POLITECNICO DI TORINO

Master degree in Computer engineering - Data science

Master Thesis

Deep learning for visual place recognition

Building a large scale software to geo-localize a given photo

Supervisors
prof. Barbara Caputo

Gabriele Moreno Berton
matricola: 252805

Academic year 2019 – 2020

Deep learning for visual place recognition
Master thesis. Politecnico di Torino, Turin.

April 2020.

Acknowledgements

Dopo aver concluso la mia tesi, è arrivato il momento più difficile, la stesura dei ringraziamenti.

A mia moglie Daria, che da anni mi supporta e mi sopporta, e che mi aiuta in tutte le piccole e grandi sfide quotidiane.

Alla professoressa Barbara Caputo, che mi ha dato la possibilità di entrare nel mondo del deep learning, e ai ragazzi del lab, che mi hanno fatto capire che si possono imparare mille cose chiacchierando alle macchinette del caffè.

Al mio compagno di tesi Valerio, senza cui non avrei raggiunto i risultati di questa tesi.

Ai miei genitori, senza i quali non sarei mai diventato la persona che sono oggi.

Ai miei amici, che riescono a farmi dimenticare dell'università e del lavoro.

Alle persone che ho incontrato nel cammino della vita, che non ho mai più rivisto, e che hanno saputo insegnarmi qualcosa.

A tutti i giusti, quelli che credono in un mondo migliore, che fanno dei sacrifici per il bene degli altri, e che mi riempiono di motivazione.

A tutti voi, che in mille modi diversi avete fatto in modo che raggiungessi questo traguardo, voglio dire grazie.

Abstract

An open problem in the artificial intelligence community is building an algorithm that is able geo-localize a given photo just by its visual information, overcoming the multiple problems related to the changes in appearance that a specific location has over time. The purpose of this thesis is to show that it is possible to create such a software, which is fast, accurate and which works on large-scale geographical areas. To achieve this it is necessary to overcome various issues, such as creating a vast dataset, its related database, building a reliable neural network and including various machine learning algorithms, in order to create a robust and efficient system. Moreover, we built a graphical user interface to make the system usable by anyone, in which the user can upload the photo and browse through the results. A second part of this thesis is about research, where the focus is on improving the accuracy and speed of the software, and overcoming the various problems that arise when the photos belong to different domains.

Contents

Chapter 1

Introduction

1.1 Thesis's objectives

The goal of this thesis is to build a user-friendly software that, given an outdoor photo as input, is able to understand where it was taken. This task is commonly known as Visual Place Recognition (VPR), and the input image is usually called query. VPR can find many application in today's world, such as 3D reconstruction, augmented reality and outdoor navigation systems, for example for self-driving cars, as GPS signal worsens in cluttered urban environments. While some systems (such as augmented reality) require knowing the 6 degrees-of-freedom from which the photo is taken, the thesis's goal is to compute the location with an acceptable margin of a few meters. Moreover the software must respect the following constraints:

- **large-scale**: be able to find the location of the query within the area of a big city;
- **scalable**: it should easily scale up to bigger geographical areas, such as whole regions or even countries;
- **fast**: the whole computation should be performed in seconds;
- **accurate**: the accuracy should be high enough for the software to be employed in real-world scenarios.

1.2 Related works and main problems

Luckily, the VPR problem is widely studied, with research on the topic steadily growing in the past decade. The task has been approached in many

different ways, but most of the problems are common among all these approaches. The problems of VPR are related to appearance changes, meaning that the same location might look very different in different moments. These appearance changes are related to:

- **lighting conditions**: the huge light variations between day and night, or even between seasons;
- **dynamic objects**: the presence of moving or temporary objects, such as vehicles or pedestrians;
- **long-term conditions**: the changes in the environment during seasons, or even man-made structures;
- **different point of views**: the same place seen from different point of views might look very different;
- **different domains**: with deep learning methods, if train and test set belong to different domains (such as urban and countryside) there might be a drop in performance.

