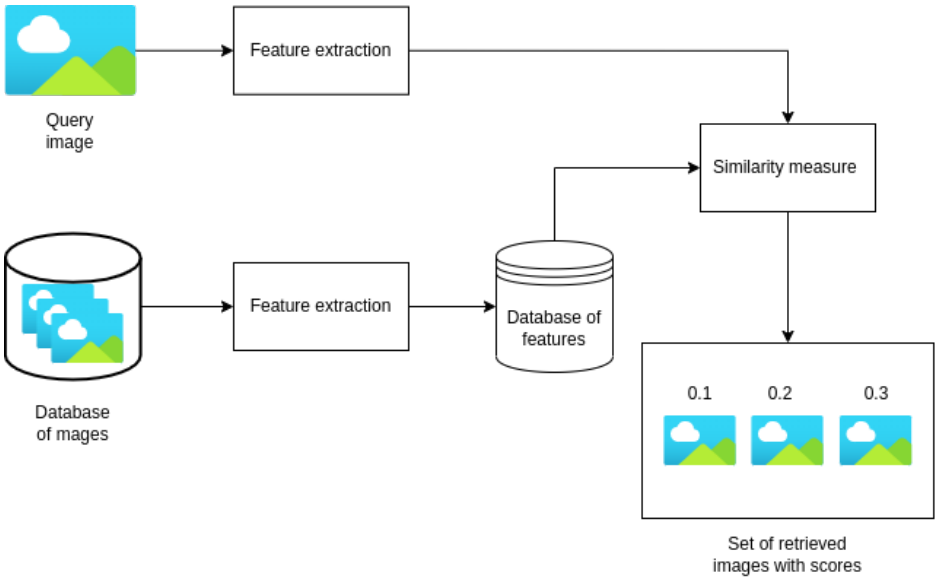


Figure 2.1: Content-based image retrieval system

As described in [7], classical content-based image retrieval system is shown in the figure 2.1. Using raw images in CBIR is impractical mostly due to the high dimensionality of the data, and also due to the fact that a lot of redundant information is embedded into an image. So the first thing that we see is that database images and query images are subjected to *feature extraction*. The purpose of feature extraction is to create a compact

image representation that includes the most relevant information. The output of feature extraction is called a *feature vector* or a *descriptor*. As we can see feature vectors of database images are stored in the database, which implies that usually, descriptors from database images are pre-computed in advance. For the query image, feature extraction is usually performed upon the query's submission. In the last stage, the descriptor of a query image is compared to the descriptors from the database to determine the similarity scores between them. Then usually top N most similar images are retrieved from the image database. Normally, feature databases can contain a large number of entries, so comparing a query descriptor with all descriptors in the database in a brute-force manner can be inefficient. In order to increase the speed of image retrieval database feature vectors are usually indexed using data structures like R-trees, K-d trees, SS-trees, quad-trees, and many more [7].

Feature extraction is a core concept in content-based image retrieval. In general, we can divide features into two main categories, local and global features. Global features are extracted from the visual content of the entire image. Some examples of global features are global color histogram [1] and Histograms of oriented gradients (HoG) [5]. Local descriptors on the other are extracted from regions of interest or objects in the image. Examples of local descriptors are Scale Invariant Feature Transform (SIFT)[3], Speed Up Robust Features (SURF)[6], Local Binary Patterns (LBP)[2], and many more. Such features are usually aggregated using methods such as bag-of-visual-words [4], VLAD [9] and Fisher Vectors [10].

Nowadays, convolutional neural networks (CNN) are used as feature extractors. Some previous methods used activations of CNNs trained on ImageNet dataset for the classification task as features for image retrieval [16], [13] [12]. Even though those descriptors from CNN showed good results, the networks were not trained specifically for the end task. In this work [11] authors re-trained such networks on the dataset that is closer to the end task, however, they were still training the network for the classification task. In [15] authors proposed an end-to-end convolutional neural network that was trained specifically for visual place recognition using a *metric learning* approach. They used a collection of geotagged images which offered a form of weak supervision for training. The metric learning approach uses two-branch Siamese or triplet networks and they use matching and non-matching pairs of images. In this approach, image embedding is learned so that Euclidean distance captures the similarity. However, the problem of data annotation for this approach is more pronounced, for the classification only a class label for an image is required, but for metric learning, labels are required per image pair. In [19] Radenovic *et al.* train CNN for image retrieval with dataset created from structure-from-motion pipeline to automatically annotated image pairs.

2.2 Geolocation based on CBIR

Content-based image retrieval can be used for predicting the geolocation of a query image. In [8] Hays and Efros collected a large database of images labeled with GPS coordinates. They created image representation using classical non-deep-learning feature extractors and for a query image, they predict its geolocation as coordinates of the most similar image in the database. It is clear that in this case the problem was formulated as content-based image retrieval. In that same paper, authors were also searching for k-nearest-neighbors in the database, using mean-shift clustering to predict coordinates of a query image. In

[17] Weyand *et al.* formulated geolocation task as a classification problem. They divided the surface of the Earth into multiple smaller regions where each region represents a target class. For an image from their dataset of geo-tagged images they assigned a class based on whether image belongs to a region that the class represents. And finally they trained a CNN model with classification loss. In [20] Vo *et al.* train a similar network as in [17], however they later use it as a feature extractor to perform image retrieval in order to find similar images in their database, and similarly to [8] they predict coordinates of a query image using kernel density estimation method.

3 Dataset overview

In this chapter we will describe our data collection process, and we will provide the analysis of the collected data. We will also describe how we split our data into datasets for image retrieval, training, testing and validation.

3.1 Data collection and analysis of collected data

For the task of image geolocation we need photos that are labeled with GPS coordinates that correspond to a place where the photo was taken. Papers such as Im2gps [8], PlaNet [17] and Revisiting im2gps [20] focus on datasets that include photos from all over the world, however datasets from these papers are not fully available as of today. For this work we decided to concentrate only on those photos that were taken in the Czech Republic, due to limited time and computational resources.

As a source of images, we choose Flickr. Flickr is an image hosting platform that hosts more than 10 billion photos and has more than 100 million registered users. More importantly for us, Flick extracts GPS coordinates (if available) from photo metadata and stores them along with a photo. Also, Flickr provides a public API (Application Programming Interface) which we can use to search and download photos. Unfortunately, this API doesn't offer a convenient way to search images based on a country. On the other hand, the API allows to search photos based on tags that users add manually to their photos. So we overcame this limitation by searching only for photos that are tagged with "czechrepublic" tag. Additionally, we search for photos that have GPS coordinates and were uploaded to Flickr between January 1, 2005 and November 11, 2020. Using this approach we were able to collect **230883** photos that were taken in the Czech Republic. In the figure A.1 from the appendix you can see several examples of photos from the collected dataset.

When we download a photo from Flickr we also store its metadata in MongoDB, a no-SQL database that is optimized for storing documents in JSON format. The metadata includes photo id, owner name and id (owner is a Flickr user who uploaded a photo), tags, GPS coordinates and URLs which we can use to download the photo. Storing metadata in MongoDB brings several advantages. Firstly, we can retrieve the metadata of an image easily and faster. But most importantly, MongoDB has built-in features for working with geospatial data. Using this MongoDB built-in functionality, we can easily perform computations on GPS coordinates which helps, for example, determine nearest neighbors of an image based on GPS, compute the geographical distance between two photos, get all photos within some radius from a query image, and etc.

Using this built-in functionality of MongoDB we can gain more information about the collected dataset. One thing we would like to know is how collected images are distributed across the Czech Republic. For this, we will use coordinates of administrative boundaries publicly available at the Czech Office for Surveying, Mapping and Cadastre website [22] and, for each image in the dataset, we will determine in which administrative region of the Czech Republic it was taken based on the available GPS coordinates. The distribution of photos by the regions is shown in the figure 3.1.

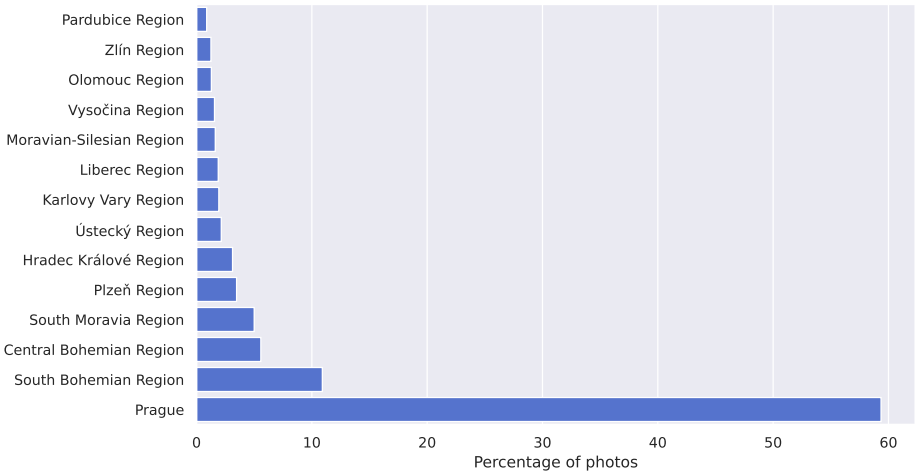


Figure 3.1: Distribution of photos by regions of Czech Republic

As we can see, almost 60% of images were taken in Prague. This can be explained by the fact that Prague is the capital and the biggest city of the Czech Republic as well as by the fact that Prague is the main tourist attraction in the Czech Republic and we collect images from Flickr which is an international platform. So we can expect that majority of photos from the Czech Republic that are hosted on Flickr are made by tourists.

Given that majority of photos in our dataset are from Prague, let's also have a look at the distribution of photos by Prague districts.

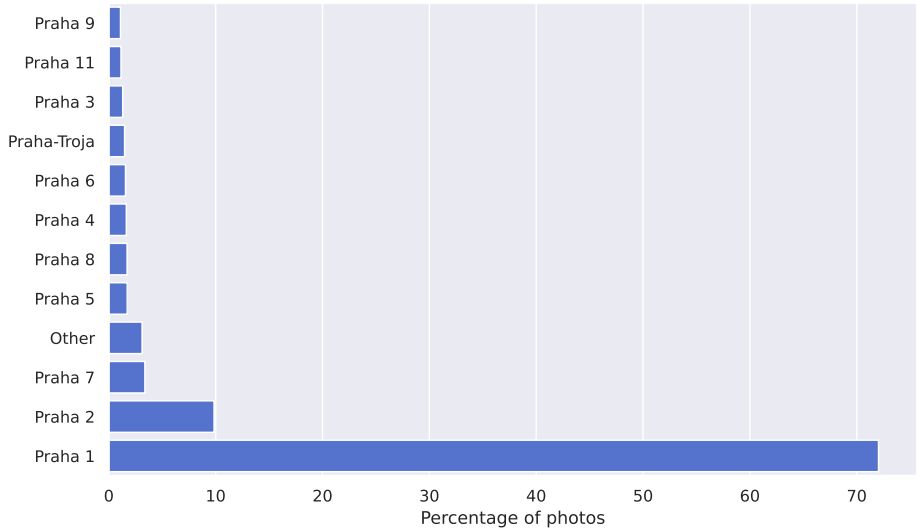


Figure 3.2: Distribution of photos by Prague districts

In the figure 3.2 X-axis represents the percentage of all photos from Prague, and the *Other* category on the Y-axis represents the combination of all Prague districts that separately have less than 1% of photos from Prague. From this distribution, we can see that majority of photos come from Prague 1 and Prague 2 city districts, which are the main touristy areas in the city.

3.2 Dataset splits

For the purpose of validation, testing and later training we split the collected dataset into four sets: database, validation queries, testing queries and training queries. The database set represents the biggest part and it's a set where we will be looking for similar images to the query images. Validation and testing query sets will serve as datasets of query images and as names imply we will use them for validation and testing. Finally, the training queries set, will serve as a dataset of query images on which we will try to learn a model to improve the results of localization later in this work. For the purpose of geolocation prediction we treat coordinates of images in the database set as known and for the queries datasets we treat locations as unknown.

To split collected data in the datasets mentioned above we employ the following strategy. We devote 95% of images to the database set, 3% of images to the training queries set and 1% to validation and testing queries sets each. However, when we split the data we also want to make sure that all photos from the same author belong only to one of the datasets. In other words, we don't want to be in a situation when one photo of an author is in the database set and another photo of the same author is in testing queries dataset. We do this since some authors can have multiple photos from the same location or an author can have a unique style of his/her photos and this can falsely boost localisation results.

In the end we end up with the following number of photos per dataset:

Dataset	Number of photos
Database set	219342
Training queries set	6936
Testing queries set	2314
Validation queries set	2291

Table 3.1: Number of photos per dataset

Lastly for this chapter let's have a look at the distribution of images by regions per dataset, similarly as we did above.

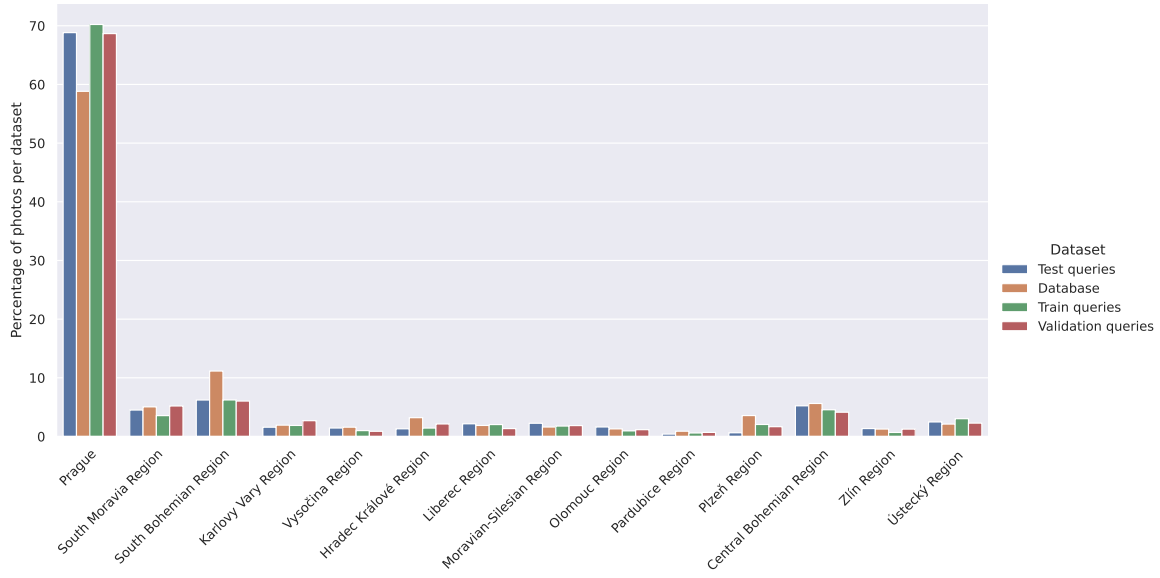


Figure 3.3: Distribution of photos by region per dataset

In the figure 3.3 Y-axis represents a percentage of photos that were taken in the region on the X-axis for each dataset. For example, we can see that 60% of images from the database set are from Prague, while around 70% from the train queries set were taken in Prague, the same goes for test queries and validation queries. So we can actually see that photos from Prague are over-represented in queries datasets compared to the database set. This happened due to the fact that when we split data we also want to keep photos from the same author in one dataset. This is not ideal, but it is something that is hard to control with this splitting strategy.

4 Geolocation with Kernel density estimation and other methods

In this chapter, we will describe how we perform image geolocation. Since we rely on content-based image retrieval to perform image geolocation, we will first explain what kind of image representation we use to perform a lookup in our database. Then we will describe in detail what methods we use for image geolocation and how we measure the performance of our models. Then we will describe how we selected parameters for individual methods and how these parameters influence the performance of a model. Lastly, we will pick the best models and compare them on the testing set.

4.1 Image representation for image retrieval

In the chapter 2, we described a problem of image retrieval. We saw that we need some compact image representation that we can use to measure similarity between a query image and images from our database. For this work, we have chosen to obtain image representations (also referred to as *descriptors* in this work) using a convolutional neural network proposed by Radenovic, Tolias, Chum in [19]. This is a state-of-the-art neural network that was designed to produce compact image representation specifically for the task of content-based image retrieval and was trained on a dataset of photos that depict some landmarks. We feed the images that we collected for this work to the network, and as output, we get a descriptor vector $\mathbf{D} \in \mathbb{R}^{2048}$ for each photo.

For two different vectors $\mathbf{p}$ and $\mathbf{q}$ we can compute a distance between them, and it should hold that for photos that are similar, the distance between their respective descriptors should be minimal. We can use either Euclidean distance

$$d_e(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^{2048} (p_i - q_i)^2} \tag{4.1}$$

or we can also use cosine distance

$$d_c(\mathbf{p}, \mathbf{q}) = \frac{\mathbf{p} \cdot \mathbf{q}}{||\mathbf{p}||||\mathbf{q}||} \tag{4.2}$$

Euclidean distance can be any non-negative real value, and the closer the distance between two descriptor vectors to zero, the more similar are the original images. Cosine distance can obtain values from $[-1, 1]$ interval, and the closer the distance between two descriptor

vectors to 1, the more similar are the original images. Note that the descriptors that we obtain from the neural network are normalized, and for the unit vectors Euclidean distance and cosine distance are proportional. However, we will still evaluate our models on both distances to see if there is some advantage of using one over another and because we will later define similarity metric differently for Euclidean and cosine distances.

Finally, we will need to find the nearest neighbors of a query image among images from our database. The naive approach would be to compute a distance between a query image descriptor and every other descriptor in our database set. However, as we stated in chapter 3 our database set contains 219342 images, so to compute the distance for every image from the database set would be computationally expensive. Another approach is to build some index (for example, using KD trees), to perform a nearest-neighbors search in high dimensional space. However, making such an index from scratch can take a lot of time, so we decided to use Faiss library [18] developed by Facebook AI Research. Faiss provides methods for efficient nearest-neighbor search in high dimensional space, GPU utilization to speed up computations, and Python interface, which helps easily integrate Faiss into our codebase.

4.2 Description of geolocation methods

To utilize image retrieval for geolocation, we can use several different methods. The first one is a simple *one-nearest-neighbor*. The idea is that we will use an image retrieval pipeline to find the closest image to a query in the descriptor space and use its coordinates as a prediction for the coordinates of a query. The disadvantage of this method is that we only use information from one neighbor, and it's not guaranteed that the closest image in the descriptor space will be the closest image in the coordinate space.

Another approach that we will try is to find several nearest-neighbors in descriptor space and compute the coordinates of a query image as an average of coordinates of those nearest neighbors. We will use the simple average and the weighted average. Let's say we have k nearest neighbours, their coordinates is two-tuple of longitude and latitude (ϕ_i, λ_i), where $i \in \{1, 2, ..., k\}$, for the simple average we can compute the coordinates of a query (ϕ, λ) as

$$(\phi, \lambda) = \left(\frac{1}{k} \sum_{i=1}^{k} \phi_i, \frac{1}{k} \sum_{i=1}^{k} \lambda_i \right) \tag{4.3}$$

For the weighted average, we can compute the coordinates of a query like this

$$(\phi, \lambda) = \left(\frac{\sum_{i=1}^{k} w_i \phi_i}{\sum_{i=1}^{k} w_i}, \frac{\sum_{i=1}^{k} w_i \lambda_i}{\sum_{i=1}^{k} w_i} \right) \tag{4.4}$$

The weight w_i, in this case, is the *similarity* of a neighbor i to the query. How we compute similarity depends on a distance metric that we use. For the Euclidean distance d_e from the equation 4.1 similarity $s_e = w$ between query $\mathbf{q}$ and some neighbour $\mathbf{p}$ is computed as

$$s_e(\mathbf{p}, \mathbf{q}) = \left(\frac{1}{d_e(\mathbf{q}, \mathbf{p})} \right)^m \tag{4.5}$$

where m is a tunable parameter added to introduce non-linearity.

For the case when we use cosine distance d_c from the equation 4.2 we compute similarity $s_c = w$ as follows

$$s_c(\mathbf{p}, \mathbf{q}) = (d_c(\mathbf{q}, \mathbf{p}) + 1)^m \tag{4.6}$$

As we stated previously, $d_c \in [-1, 1]$, so we add one to the distance to avoid negative values in similarity. The term m also introduces non-linearity, so the more two image descriptors are similar, the more they contribute to similarity.

Finally, we will use *kernel density estimation* (KDE) as described in [20] to estimate a distribution of coordinates of nearest-neighbors, and we will pick coordinates with the highest density. Let's formulate kernel density estimation with Gaussian kernel as follows

$$\hat{f}(\mathbf{x}) = \frac{1}{(\sum_{i=1}^{k} w_i)|\Sigma|^{\frac{1}{2}}} \sum_{i=1}^{k} w_i \mathcal{N}(\mathbf{x}; \mathbf{x}_i, \Sigma) \tag{4.7}$$

This means that we will estimate a density function $\hat{f}$ in the point $\mathbf{x} = (\phi, \lambda)$, for this we will use k nearest-neighbors, which have GPS coordinates $\mathbf{x}_i$ and w_i is a weight, which corresponds to the similarity between i-th neighbor and a query. We compute similarity in the same way as described in the equations 4.5 and 4.6 depending on which distance metric we will use. Notation $\mathcal{N}(\mathbf{x}; \mathbf{x}_i, \Sigma)$ represents a value of Gaussian (Normal) density function in point $\mathbf{x}$ with mean $\mathbf{x}_i = (\phi_i, \lambda_i)$ and covariance matrix $\Sigma = \sigma^2 I$, where σ is tunable parameter. Note that we talk about multivariate normal distribution since we estimate a distribution of coordinates in GPS space where each point is two-tuple of longitude and latitude. We also divide by $\sum_{i=1}^{k} w_i$ to ensure that $\int \hat{f}(x)dx = 1$.

After we estimated a density function $\hat{f}$ we choose optimal coordinate $\mathbf{x}^*$ for a query as follows

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}}\, \hat{f}(\mathbf{x}) \tag{4.8}$$

Another important detail to describe is how we sample points in GPS space for kernel density estimation. In theory, point $\mathbf{x}$ in equations 4.7 and 4.8 comes from a space of all GPS points in the Czech Republic. However, it's computationally infeasible to generate all those points and compute KDE. Instead, for each nearest-neighbor, we construct a grid around it with a total number of points equal to $(2bn + 1)^2$. We also construct a grid based on a σ parameter we use for KDE.

There are 3 important parameters that determine how we generate a grid: b, n and σ. To understand how we generate the grid, let's have a look at figure 4.1 below. A cross in the middle is a point in GPS space with coordinates (ϕ, λ), representing some nearest-neighbor of a query. Parameter σ determines how far away from the neighbor generated points will be in GPS space. Parameter b determines how many σ-intervals we will use. In the figure σ-intervals are depicted by bigger dashed lines. We can also see that the half-width and half-height of a grid in the image equals 3σ, which means that in this example $b = 3$. This implies that coordinates of a top-right point in the grid will be $(\phi + 3\sigma, \lambda + 3\sigma)$ and coordinates of bottom-left point in the grid will be respectively $(\phi - 3\sigma, \lambda - 3\sigma)$. Parameter n determines how many points we will generate per one σ-interval, this corresponds to smaller dotted lines in the image. So the height and the width of a grid, **in terms of a number of points**, is

$2 * b * n + 1$ since we use b σ-intervals and generate n points, and since we want to compute the whole width and height, we must multiply it by two. We also include coordinates of the original point in the grid, and therefore, we add one to the number of points.

In the end, we get a space of $k(2bn+1)^2$ points on which we run kernel density estimation. In this work we use $b = 3$ and $n = 3$.

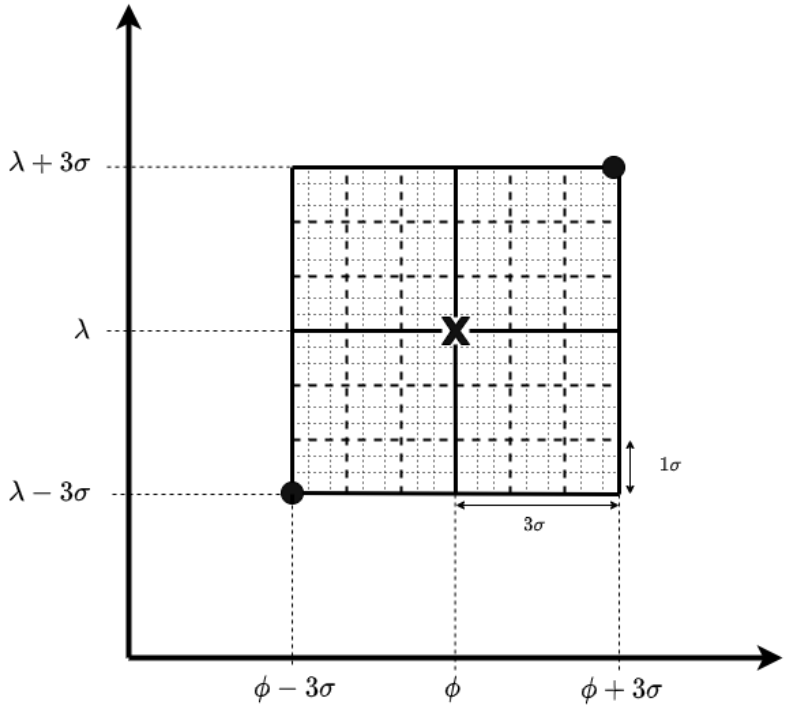


Figure 4.1: Illustration of sampling points in GPS space

4.3 Accuracy metric

To determine how well our models perform, we use the accuracy metric, which is the proportion of correctly predicted locations of queries to the total number of queries.

$$\text{Accuracy} = \frac{\text{number of correctly predicted locations}}{\text{total number of queries}} \tag{4.9}$$

We say that location is predicted correctly if the geographical distance between the predicted location and true location is within some threshold. Throughout this work, we will determine accuracy for several thresholds, namely, 10 meters, 100 meters, 500 meters, 1 kilometer, and 5 kilometers. For example, for the threshold of 10 meters, we say that prediction is correct if the distance between the predicted location and true location is equal or less than 10 meters, and similarly for other thresholds. For two points in GPS space $\mathbf{a} = (\phi_1, \lambda_1)$ and $\mathbf{b} = (\phi_2, \lambda_2)$, where ϕ_i and λ_i are latitude and longitude respectively, we compute a distance between them using Haversine distance

$$d_h(\mathbf{a}, \mathbf{b}) = 2r \arcsin \sqrt{hav(\phi_2 - \phi_1) + hav(\lambda_2 - \lambda_1)[1 - hav(\phi_1 - \phi_2) - hav(\phi_1 + \phi_2)]} \tag{4.10}$$

where r is the radius of the Earth and function hav is

$$hav(\theta) = \sin^2 \frac{\theta}{2} = \frac{1 - \cos\theta}{2} \tag{4.11}$$

4.4 Parameter selection

Some methods that we described in the previous section have parameters that we need to pick correctly to ensure that they produce the best results. This process is called parameter selection. Geolocation based on averaging requires to pick number k for the number of nearest-neighbors, while for the weighted average, we also need to pick k as well as m, which is the exponential term in equations 4.5 and 4.6. For the geolocation method based on kernel density estimation, we need to select k, m and σ. Since the 1-nearest-neighbor method doesn't have any parameters we won't touch it in this section. To find the best parameters, we use the grid search approach, which means that for each parameter k, m and σ, we define a set of values, and for each method, we will compute all possible combinations of relevant parameters and we will run a model on each combination. We will perform parameter selection on the validation set and measure accuracy on different thresholds, as described in the section 4.3 above.

4.4.1 Parameter selection for the method based on averaging

For this method, we only need to select one parameter, which is k. We will try the following values: $\{2, 3, 4, 5, 8, 10, 20, 40, 70, 90, 100\}$ and we will perform parameter selection using both cosine and Euclidean distances.

In the figure 4.2 below we can see accuracy plotted for each parameter k for different distance thresholds. It is apparent from the graphs that with increasing number of nearest-neighbors accuracy of the method decreases. This is most probably given by the fact that averaging is susceptible to outliers.

Also, we run experiments both for Euclidean distance and cosine distance, but the results are nearly identical. This is most certainly because, as we stated previously, Euclidean and cosine distances are proportional, and we don't use similarity in this case, therefore we obtain the same ranking. Also, we don't include a graph for parameter selection with cosine distance in this chapter to reduce clutter, but it's included in the appendix (see figure A.2).

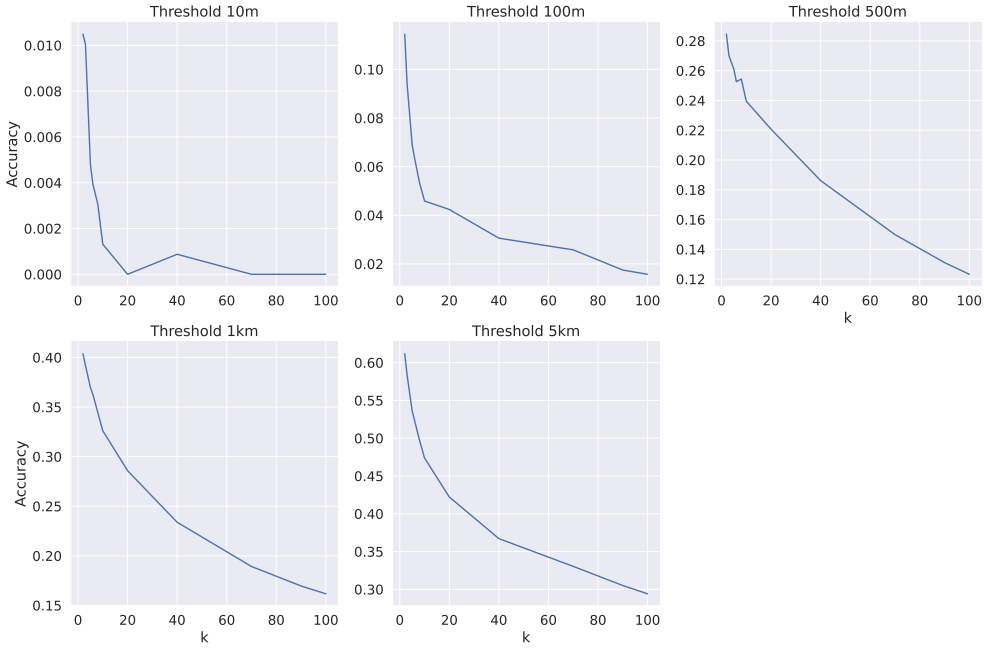


Figure 4.2: Accuracy of averaging method for different parameters and different thresholds

In the table 4.1 below we provide the best results and corresponding parameters for different thresholds. As we can see, the best results are obtained for $k = 2$ on different thresholds.

Threshold	k	Accuracy
10m	2	0.0104
100m	2	0.1145
500m	2	0.2845
1km	2	0.4038
5km	2	0.6118

Table 4.1: Best parameters for the method based on averaging

4.4.2 Parameter selection for method based on weighted averaging

For this method we need to select k and m, firstly let's define a set of values for each parameter:

1. $k \in \{2, 3, 5, 6, 8, 10, 20, 40, 70, 90, 100\}$
2. $m \in \{1, 2, 3, 4, 5, 6, 7, 8, 10, 15\}$

We will run parameter selection on all combinations of the values above, which amounts to a total of 110 combinations.

The figure 4.3 below shows the results of parameter selection with Euclidean distance. We can see that the accuracy, similarly to the averaging method, also decreases with an increasing number of neighbors. However, increasing m reduces the loss in accuracy.

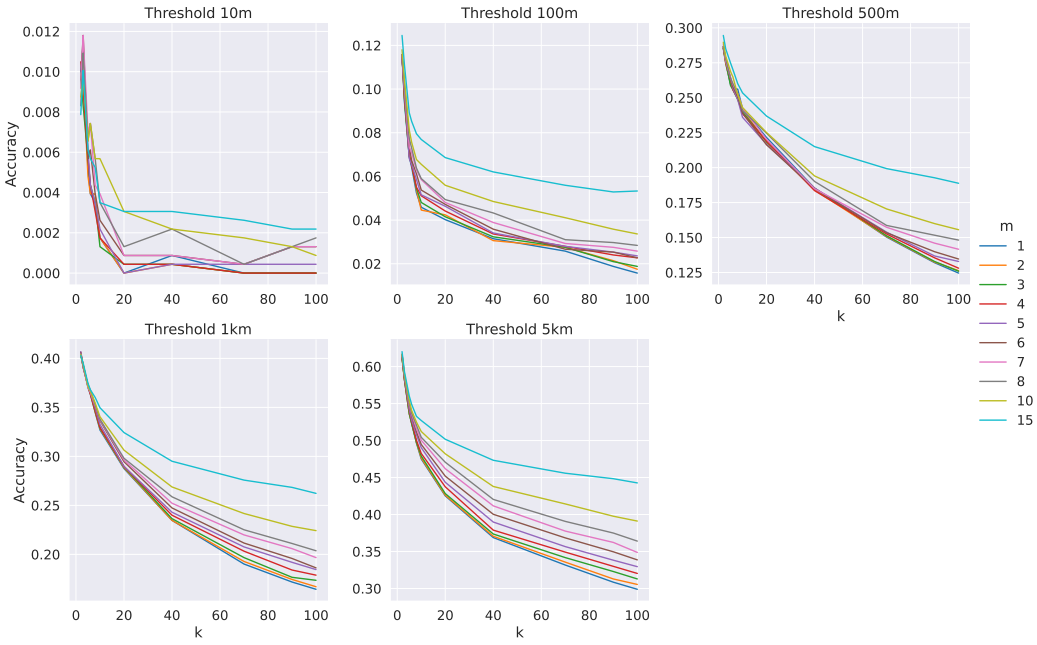


Figure 4.3: Parameter selection for weighted averaging with Euclidean distance

The results for running parameter selection with cosine distance are shown in figure A.3. The behavior is similar, accuracy decreases with the number of neighbors, however, the effect of the parameter m is less pronounced.

Table 4.2 shows best results and corresponding parameters for different distance metrics. We can see that using Euclidean distance is slightly better than cosine distance.

	Euclidean distance			Cosine Distance		
Threshold	k	m	Accuracy	k	m	Accuracy
10m	3	7	0.0118	2	2	0.0104
100m	2	15	0.1245	2	5	0.1171
500m	2	15	0.2945	2	15	0.2875
1km	2	5	0.4069	2	4	0.4055
5km	2	15	0.6201	2	15	0.6145

Table 4.2: Best parameters for the method based on weighted averaging for Euclidean and cosine distances

4.4.3 Parameter selection for KDE method

For this method we will need to select 3 parameters, k, m and σ. Let's define a set of values for these parameters:

1. $k \in \{10, 20, 30, 40, 50, 60, 80, 100\}$
2. $\sigma : \{0.0001, 0.00025, 0.0005, 0.001, 0.0025, 0.005, 0.0075, 0.01\}$
3. $m \in \{2, 5, 8, 10\}$

The total number of all combinations for these parameters is 256.

The figure 4.4 shows the results of parameter selection for the KDE method with Euclidean distance. Since we are selecting 3 parameters, the visualization is a bit more complicated. Each row in the graph represents different distance thresholds, each column represents different values of m, and lines of different colors represent different values of σ. On the x-axis, we have parameter k, and on the y-axis, we have accuracy.

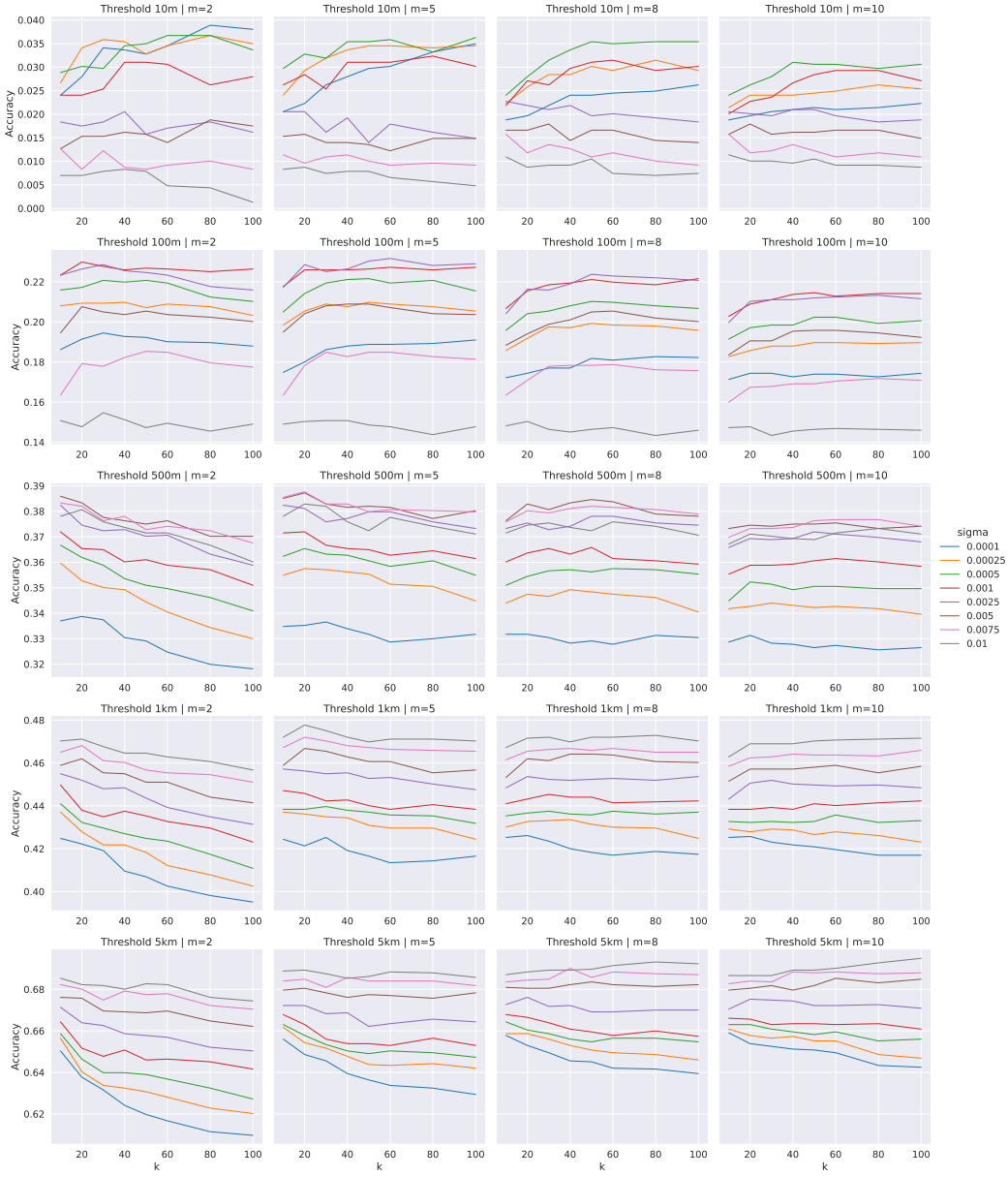


Figure 4.4: Parameter selection for KDE with Euclidean distance

From the graph above it's harder to understand how different parameters influence the performance of the KDE model. Let's try to examine the influence of each parameter individually.

The figure 4.5 below shows values of σ parameter on the x-axis and the average accuracy for the respective value of σ on the y-axis. We have 8 values for parameter k and 4 values for parameter m, so there are 32 combinations of k and m for each value of σ, and we compute average accuracy for each σ among those 32 runs. The vertical lines in the plots represent the standard deviation of accuracy, which helps to understand the variability of accuracy for a given value of σ.

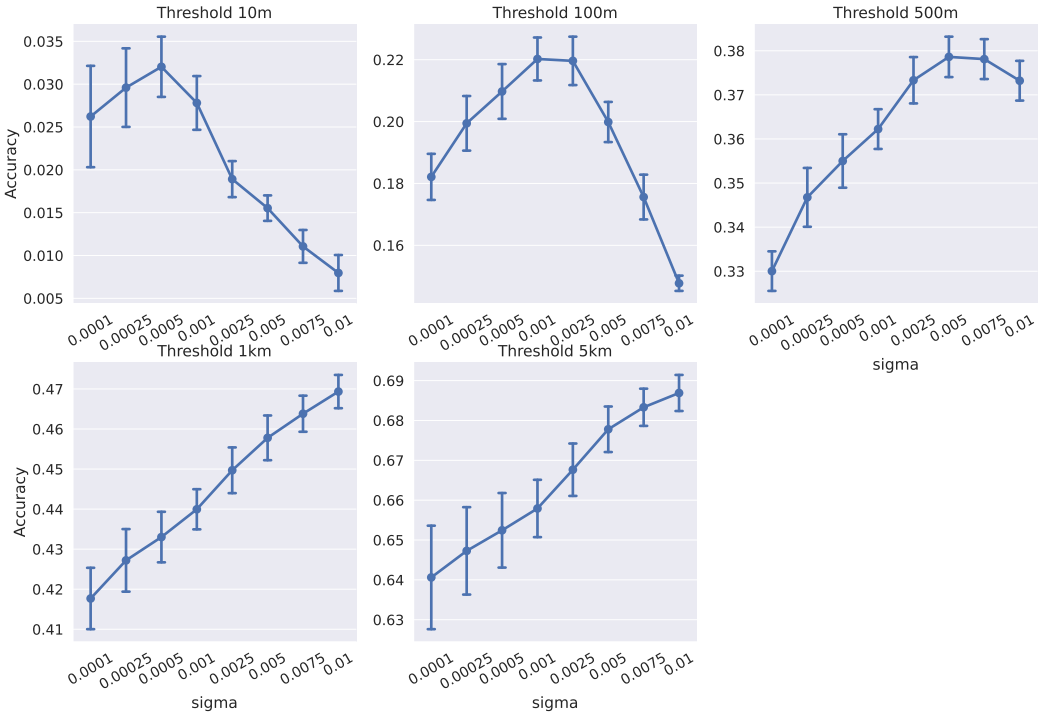


Figure 4.5: Average accuracy for different values of σ. KDE with Euclidean distance.

In the figure 4.5 we can see that value of σ has a strong influence on the accuracy of the model. We can also see that different values of σ perform better for different distance thresholds. And it makes sense if you think about σ as the standard deviation of a normal distribution (we use multivariate normal distribution in the KDE, but for the sake of example, let's think about a univariate case). Let's assume that we build a univariate normal distribution where a random variable is either a longitude or latitude coordinate. The points close to the mean will have a higher value of the density function, and the points that are 3σ away from the mean will have a very small value of the density function. Also note, that longitude and latitude are measured in degrees, and the change of 0.001° in a coordinate is approximately equal to the change of 100 meters in distance. In the KDE method, we construct a normal distribution for each nearest-neighbor of a query and then sum them up. So when we set $\sigma = 0.001$, this means that only points that are approximately within 300 meters (or within 3σ) from a nearest-neighbor will contribute to the final density function. That's why we see that $\sigma = 0.001$ or $\sigma = 0.0025$ perform better on the 100 meter threshold, and $\sigma = 0.005$ or $\sigma = 0.0075$ perform better on the 500 meter threshold.

Let's also see how the other two parameters influence the accuracy of the KDE method. The figure 4.6 is similar to the figure 4.5, only it shows the relation between average accuracy on the y-axis and parameter k on the x-axis. Here we can see that for distance thresholds of 500m, 1km and 5km using more neighbors reduces the performance, and for those levels, the optimal value of k is probably around 20. And for the threshold of 10m and 100m, the optimal value of nearest neighbors is around 40 to 60. There is also a lot of variability in accuracy, probably because parameter σ has the most influence on the accuracy.

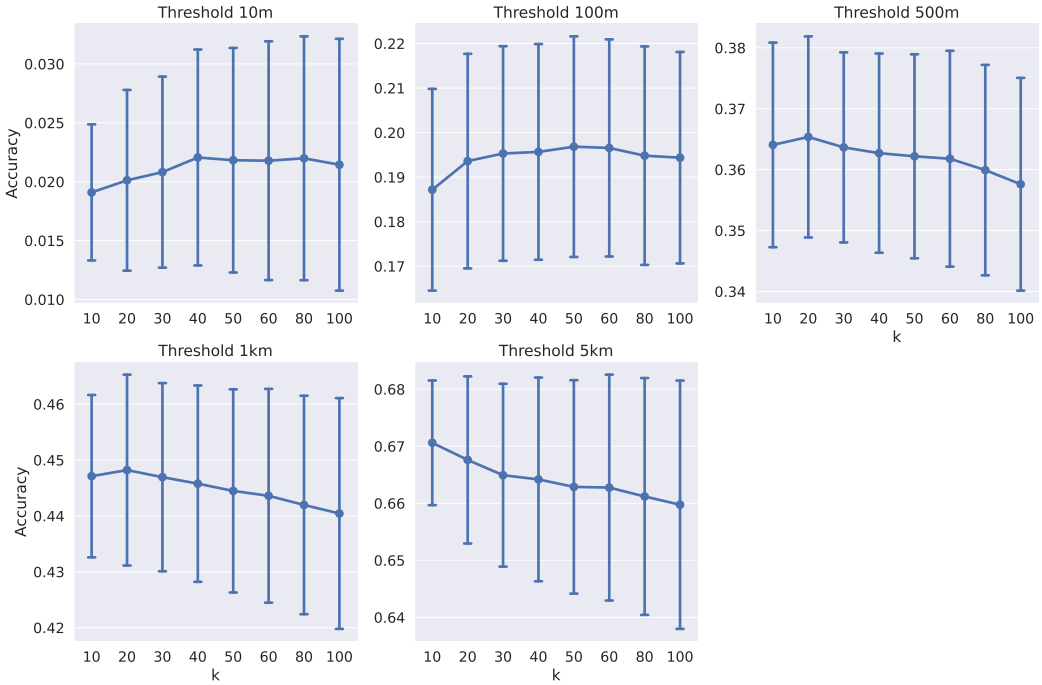


Figure 4.6: Average accuracy for different values of k. KDE with Euclidean distance.

The figure 4.7 shows the relation between average accuracy and parameter m. Here we can see that for the thresholds of 100m, 500m and 1km, the best value of m is around 5, and the performance starts to decrease with an increase in m. For the 10m threshold preferred value of m seems to be 2. And for the threshold of 500m, accuracy increases with increasing m, and the best value appears to be 10. There is also a lot of variability in the accuracy, which is probably again caused by the fact that σ is more important to the accuracy on a certain distance threshold.

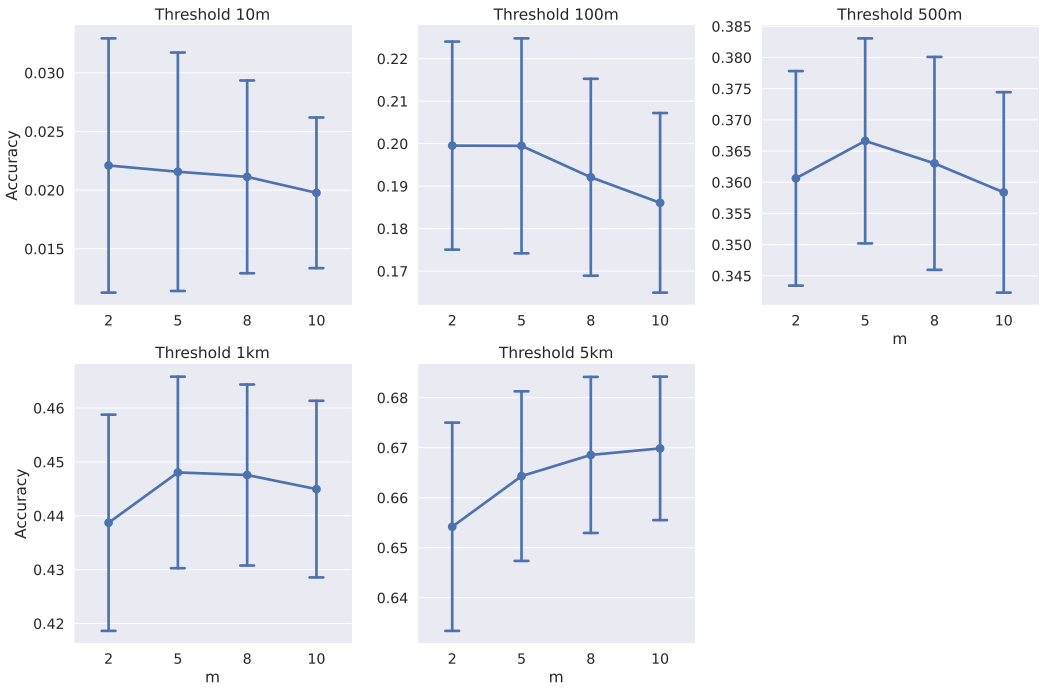


Figure 4.7: Average accuracy for different values of m. KDE with Euclidean distance.

We performed the same analysis for the KDE with cosine distance, but results are largely the same, so we won't discuss them in detail here, but we add the graphs to the appendix (see figures A.4, A.5, A.6 and A.7).

The table 4.3 below shows the best results and corresponding parameters for different distance metrics for the KDE method. We can see that the difference in the accuracy between KDE with Euclidean and cosine distances is practically negligible.

	Euclidean distance				Cosine distance			
Threshold	k	m	σ	Accuracy	k	m	σ	Accuracy
10m	80	2	0.0001	0.0388	100	10	0.0001	0.0410
100m	60	5	0.0025	0.2316	100	10	0.001	0.2342
500m	20	5	0.0075	0.3876	10	10	0.005	0.3885
1km	20	5	0.01	0.4777	20	10	0.01	0.4759
5km	100	10	0.01	0.6949	20	10	0.01	0.6879

Table 4.3: Best parameters for KDE method with Euclidean and cosine distances

4.5 Selection of the best method and performance analysis

In the previous section, we've selected the best parameters for individual methods, and for this we were using the validation set. In this section, we will choose the best models and compare their performance on the testing set. However, some models have the same best parameters across different distance thresholds, but as we've seen with the KDE model,

parameter choice determines how well the model will perform on a certain threshold. Because of this, we will only choose the model that performs best on the 100m threshold since it's hard to keep track of all the models on different thresholds. We choose the 100m threshold firstly because modern consumer-grade GPS systems have an accuracy of 5 to 10 meters, so predictions on this level may suffer from inaccuracy of GPS coordinates. Secondly, because the 100m threshold is still quite detailed and predictions at this level should be possible in the city. In the end, we will also analyze the model that performs the best on the testing set.

For this section we've chosen the next models:

1. 1-nearest-neighbour
2. Model based on averaging, with $k = 2$
3. Model based on weighted averaging, with $k = 2$ and $m = 15$
4. Model based on KDE, with $k = 60$, $m = 5$ and $\sigma = 0.0025$.

Also, since in the previous section we found that using cosine distance doesn't bring additional advantages we will only compare models using Euclidean distance since the descriptors that we obtained from the neural network were optimized for Euclidean distance.

In the figure 4.8 below we can see how methods compare on the testing set. In the figure, "kde" denotes kernel density estimation method, "nn" denotes the 1-nearest-neighbor method, "avg" and "w_avg" denote averaging method and weighted averaging method, respectively. We can see that kernel density estimation outperforms all other methods by a significant margin, except on the 5km threshold where it has the same performance as 1-nearest-neighbour, which is probably caused by the fact that we use σ tuned for 100m threshold. Also, we can see that 1-nearest-neighbour outperforms methods based on averaging.

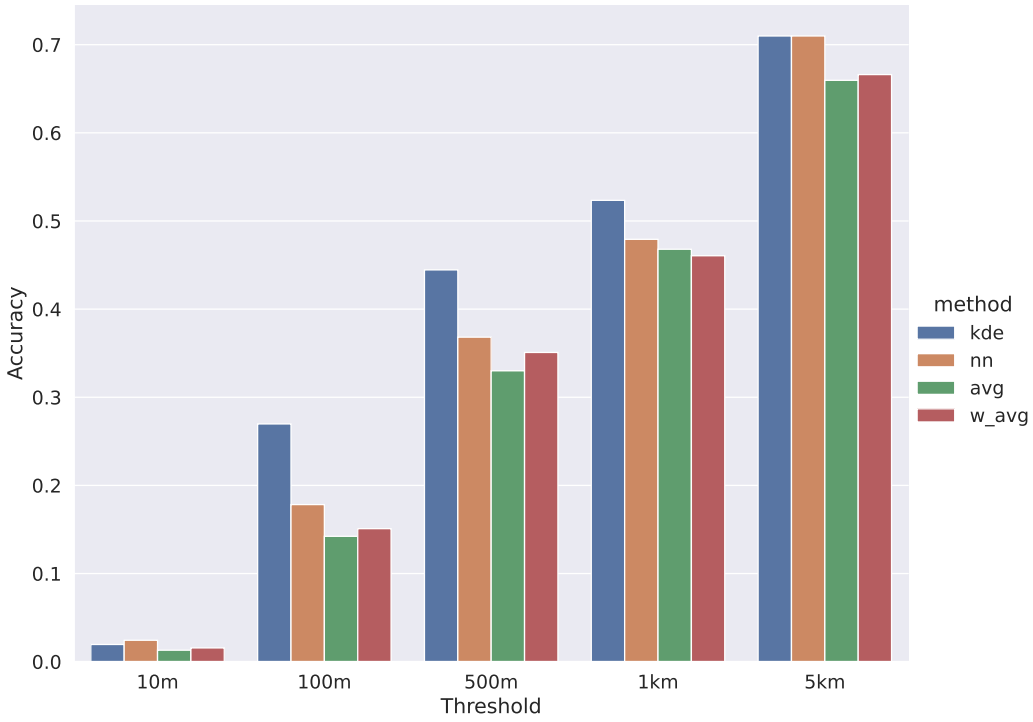


Figure 4.8: Comparison of geolocation methods

The same information but with concrete numbers is provided in the table below.

Threshold	KDE	NN	AVG	W_AVG
10m	0.0195	**0.0242**	0.013	0.0156
100m	**0.2697**	0.1782	0.1422	0.1509
500m	**0.4444**	0.3681	0.33	0.3508
1km	**0.5234**	0.4791	0.4679	0.4605
5km	**0.7098**	**0.7098**	0.6595	0.66

Table 4.4: Accuracy for different methods on different thresholds

It's clear that the KDE model performs better than the others. When we described our dataset, we saw that we had significantly more photos from Prague than from the other regions. From the figure 4.9 below we can see that the performance is significantly better in Prague than in the other regions, which is expected due to the imbalance in the dataset.

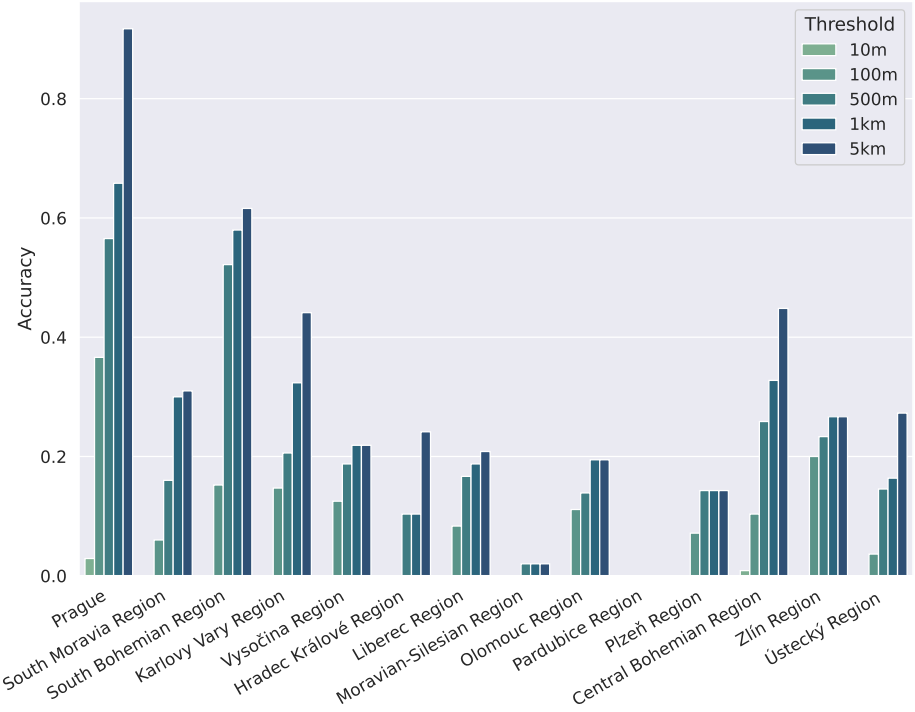


Figure 4.9: Performance of KDE model by different regions

In the table 4.5 below we compare Prague with the rest of the Czech Republic, and we can see that performance is 2 to 3 times better in Prague.

Threshold	Prague	Other
10m	0.0289	0.0014
100m	0.3662	0.094
500m	0.5641	0.2387
1km	0.6581	0.2966
5km	0.9171	0.3473

Table 4.5: Accuracy of KDE model in Prague vs the rest of Czech Republic

4.6 Summary

In this chapter, we described what kind of image representation we use in our image retrieval pipeline and discussed what type of distance metric we use to compute a distance between image descriptors. Then we described methods that we use for geolocation, namely these were 1-nearest-neighbor, averaging and weighted averaging methods and kernel density estimation method. We also discussed how we measure accuracy and that we track accuracy on 10 meters, 100 meters, 500 meters, 1 kilometer and 5 kilometers distance thresholds. Then for each geolocation model, we performed parameter selection and discussed how different parameters influence the accuracy of a model. We also determined the set of parameters that

perform the best on a given distance threshold for each model. Finally, we compared the performance of different models on the test set and saw that the KDE model outperforms the other models. However, due to our dataset's limitations, we saw a significant difference in the performance between Prague and other regions in the Czech Republic.

5 Improving Performance of KDE Geolocation Method

This chapter will describe our approach to improving the results of geolocation predictions for the method based on kernel density estimation described in the previous chapter. However, since our approach is based on artificial neural networks, the first section of this chapter will be dedicated to the theoretical overview of ANN. In the second section of this chapter, we will describe our approach in detail. And in the third section, we will show our experimental results.

5.1 Overview of Artificial Neural Networks

Artificial neural networks (ANN) are computing systems inspired by biological neural networks that constitute human or animal brains. An ANN is based on a collection of connected units called artificial neurons. An example of an artificial neuron is shown in figure 5.1 below.

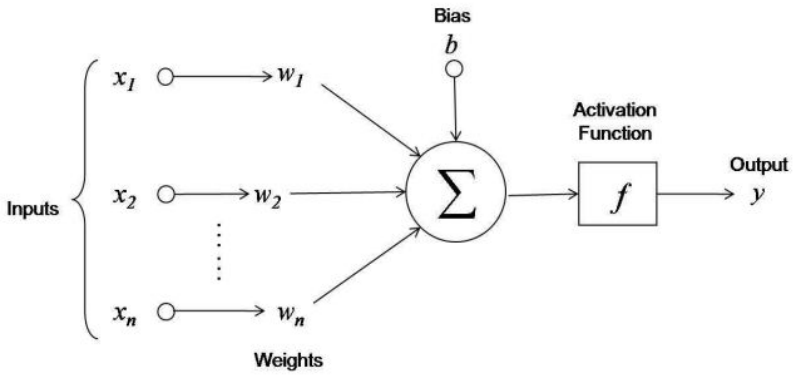


Figure 5.1: Artificial neuron. Image from [14]

Each neuron can have multiple *inputs* x_i multiplied by *weights* w_i and then summed in an adder, and with added *bias* b sent to and *activation function* f. Mathematically we can describe a neuron as follows

$$y = f(\sum_{i=1}^{n} x_i w_i + b) \tag{5.1}$$

Where f is usually some non-linear function, the most popular activation functions for ANNs are the sigmoid function, hyperbolic tangent function, rectified linear unit (ReLU), and softmax.

Neurons organized into a collection constitute a *fully connected* layer. This layer is defined by the number of inputs n and outputs m, and without an activation function can be represented as a basic linear transformation

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{5.2}$$

Where W is a matrix of weights of size $n \times m$, x is a vector of inputs of size m, and b is a vector of biases of size m.

Artificial neural networks consist of multiple layers. Also, as a useful abstraction, activation functions are separated into different layers. We use ReLU and softmax functions in this work, so let's discuss them in more detail. Mathematically ReLU function can be described as follows

$$f(x) = max(0, x) \tag{5.3}$$

Basically, ReLU returns 0 if the input x is less or equal to zero or it returns a value of input x otherwise. The advantage of the ReLU activation function is that it adds non-linearity to the network but it doesn't suffer from the vanishing gradient problem as a sigmoid or hyperbolic tangent.

The softmax function can be mathematically described in the following way. Given the input vector $\mathbf{x}$ of length n and parameter τ which is called temperature, softmax function will output vector $\mathbf{y}$ where y_i is

$$y(\mathbf{x})_i = \frac{e^{\frac{x_i}{\tau}}}{\sum_{j=1}^{n} e^{\frac{x_j}{\tau}}} \tag{5.4}$$

The softmax function will output a vector of values between 0 and 1 that will sum to 1. The output of the softmax function represents a probability distribution, and parameter τ helps control the final distribution. The softmax function is often used as a smooth and differentiable alternative to argmax function.

As we've seen, a fully connected layer has weights $\mathbf{W}$ and biases $\mathbf{b}$. These are the trainable parameters of such a layer. A neural network can consist of multiple layers and has different trainable parameters. Let's define Θ as a set of all trainable parameters of a network. To train a network means we what to find such a set of trainable parameters Θ^* that will minimize a *loss function* $\mathcal{L}(\Theta)$

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) \tag{5.5}$$

There exists a variety of different loss functions, depending on the task. For example, for the regression task, we can use mean squared error, for the classification task, we can use cross-entropy loss, and for metric learning, triplet loss or contrastive loss is used. In this work, we will use cross-entropy loss so let's discuss it further. Firstly, neural networks

are trained on a set of training examples $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)|i \in \{1, 2, ..., N\}\}$. Here $\mathbf{x}_i$ is an input vector, and $\mathbf{y}_i$ is a target vector of size C which corresponds to the number of classes. Additionally, a vector $\mathbf{y}_i$ is one-hot encoded, meaning that y_{ic} is equal to 1 if example i belongs to class c otherwise it's 0. With this, we can formulate cross-entropy loss as follows

$$\mathcal{L}(\Theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{5.6}$$

where $\hat{y}_{ic}$ is c-th element of output of the network for example i from the training set.

Another critical aspect to discuss is how to minimize the loss function with respect to all parameters of a network. There are two crucial algorithms at the heart of loss minimization: *the backpropagation algorithm*, and the *gradient descent*. The backpropagation algorithm tells how to compute a gradient of the loss function with respect to its parameters $\nabla\mathcal{L}(\Theta)$.

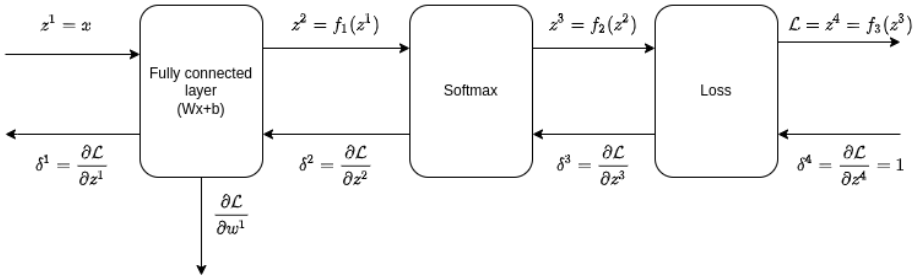


Figure 5.2: Neural network illustration with backpropagation. Image from [21]

In the figure 5.2 above we can see an example of a neural network that consists of 3 layers: a fully connected layer with parameters w^1, softmax, and loss function. Here z^1 indicates input to the network, z^2 is the output of the fully connected layer and etc. If we would want to compute the derivative of loss with respect to parameters of the fully connected layer, we could use the derivative chain rule, and we would get

$$\frac{\partial \mathcal{L}}{\partial w^1} = \frac{\partial \mathcal{L}}{\partial z^3}\frac{\partial z^3}{\partial z^2}\frac{\partial z^2}{\partial w^1} \tag{5.7}$$

However, for big networks, it's impractical. Therefore, backpropagation uses divide and conquer approach and defines the so-called sensitivity.

$$\delta_i^l = \frac{\partial \mathcal{L}}{\partial z_i^l} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1}\frac{\partial z_j^{l+1}}{\partial z_i^l} \tag{5.8}$$

Where subscript i indicates the fact that $\mathbf{z}^l$ and δ^l are vectors. And superscript l indicates a layer.

Notice that sensitivity is defined recursively, and in order to compute the sensitivity of l-th layer, we need to compute the sensitivity of layer $l + 1$.

Lastly, we can compute the derivative of loss with respect to some parameters w^l as follows

$$\frac{\partial \mathcal{L}}{\partial w_i^l} = \sum_j \frac{\partial \mathcal{L}}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial w_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial w_i^l} \tag{5.9}$$

From the equations 5.8 and 5.9, we can see that for each layer, we need to define only the derivative of output w.r.t. input $\frac{\partial z^{l+1}}{\partial z^l}$. If a layer has trainable parameters, we also need to define the derivative of output w.r.t. parameters $\frac{\partial z^{l+1}}{\partial w^l}$ for the backpropagation algorithm to work.

After we have computed the gradient of loss, we can use gradient descent to update the network's parameters iteratively

$$\Theta_{k+1} = \Theta_k - \alpha \nabla \mathcal{L}(\Theta_k) \tag{5.10}$$

Where Θ_k is a set of network parameters at step k, and α is called learning rate.

Finally, it's usually infeasible to compute gradient descent steps on the whole training set due to memory constraints. Therefore, for each iteration of gradient descent, we take only part of the examples from the training set and compute the gradient only for these examples. For the next step, we take another subset of examples that haven't been seen yet, and we repeat this until we use all examples from the training set. These subsets of the training set are called *mini-batches*, and when we iterated over all mini-batches, this is called one *epoch*, and we can train a network for multiple epochs.

5.2 Proposed solution overview

In the chapter 4 we've seen that the geolocation model based on kernel density estimation performed better than 1-nearest-neighbor and averaging models. In this section, we will try to improve the results of the KDE model even more. The obvious solution to this problem would be to learn a new image representation on the dataset that is annotated specifically for this task. However, as we discussed in the chapter 2, the problem with this approach is that creating annotations for the task of metric learning is a challenging problem that may require a lot of manual processing. Instead, we can optimize the existing descriptor that we obtained from the CNN by Radenovic *et al.* [19]. This means that instead of learning completely new descriptors from scratch, we can try to learn some transformation $T : \mathbb{R}^{2048} \mapsto \mathbb{R}^{2048}$ that will apply to descriptors to improve the results of the KDE model. Our motivation for choosing this approach is that it's a potentially easier task to tune existing descriptors rather than learn an entirely new representation.

5.2.1 Neural network

We propose to learn transformation $T : \mathbb{R}^{2048} \mapsto \mathbb{R}^{2048}$ using the neural network shown in the figure 5.3 below.

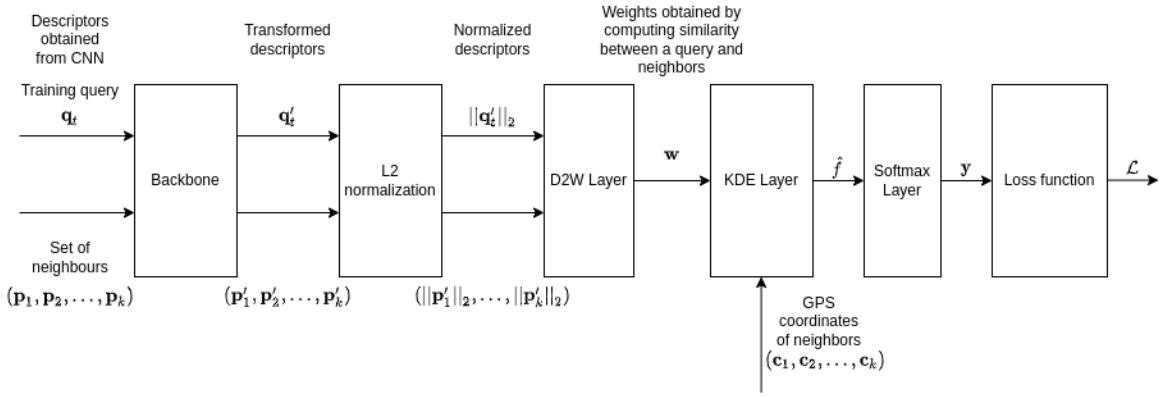


Figure 5.3: The architecture of the proposed neural network

The first thing we can see in the image is that we feed a descriptor of a training query $\mathbf{q_t}$ and the set of descriptors of neighbors $(\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k)$ for this query to the module called *backbone*. We will discuss later how we prepare training examples $(\mathbf{q_t}, (\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k))$, for now let's describe the architecture of the network. The backbone is basically a transformation function that we want to learn. In this work we will experiment with two different backbones:

1. Backbone that consists of a single fully connected layer with the number of inputs and outputs equal to 2048.

2. Backbone that consists of the fully connected layer, ReLU, and another fully connected layer. Both fully connected layers have the number of inputs and outputs equal to 2048.

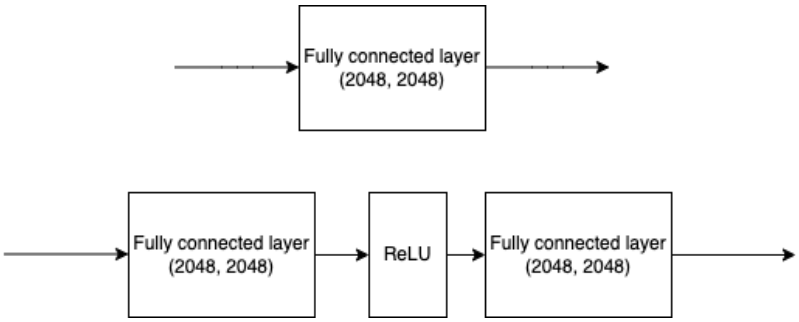


Figure 5.4: Illustration of two configurations of the network's backbone

After the input descriptors are transformed in the backbone, they are normalized in the L2 normalization layer.

After the normalization layer, we can see two custom layers, the first one is the descriptors-to-weights (D2W) layer, and the second one is the KDE layer.

D2W layer computes similarity as described in the equation 4.5 between transformed training query $||\mathbf{q}'_t||_2$ and neighbours $(||\mathbf{p}'_1||_2, ..., ||\mathbf{p}'_k||_2)$. The output of the D2W layer is vector $\mathbf{w}$ whose length is equal to k. D2W layer has parameter m that corresponds to

exponential term in the equation 4.5, we can choose whether we want this parameter to be trainable or not.

KDE layer performs kernel density estimation as described in the equation 4.7. As input it accepts weights $\mathbf{w}$, and a set of GPS coordinates $\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_k\}$ that correspond to GPS coordinates of neighbours $(\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k)$. Similarly as we described in section 4.2 we sample GPS space to create a set of points $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$ on which we will compute KDE. The output of KDE layer is vector $\hat{f}$ which represents a density function. The size of this vector is equal to $|\mathcal{X}| = N = k(2bn+1)^2$ where $b = 3$ and $n = 3$, the same as was discussed in section 4.2. We can mathematically describe the KDE layer by rewriting the equation 4.7 using notation from this section:

$$\hat{f}(\mathbf{x}) = \frac{1}{(\sum_{i=1}^{k} w_i)|\Sigma|^{\frac{1}{2}}} \sum_{i=1}^{k} w_i \mathcal{N}(\mathbf{x}; \mathbf{c}_i, \Sigma) \tag{5.11}$$

where w_i is i-th element of vector $\mathbf{w}$, $\mathbf{c}_i \in \mathcal{C}$, and $\mathbf{x} \in \mathcal{X}$. Lastly, $\Sigma = \sigma^2\mathbf{I}$, where σ is a parameter of the KDE layer, and we can choose whether we want this parameter to be trainable or not.

After D2W and KDE layers, we can see the softmax layer and a loss function in figure 5.3. The softmax function was described in the equation 5.4, and the loss function that we use is cross-entropy loss described in the equation 5.6. The fact that we use softmax and cross-entropy loss is given by how we formulate the optimization problem. The KDE layer outputs a vector, where each each element represents a value of density function $\hat{f}$ for each point $\mathbf{x} \in \mathcal{X}$. From section 4.2 we can remember that we predict coordinates of a query as $\mathbf{x}^* = \text{argmax}_{\mathbf{x}\in\mathcal{X}} \hat{f}(\mathbf{x})$. We cannot use argmax function in the neural network since it's not differentiable. Instead, we can use the softmax function as a smooth substitute for argmax, and using cross-entropy loss, we can penalize the wrong prediction of coordinates. In other words, we frame training of neural network as a classification problem, where we predict coordinates of a query by classifying it into one of N classes, where $N = |\mathcal{X}| = k(2bn+1)^2$, and each class corresponds to one point in set $\mathcal{X}$ of GPS points.

After we have trained the neural network, the idea is to obtain new descriptors by discarding all layers after the L2 normalization layer and feed descriptors obtained from CNN [19] to our network, see figure 5.5.

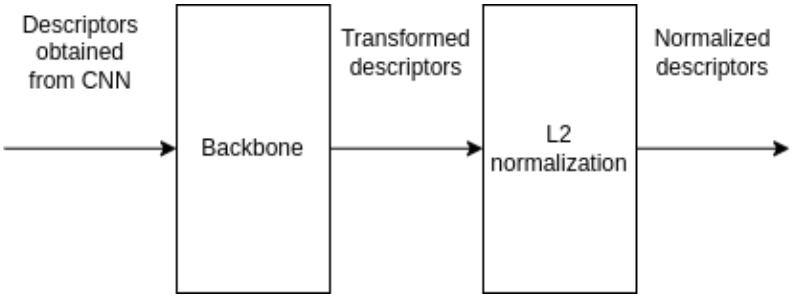


Figure 5.5: Illustration of our neural network during inference

In figure 5.3 we saw that we need to provide two inputs to the backbone, a training query and corresponding neighbors. While in figure 5.5 we only need to provide one input

to the backbone. This might be confusing, but our implementation has only one input for the backbone and L2 normalization layer (though the D2W layer does require two inputs). And what we actually do is that we firstly pass training query through the backbone and then pass neighbors through the backbone, and the same for the L2 normalization layer. For our implementation, we've chosen the PyTorch framework and in this framework, passing one input through the fully connected layer and then the other input through the same layer during training will result in accumulating of gradients.

We illustrate the difference between forward propagation during training and inference in the listings 5.1 and 5.2, respectively. Code in these listings is Python-like pseudo-code, and it does not represent the actual code from our implementation but serves as an illustration.

```
def forward(self, training_query, neighbours, neighbour_coordinates):
    """
    self.backbone - represents backbone of the network
    self.l2_norm_layer - represents L2 normalization layer
    self.d2w - represents descriptors-to-weights layer
    self.kde - represents the KDE layer
    self.softmax - represents softmax layer
    """
    transformed_query = self.backbone(training_query)
    transformed_neighbours = self.backbone(neighbours)

    normalized_query = self.l2_norm_layer(transformed_query)
    normalized_neighbours = self.l2_norm_layer(transformed_neighbours)

    w = self.d2w(normalized_query, normalized_neighbours)

    kde_f = self.kde(w, neighbour_coordinates)

    return self.softmax(kde_f)
```

Listing 5.1: Forward propagation of the network during training

```
def forward(self, descriptors):
    """
    self.backbone - represents backbone of the network
    self.l2_norm_layer - represents L2 normalization layer
    """
    transformed_descriptors = self.backbone(descriptors)
    normalized_descriptors = self.l2_norm_layer(transformed_descriptors)

    return normalized_descriptors
```

Listing 5.2: Forward propagation of the network during inference

5.2.2 Training Examples

In the previous subsection, we saw that for training the neural network, we need to provide inputs, such as training query $\mathbf{q}$, a set of neighbors $(\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k)$ as well as the set of coordinates of the neighbors $(\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_k)$, and additionally, we need to provide the classification target for each query. In this subsection, we will describe how we generate these training examples. To prepare the collections of training examples, we use training queries set as a

source of query images, and we use database set as the source of neighbors. Additionally, we use queries from validation queries set to perform validation. For this work, we have prepared two collections of training examples, the first one we call the *"top70" training set*, and the second one we call the *"intersection" training set*.

Let's start by describing the "top70" set. For each query from the training queries set, we search for 70 nearest-neighbors in the descriptors space. This means that we find 70 descriptors, from the database set that have the lowest Euclidean distance from the query descriptor for each training query. Then for each query, we compute its prediction target on the fly during training. The target is computed as follows. Given the true coordinates of a training query, we find the closest coordinates in the space of generated GPS coordinates $\mathcal{X}$. Remember that the set $\mathcal{X}$ represents a space of GPS coordinates that we generated to compute KDE. The motivation for training the network on these training examples is that we assume that the top 70 neighbors that we get are indeed similar, however, the most similar neighbor in descriptor space is not the closest neighbor in coordinates space. So by using these training examples, we hope to optimize descriptors such that transformed descriptors will become closer if they are similar and closer in GPS space.

We generate the "intersection" training set in the following way. For a query image, we find 100 closest images from the database set in the descriptor space (let's call this set D). Additionally, we find 100 nearest images from the database set in GPS space (let's call this set C). Then we select all those photos that are both in the set D and in the set C. We call these selected photos the intersection. We add all images from the intersection to the list of neighbors. For this collection, we have 100 neighbors in total. After we added the intersection to the list, we fill the rest by adding an even amount of photos from D and C sets. For each query, we compute the geographical distance between the query and all images from the intersection and we assign the target to each query as coordinates of the closest image from the intersection. However, there are cases when the intersection can be empty. In these cases, we evenly add images from the D and C spaces to the list of 100 neighbors and the target, and we assign the target as the coordinates of the closest image (in terms of GPS distance) from the D space. Motivation for using these training examples is similar to the motivation of using the "top70" set, however, we also add points that are close in GPS space but not necessarily similar. So we hope that during training, the network will learn a transformation that will optimize descriptors that are similar and close in GPS space but will keep further in descriptor space those points that are close in GPS space but far in descriptor space.

For both "top70" and "intersection" training sets, we create respective validation sets, in the same manner, using validation queries instead of training queries.

5.3 Experiments and results

This section will present our experiments on training the proposed network and discuss the results. We trained networks with both 1 layer backbone and 2 layers backbone, and we trained them on both training sets. Before discussing the experiments, let's first describe how we perform training. During training, we have three major steps—training step, validation step, and testing step.

During the training step, we split the training set into mini-batches and perform optimization. During this step the weights are updated. For each training step, we record the average loss across all mini-batches.

We split the validation set into mini-batches during the validation step and compute the average loss on all mini-batches. We use validation loss to estimate how the model performs on the data that it was not trained on.

We use the network's backbone to transform descriptors from the database set and validation queries sets during the testing step. Using the transformed descriptors, we perform KDE geolocation prediction, and we record the accuracy of the prediction on five different thresholds: 10m, 100m, 500m, 1 km and 5km. We use this step to evaluate how transformed descriptors influence predictions of the KDE method.

Additionally, we will train the network using two different optimizers and then select the optimizer that suits better. We use Stochastic Gradient Descent (SGD) with Momentum and Adam optimizer. For the SGD with momentum, we choose the value of momentum $\gamma = 0.9$ and keep it fixed across all experiments. For both SGD and Adam, we will need to pick the appropriate learning rate α.

5.3.1 Training network with one layer backbone

Let's start by training the network with one layer backbone on the "top70" training set.

Our network is quite small, therefore it doesn't take much time to train. We can utilize this to find the best combination of optimizer and learning rate by running multiple training procedures for three epochs and record the loss and testing accuracy. We will train the network with the following combinations of optimizer and learning rate:

- For SGD learning rate $\alpha \in \{0.001, 0.005, 0.0075, 0.01, 0.05, 0.1, 0.5\}$
- For Adam $\alpha \in \{0.00005, 0.00001, 0.0005, 0.001, 0.01, 0.1\}$

Also, for this experiment, we set the parameter of the D2W layer $m = 3$, and we set the KDE layer $\sigma = 0.001$ and make them both constant, i.e. we don't learn these parameters during training.

The figure 5.6 shows training and validation losses for each epoch. We can see that for the majority of runs, losses decrease.

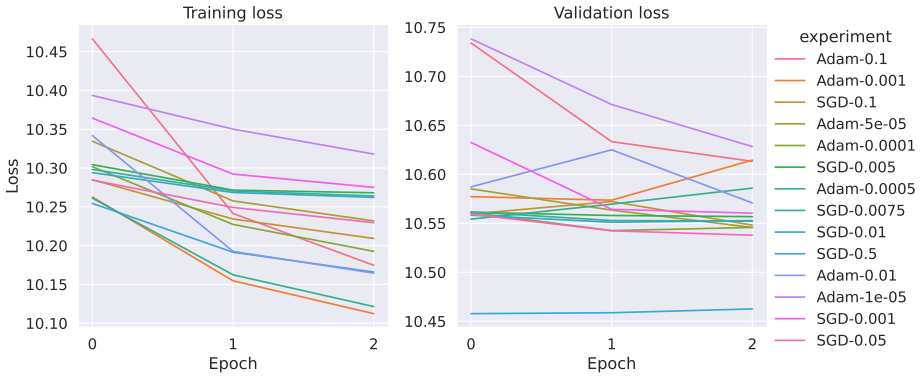


Figure 5.6: Training and validation losses

However, let's look at the figure 5.7, which shows the accuracy of the KDE model for transformed descriptors. We can see that the KDE model's performance significantly reduces for some combinations of optimizer and learning rate . Also note, that on the x-axis of this figure, we have -1 as the first epoch. This simply means that we run a testing step before any training has started.

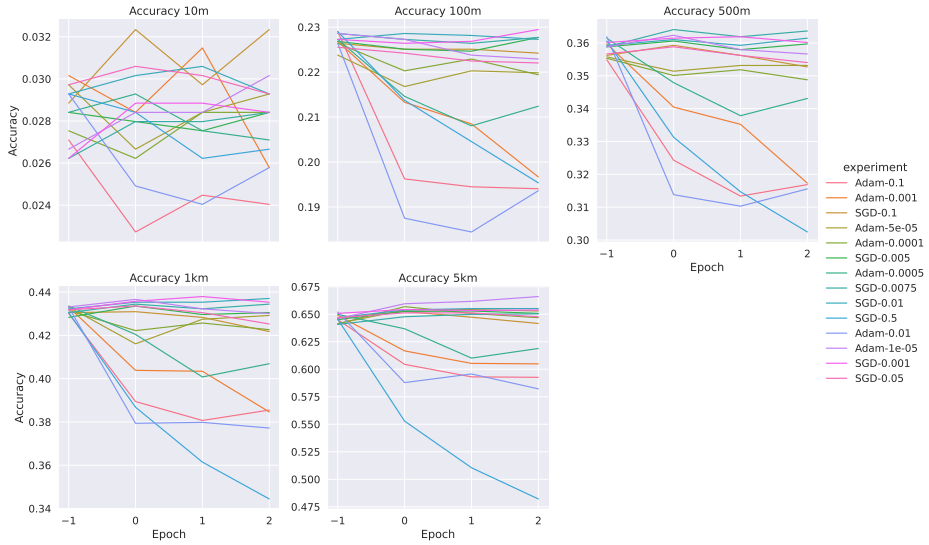


Figure 5.7: Testing accuracy for multiple runs when training the network

In the figure 5.8 we show accuracy for the same runs, but we remove those runs that significantly reduce performance from the graph, so we could better see the rest. From the figure, it seems like the change of accuracy for most remaining runs is insignificant. It also appears that accuracy tends to decrease.

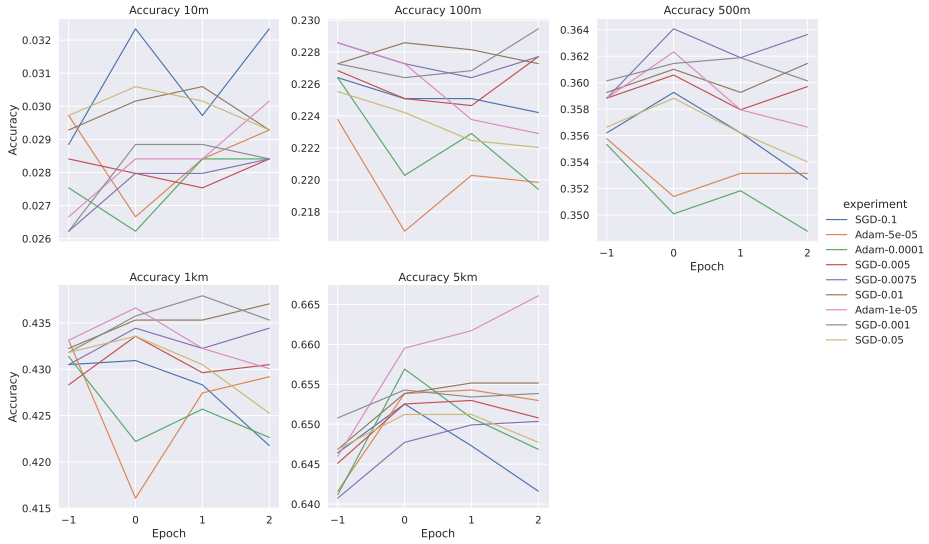


Figure 5.8: Testing accuracy for multiple runs when training the network

From the figure 5.8 it seems that training with SGD optimizer and learning rate $\alpha = 0.001$ produces the best results on 100m threshold, even though the change in the accuracy is insignificant. We try to train the model with these parameters for eight epochs to see if this will help to improve the results. However, in the figure 5.9 we can see no significant improvement in the accuracy.

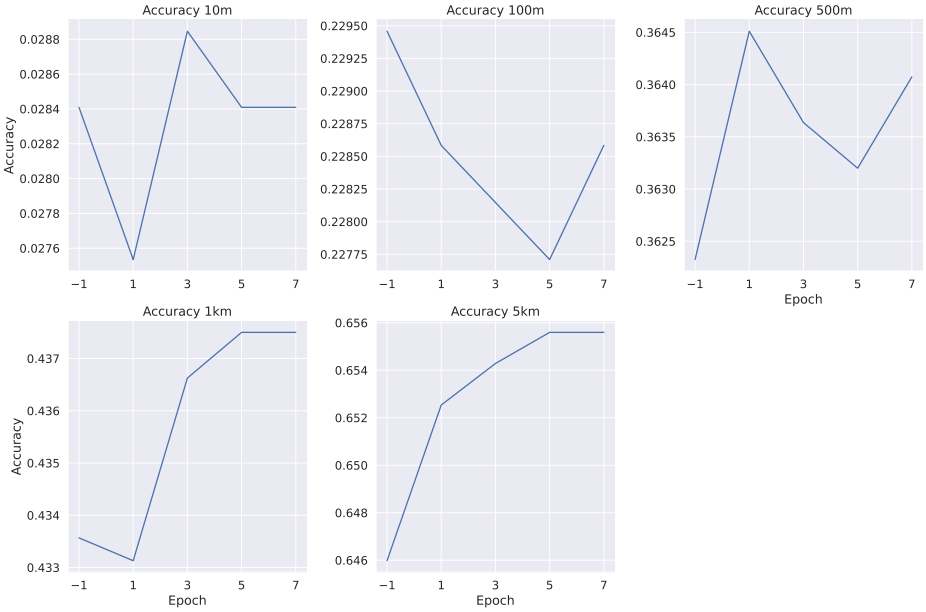


Figure 5.9: Training of the one-layer model on "top70" dataset using SGD with momentum and $\alpha = 0.001$

We further train the network on the "intersection" training set to see if this will change the outcome.

We also start by training the network using different optimizers and learning rates. For this experiment, we use fewer combinations than for the previous one, namely

- For SGD learning rate $\alpha \in \{0.001, 0.01, 0.1, 0.5\}$
- For Adam $\alpha \in \{0.00005, 0.00001, 0.0005, 0.0001, 0.001\}$

In the figure 5.10 we can again see that for the majority of combinations performance of the KDE model drops significantly. However, for SGD optimizer with learning rate $\alpha = 0.05$ and $\alpha = 0.1$ we can see significant improvements on the 10m and 5km thresholds.

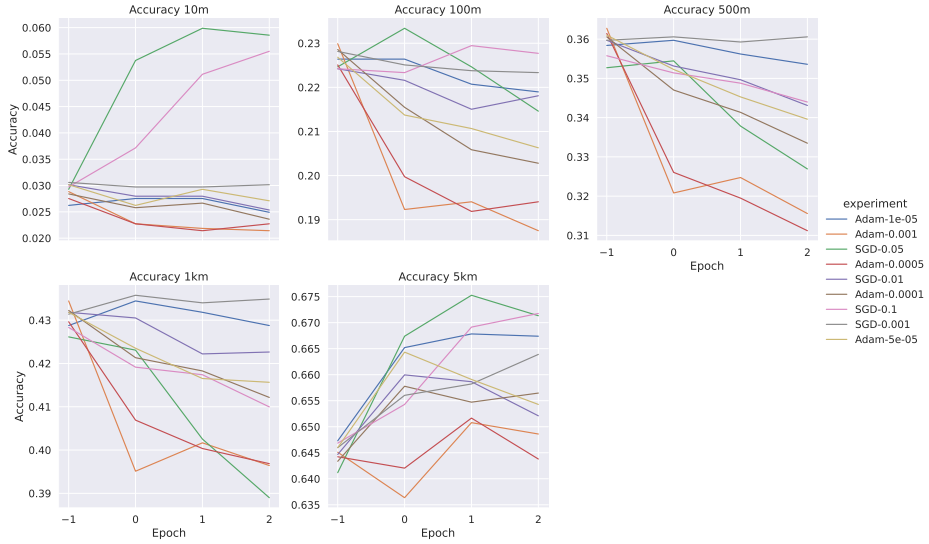


Figure 5.10: Testing accuracy for multiple runs when training the network

We train the network using SGD optimizer with a learning rate $\alpha = 0.05$ and $\alpha = 0.1$ for six epochs to see if this will result in additional improvements. The testing accuracy for respective parameters is shown if figures 5.11 and 5.12. From these graphs, we can see that additional training does not further improve results.

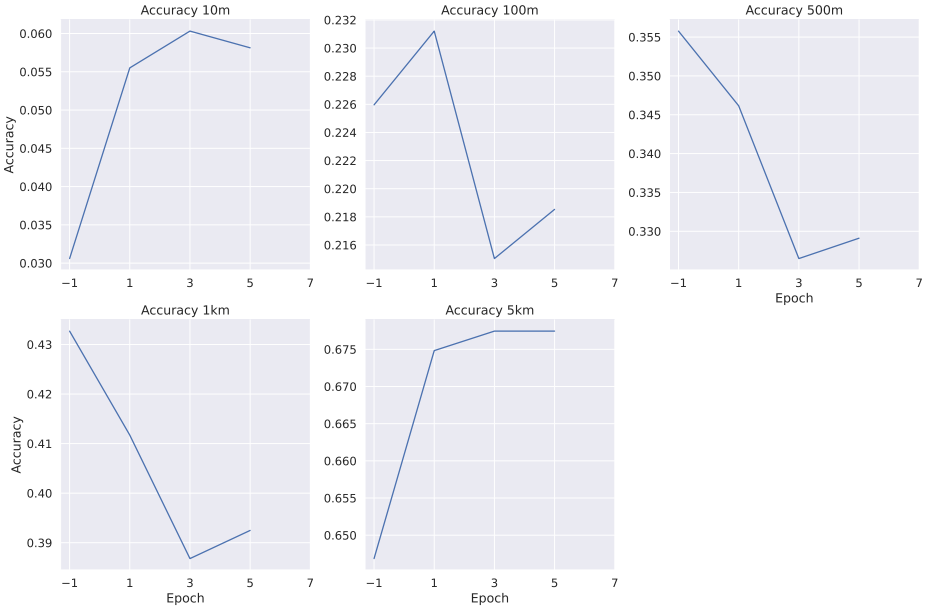


Figure 5.11: Testing accuracy for training the network on the ”intersection” training set with SGD optimizer and learning rate $\alpha = 0.05$

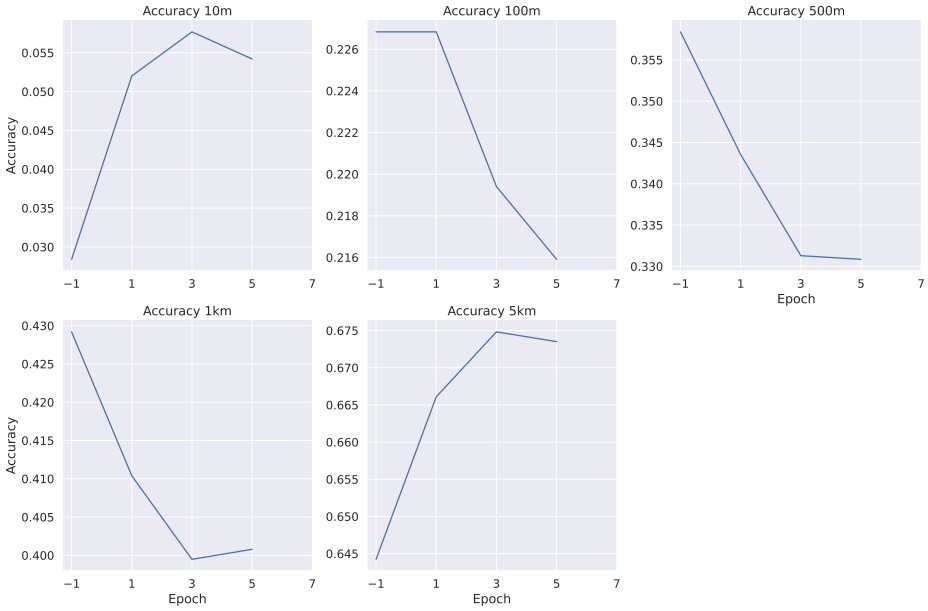


Figure 5.12: Testing accuracy for training the network on the "intersection" training set with SGD optimizer and learning rate $\alpha = 0.1$

In chapter 4 we saw that parameter σ of the KDE methods influences accuracy on certain thresholds. We try to set $\sigma = 0.005$ and train the network on these data to see if this can help improve results not only on 10m thresholds but also on 100m and 500m. However, we saw no improvements. We also try to train the network when the parameter σ is trainable. However, during training, σ becomes very large, which significantly reduces the performance of the KDE method.

5.3.2 Training network with two-layer backbone

In this subsection, we train a network with two-layer backbone. The motivation for this is to see if using a more complex architecture, compared to the 1 layer backbone, can improve the performance of the KDE geolocation method. First, we will train the network on the "top70" training dataset and then on the "intersection" training dataset.

We also start by training the network using different optimizers and learning rates. For this experiment, we use the following combinations

- For SGD learning rate $\alpha \in \{0.001, 0.01, 0.1, 0.5\}$
- For Adam $\alpha \in \{0.00005, 0.00001, 0.0005, 0.0001, 0.001\}$

We see a similar situation in the figure 5.13 as in the previous experiments on one layer backbone.

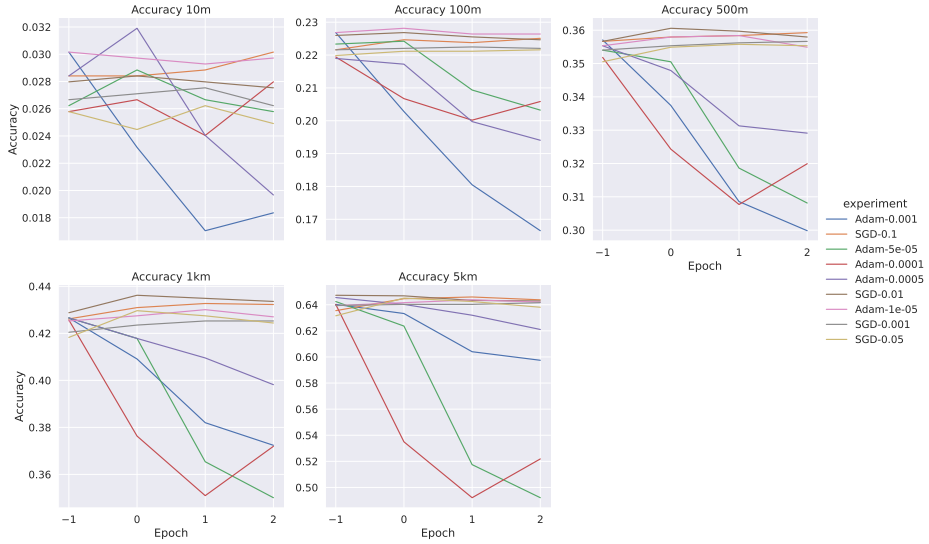


Figure 5.13: Testing accuracy for multiple runs when training two-layer backbone on "top70" training dataset

For some combinations, testing accuracy reduces significantly. Therefore we remove them from the graph, and in the figure 5.14 we show results without those combinations. From this figure, we don't see any improvements in the accuracy of the KDE method.

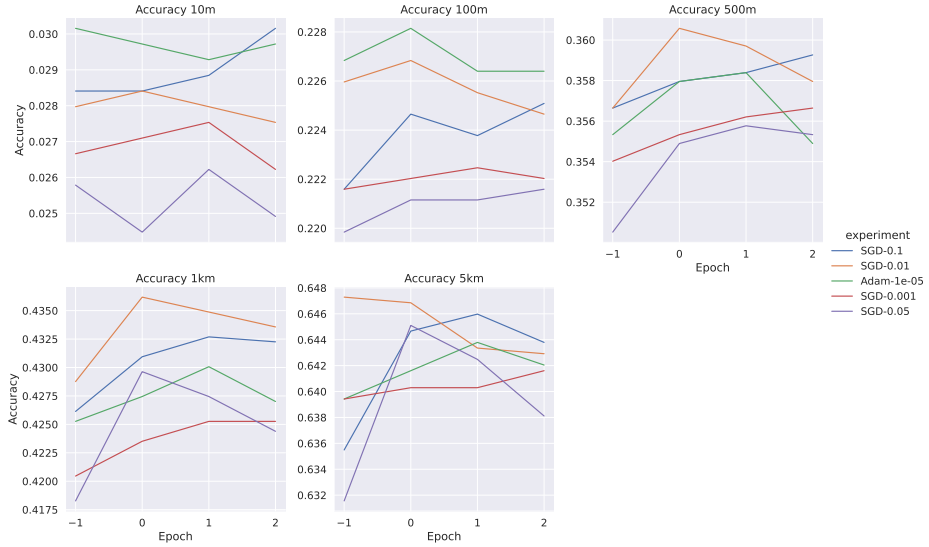


Figure 5.14: Testing accuracy for multiple runs when training two-layer backbone on "top70" training dataset

Additionally, we train the network with a two-layer backbone on the "intersection" training set. Similarly, as in previous experiments, we train the network for different combinations of optimizer and learning rate. We use the same combinations as in the last experiment.

In the figure 5.15 we see similar results as when we trained the network with a one-layer

backbone. The only improvement we can see is that on 10m threshold when trained with Adam optimizer with learning rate $\alpha = 0.00005$.

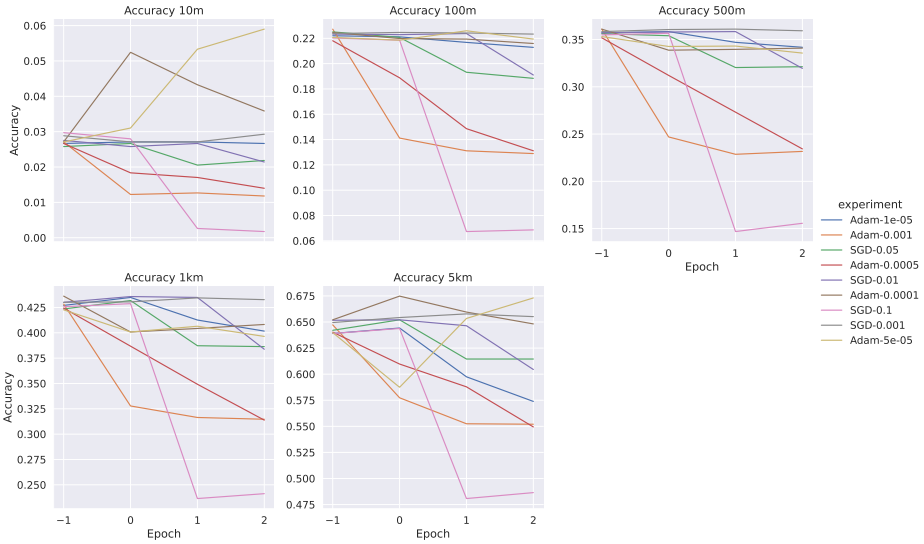


Figure 5.15: Testing accuracy for multiple runs when training two-layer backbone on "intersection" training dataset

We also tried to train the network with Adam optimizer and learning rate $\alpha = 0.00005$ for six epochs. However, performance starts to decrease after the third epoch.

5.3.3 Results

In the experiments described in previous subsections, we tried to train the neural network to improve image representations for the KDE geolocation method. We trained two versions of the network with different sets of hyper-parameters and optimizers on two different training sets. As we can see from those experiments, we could not significantly improve the results of geolocation prediction with the chosen approach. However, there were minor improvements in geolocation prediction accuracy, namely on the 10m and 5km accuracy thresholds.

We would like to know how the performance of the KDE geolocation method with our optimized descriptors compares to the performance with the original descriptors on the testing queries set. When we were training the network, during testing we performed KDE geolocation with parameters $k = 70$, $m = 3$ and $\sigma = 0.001$. However, the best parameters for the KDE method with original descriptors were $k = 60$, $m = 5$, $\sigma = 0.0025$. So let's see how KDE with optimized descriptors compares to this original KDE. For this, we transform descriptors from the database set and testing queries set using our neural network with one layer backbone, trained on the "intersection" training set. After that we will perform KDE geolocation prediction with transformed descriptors, using both $(k = 70, m = 3, \sigma = 0.001)$ and $(k = 60, m = 5, \sigma = 0.0025)$ sets of parameter, and we will compare it to the KDE method with original descriptors.

Threshold	KDE optimized ($k = 70, m = 3, \sigma = 0.001$)	KDE optimized ($k = 60, m = 5, \sigma = 0.0025$)	KDE original ($k = 60, m = 5, \sigma = 0.0025$)
10m	0.039	0.029	0.019
100m	0.222	0.228	0.269
500m	0.346	0.378	0.444
1km	0.411	0.441	0.52
5km	0.705	0.715	0.709

Table 5.1: Accuracy of KDE with optimized descriptors vs. KDE with original descriptors on different distance thresholds

This table shows that transformed descriptors outperform the original by 2% and 1% on the 10m threshold. However, the difference on the 5km threshold is insignificant. Moreover, this performance gain on the 10m threshold comes at the price of reduced performance on 100m, 500m and 1km thresholds.

Also, in chapter 4 we discussed that choosing different sigma influences the accuracy on different thresholds. Let's compare KDE geolocation with original and transformed descriptors using more suitable parameters for predictions on the 10m threshold.

Threshold	KDE optimized ($k = 60, m = 3, \sigma = 0.0001$)	KDE original ($k = 60, m = 3, \sigma = 0.0001$)
10m	0.052	0.038
100m	0.179	0.226
500m	0.302	0.392
1km	0.364	0.477
5km	0.692	0.665

Table 5.2: Accuracy of KDE with optimized descriptors vs KDE with original with parameters suitable for 10m threshold

The table above shows that the methods with transformed descriptors perform slightly better on 10m and 5km distance thresholds. However, compared to the KDE method with original descriptors, it performs significantly worse on 100m, 500m and 1km thresholds.

Lastly, let's do the same, but now choose more suitable parameters for predictions within the 5km threshold. The table below shows that KDE with optimized descriptors performs slightly worse on 5km and slightly better on 10m thresholds. But again, the performance of KDE is significantly worse on 100m, 500m and 1km thresholds with optimized descriptors.

Threshold	KDE optimized ($k = 60, m = 10, \sigma = 0.05$)	KDE original ($k = 60, m = 10, \sigma = 0.05$)
10m	0.013	0.005
100m	0.108	0.121
500m	0.330	0.386
1km	0.456	0.530
5km	0.730	0.740

Table 5.3: Accuracy of KDE with optimized descriptors vs KDE with original with parameters suitable for 5km threshold

Overall we didn't manage to improve the performance of the KDE geolocation method using transformed KDE. When we managed to improve performance on the 10m threshold, it was at a price of worse performance on other thresholds. is significantly worse on 100m, 500m and 1km thresholds

6 Summary

This thesis aimed to build an image localization system that relies on a content-based image retrieval pipeline, given a large dataset of geotagged images, and seek improvements of the system by learning an improved image representation. In this final chapter, we give a brief summary of this thesis.

In chapter 1 we gave a brief introduction to this thesis, stated the goal, and outlined the structure of the thesis.

In chapter 2 we provided an overview of the theoretical background of content-based image retrieval and its application in image geolocation. We studied how compact image representations are obtained using classical methods and convolutional neural networks. We also studied how the problem of image geolocation is solved through content-based image retrieval by reviewing related work.

In chapter 3 we described how we collected the dataset for this thesis. We used an image hosting platform called Flickr to download publicly available photos taken in the Czech Republic and had GPS coordinates. In total, we collected 230883 geotagged photos. We also provided a simple analysis of collected data and showed that the majority of photos in our dataset were taken in Prague. We also described how we split our data into 4 datasets: database set, training queries set, testing queries set, and validation queries set. The database set represented the biggest part, and it's a set where we were looking for similar images to the query images. Validation and testing query sets were used as datasets of query images for validation and testing. We used the training queries set to train the model to improve the results of the geolocalization model.

In chapter 4 we described the image representation that we use and that we obtained them from the convolutional neural network proposed by Radenovic *et al.* [19]. Then we defined the distance metrics and similarity functions that we use to determine the similarity between images. Further, we introduced the methods that we used to predict image geolocation; namely, we described 1-nearest-neighbor methods, two methods that use coordinates averaging to predict the location, one which uses simple averaging and the other uses weighted averaging, and lastly we described the geolocation method based on kernel density estimation. We also mentioned how we calculate the methods' accuracy and measure accuracy on different distance thresholds. Later we performed experiments on these methods to determine which parameters give the best results and how different parameters influence the accuracy of a model. Lastly, we chose the best models and compared them, and we've seen that the geolocation method based on kernel density estimation outperforms the other methods.

Lastly, in chapter 5 we proposed a method to improve the performance of the KDE method by learning an improved image representation. Since our approach is based on the artificial neural network, we firstly reviewed a theoretical background of ANNs. We discussed what are neural networks and how they are trained. We described our approach to improving the results of the KDE method by learning some transformation that could be applied to existing descriptors. We discussed in detail the architecture of our neural network and all inputs and outputs, as well as we described how we generate the training example on which we train the neural network. Lastly, we tried to train the neural network. However, we showed that we were not able to obtain image representations that would significantly improve the performance of the kernel density estimation method, though we were able to get some minor improvements on the 10-meter accuracy threshold. Still, this improvement came at the cost of reduced performance on higher distance thresholds.

A Appendix

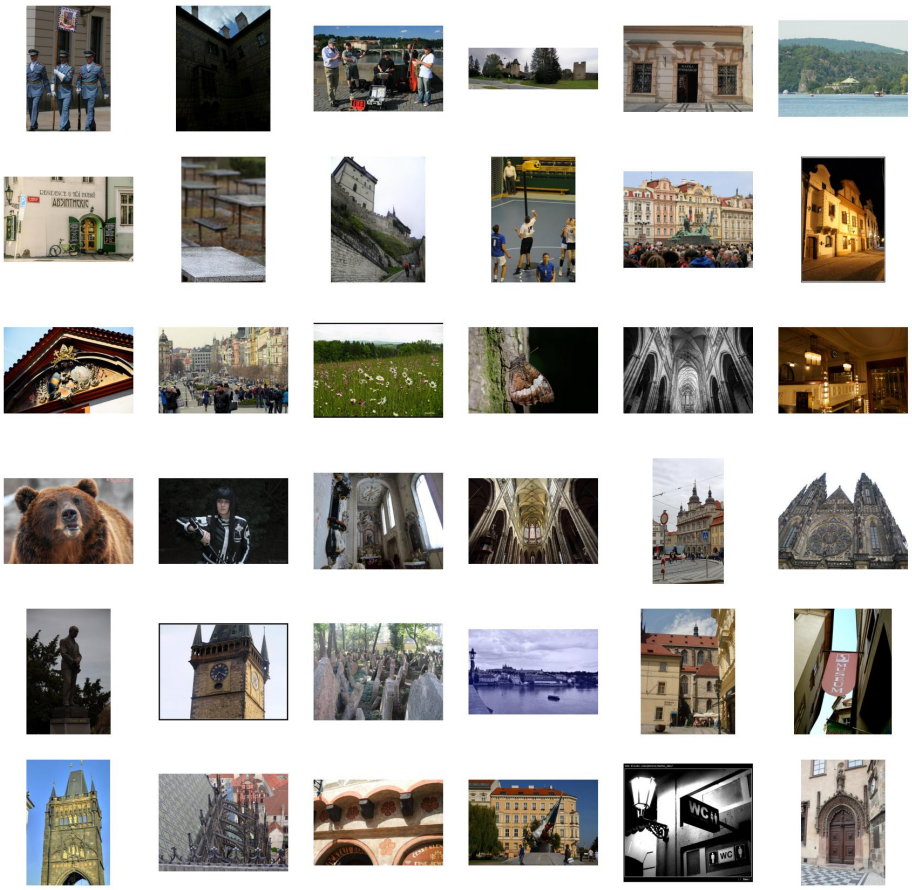


Figure A.1: Example of 36 images from the dataset

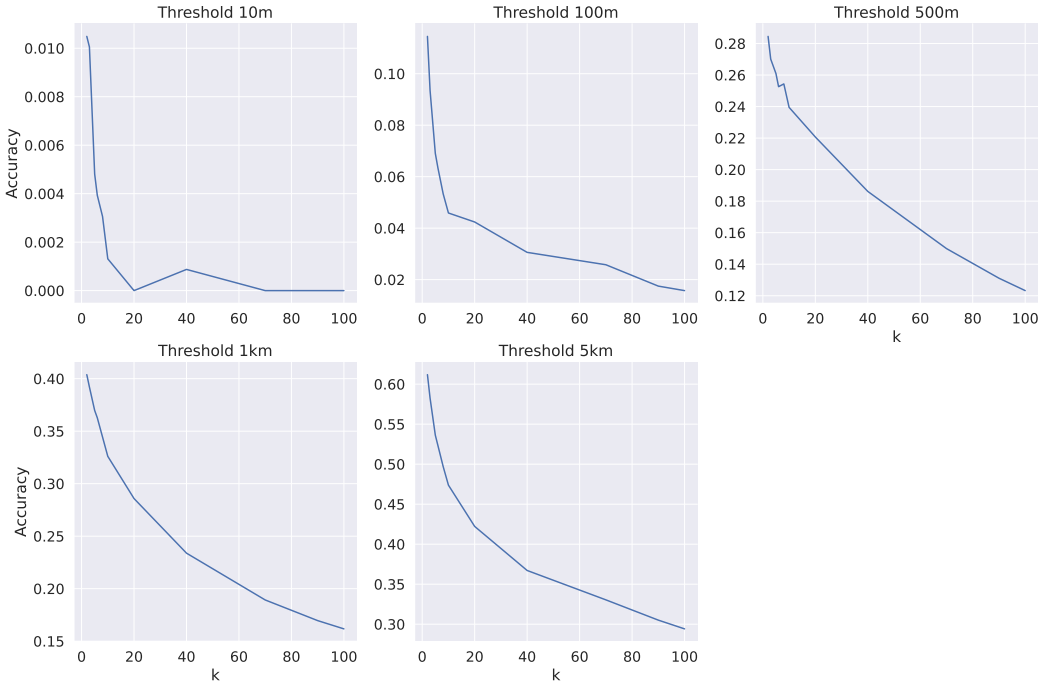


Figure A.2: Parameter selection for averaging method with cosine distance

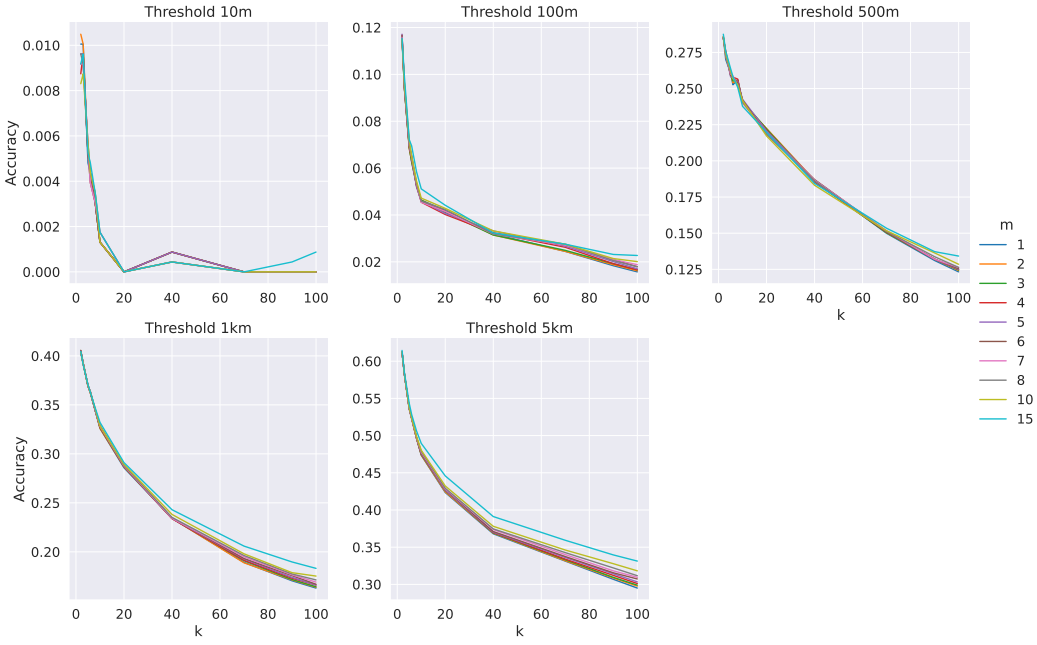


Figure A.3: Parameter selection for weighted averaging method with cosine distance

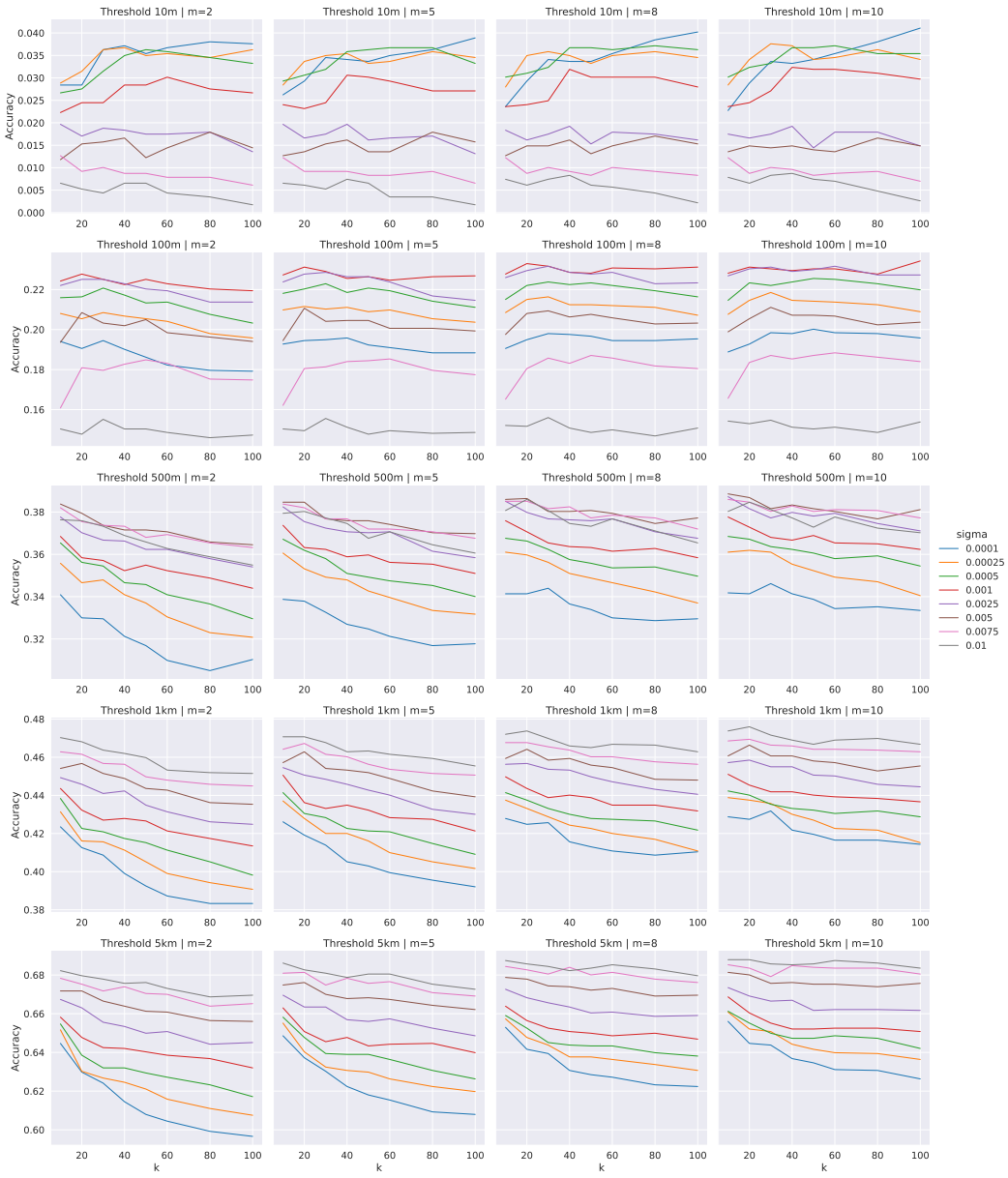


Figure A.4: Parameter selection for KDE with cosine distance

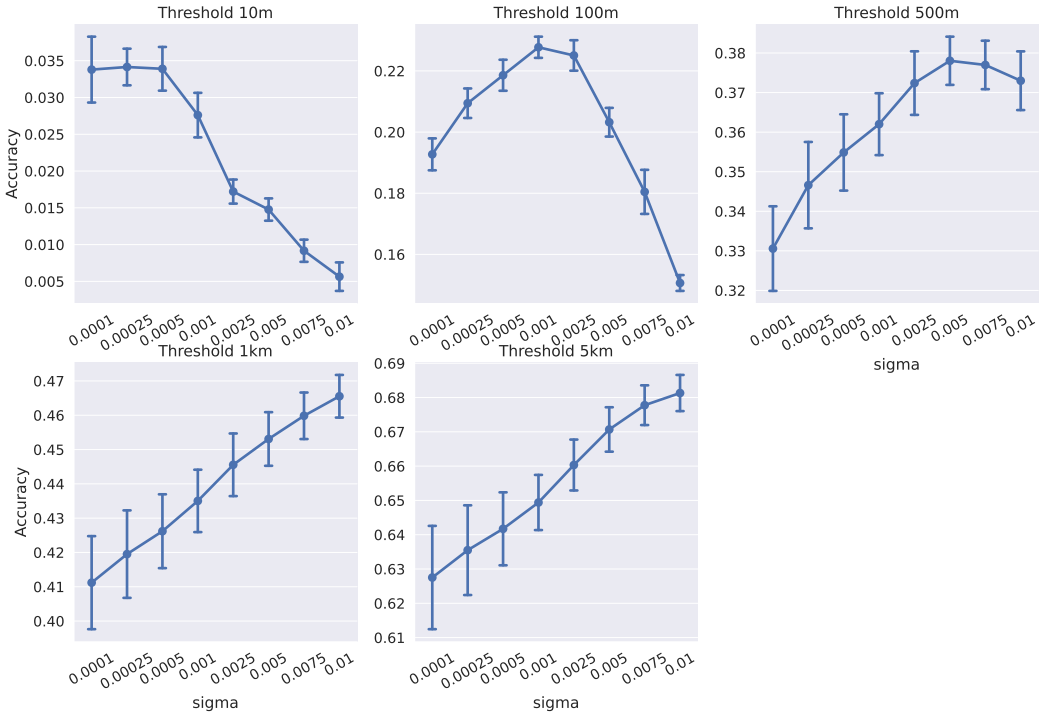


Figure A.5: Average accuracy for different values of σ. KDE with cosine distance.

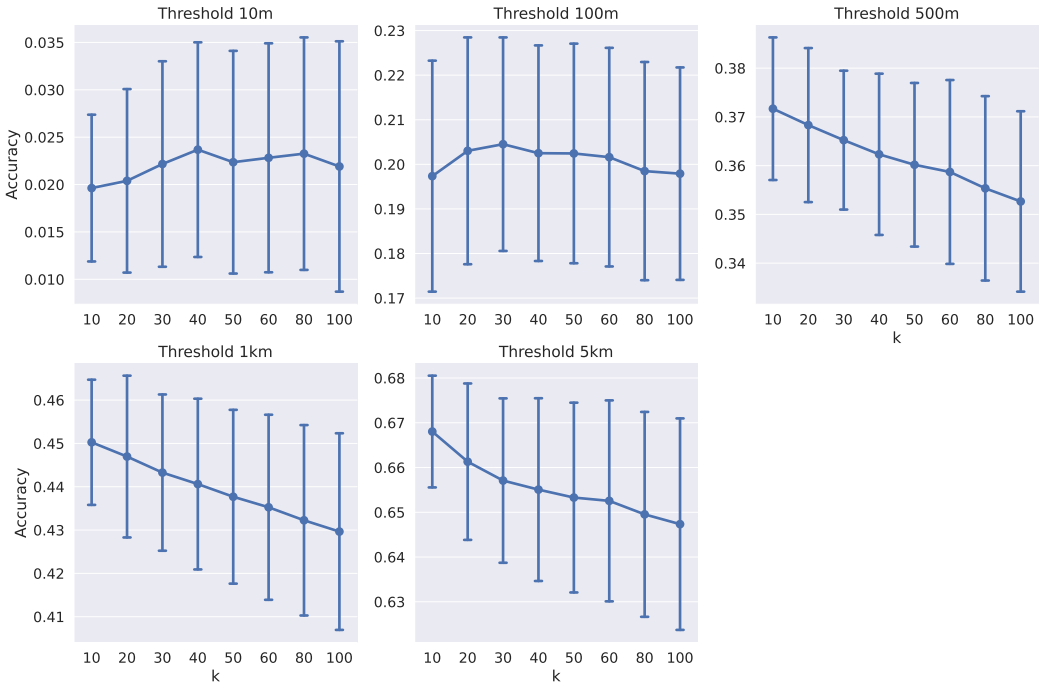


Figure A.6: Average accuracy for different values of k. KDE with cosine distance.

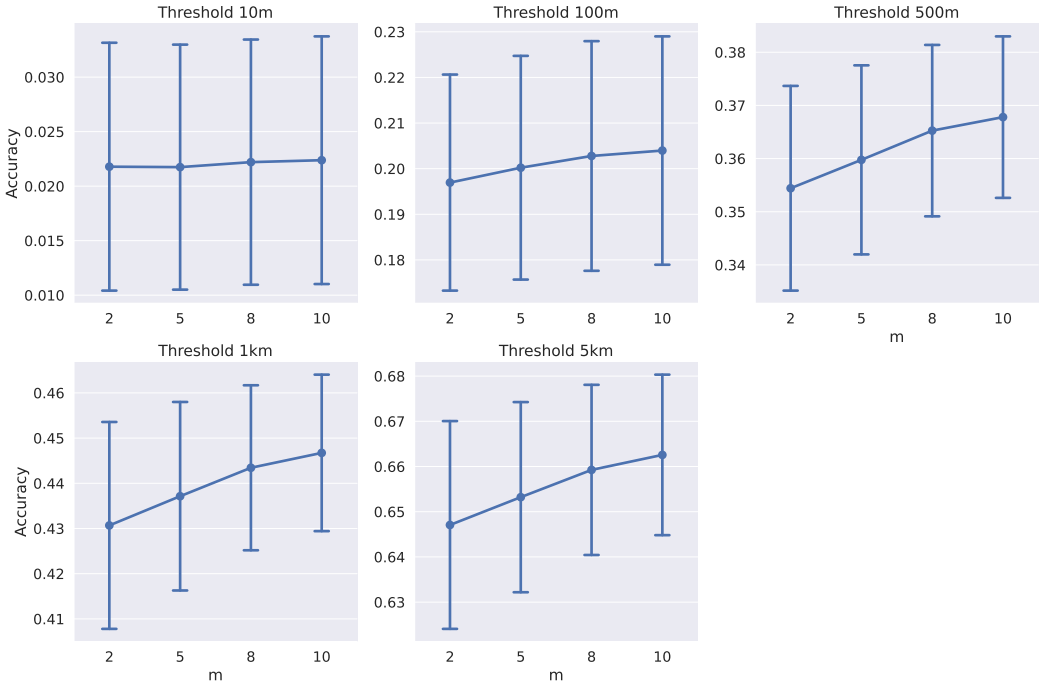


Figure A.7: Average accuracy for different values of m. KDE with cosine distance.

Link to the github repository with implementation https://github.com/ABlack-git/diploma-thesis

Bibliography

[1] M.J. Swain and D.H. Ballard. "Indexing via color histograms". In: *[1990] Proceedings Third International Conference on Computer Vision*. 1990, pp. 390–393. DOI: 10.1109/ICCV.1990.139558.

[2] T. Ojala, M. Pietikainen, and D. Harwood. "Performance evaluation of texture measures with classification based on Kullback discrimination of distributions". In: *Proceedings of 12th International Conference on Pattern Recognition*. Vol. 1. 1994, 582–585 vol.1. DOI: 10.1109/ICPR.1994.576366.

[3] David Lowe. "Object Recognition from Local Scale-Invariant Features". In: *Proceedings of the IEEE International Conference on Computer Vision* 2 (Jan. 2001).

[4] Sivic and Zisserman. "Video Google: a text retrieval approach to object matching in videos". In: *Proceedings Ninth IEEE International Conference on Computer Vision*. 2003, 1470–1477 vol.2. DOI: 10.1109/ICCV.2003.1238663.

[5] N. Dalal and B. Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. 2005, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.

[6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF: Speeded up robust features". In: vol. 3951. July 2006, pp. 404–417. ISBN: 978-3-540-33832-1. DOI: 10.1007/11744023_32.

[7] Alaa Halawani et al. "Fundamentals and Applications of Image Retrieval: An Overview." In: *Datenbank-Spektrum* 18 (Jan. 2006), pp. 14–23.

[8] James Hays and Alexei A. Efros. "IM2GPS: estimating geographic information from a single image". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587784.

[9] Hervé Jégou et al. "Aggregating local descriptors into a compact image representation". In: July 2010, pp. 3304–3311. DOI: 10.1109/CVPR.2010.5540039.

[10] Florent Perronnin et al. "Large-scale image retrieval with compressed Fisher vectors". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2010, pp. 3384–3391. DOI: 10.1109/CVPR.2010.5540009.

[11] Artem Babenko et al. *Neural Codes for Image Retrieval*. 2014. arXiv: 1404.1777 [cs.CV].

[12] Ali Sharif Razavian et al. *CNN Features off-the-shelf: an Astounding Baseline for Recognition*. 2014. arXiv: 1403.6382 [cs.CV].

[13] Artem Babenko and Victor Lempitsky. *Aggregating Deep Convolutional Features for Image Retrieval*. 2015. arXiv: 1510.07493 [cs.CV].

[14] Cornell University. *Neural Networks and Machine Learning*. 2015. URL: http://blogs.cornell.edu/info2040/2015/09/08/neural-networks-and-machine-learning/.

[15] Relja Arandjelović et al. *NetVLAD: CNN architecture for weakly supervised place recognition*. 2016. arXiv: 1511.07247 [cs.CV].

[16] Ali Sharif Razavian et al. *Visual Instance Retrieval with Deep Convolutional Networks*. 2016. arXiv: 1412.6574 [cs.CV].

[17] Tobias Weyand, Ilya Kostrikov, and James Philbin. "PlaNet - Photo Geolocation with Convolutional Neural Networks". In: *CoRR* abs/1602.05314 (2016). arXiv: 1602.05314. URL: http://arxiv.org/abs/1602.05314.

[18] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *arXiv preprint arXiv:1702.08734* (2017).

[19] Filip Radenovic, Giorgos Tolias, and Ondrej Chum. "Fine-tuning CNN Image Retrieval with No Human Annotation". In: *CoRR* abs/1711.02512 (2017). arXiv: 1711.02512. URL: http://arxiv.org/abs/1711.02512.

[20] Nam N. Vo, Nathan Jacobs, and James Hays. "Revisiting IM2GPS in the Deep Learning Era". In: *CoRR* abs/1705.04838 (2017). arXiv: 1705.04838. URL: http://arxiv.org/abs/1705.04838.

[21] Jan Drchal. *Statistical Machine Learning. Lecture 6: Artificial Neural Networks*. 2019. URL: https://cw.fel.cvut.cz/b191/_media/courses/be4m33ssu/anns_ws19.pdf.

[22] Czech Office for Surveying Mapping and Cadastre. *File of Administrative Boundaries and Cadastral Units Boundaries of the CR*. URL: https://geoportal.cuzk.cz/(S(f3gghnfoxhphfxzl05utvssm))/Default.aspx?lng=EN&mode=TextMeta&side=dsady_RUIAN&metadataID=CZ-CUZK-SH-V&mapid=5&menu=252.