1.3 Our contribution: research & development

We chose to approach the problem using image retrieval techniques, which are based on comparing the query with a large number of images of which the location is known, called database images. To this end, it was needed to build a vast dataset of images, and we chose to collect images from Google Street View. The data collection process is thoroughly explained in Section 3. We then built a dataset with 1 million images of Turin, a smaller subset with 30 thousand images, and a third dataset of images collected with a phone, to test our software. Each of these images is linked to a record in a MySQL database, where their coordinates and other metadata are saved. In Section 4.1 we explain the algorithms used, and we talk about their implementation aimed at high scalability.
Moreover, we present the results of our system, showing that it can quickly and accurately locate a given photo within Turin, being able to find the correct location of 86% of the images with only 20 predictions within seconds (Section 4.2.10.3). We then built a GUI, in order to make the software practical and easy to use, as shown in Section 4.2.10.4.

From this point, our thesis took a path more oriented towards research, by exploring new techniques aimed at improving our results. To this end, we created new datasets (Section 3.6.4), in which we try to overcome some of the basic VPR problems, and we experimented new algorithms (Section 4.1.4), aimed at reducing the loss of accuracy that results from a domain shift. Finally, we built a novel implementation (Section 4.1.3) which helped us achieve better results on our datasets.

Chapter 2

Related Works

2.1 The landscape

Research on visual place recognition has been steadily growing in the last decade. This increase in interest is due to the creation of large geo-localized images datasets, the easiness in acquiring new data (e.g. through camera on smart phone) and the limitations of localization and orientation systems (e.g. GPS signal worsens in urban cluttered environments), which intrinsically have a high degree of approximation. Moreover, an increasing number of present-day practical applications would rely on a ideally perfect visual based localization system, such as 3D reconstruction, consumer photography -"Where did I take these photos?"-, augmented reality, and outdoor or indoor navigation systems, which are essential for self-driving cars and robotics. There is no standardized designation for visual place recognition, methods name vary from one paper to another. The most common ones are visual place recognition, visual based localization, structure-based localization, visual geo-localization, image-based pose estimation, and a number of rearrangements of these terms. Most of current approaches for VPR rely on image retrieval, while others try to exploit 3D methods, and some even view VPR as a classification task.

2.2 Image retrieval

An image retrieval system is a system that, given a dataset of images, called database or gallery, and a single other image, called query, is able to retrieve the images in the database that are most similar to the query. Image retrieval

can have lots of practical applications, one of the most common being face retrieval. In this task, the database is made of images of faces, usually annotated with the person's full name. Then, when a new image of a face is given to the system, this has to be able to understand which person the new image belongs to, by searching the most similar face in the database. This task is similar to the VPR task, where the database is made of images of places, and the queries can be new images uploaded by a user. In the vast majority of cases, an image retrieval system is based on 4 steps:

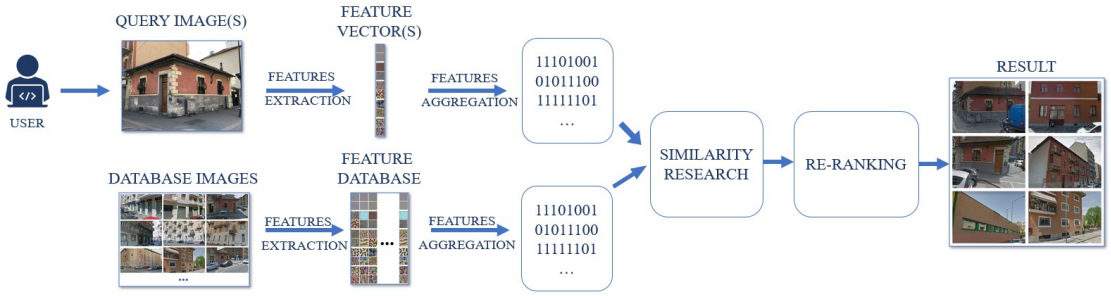


Figure 2.1: Architecture of a standard image retrieval system

- features extraction, which is the extraction of the descriptors that give meaningful information about the image;
- features aggregation, which is the rearrangement of features, preparing them for the next step;
- similarity research, which is the algorithm that takes the features in input and outputs the likelihood of their images representing the same place;
- candidates re-ranking, after having found a restricted number of potential candidates, re-ranking them from the most likely positive.

2.2.1 Features extraction

The task of extracting the best descriptors to represent an image has been approached in a huge variety of methods. The goal is to have features that incorporate the greatest amount of discriminant information, possibly without requiring too much memory, and in a fast and light way. The types of features used belong to one of these 3 groups: local, global, hybrid.

2.2.1.1 Local features

In the pre-deep-learning era, local features were without a doubt the most used in visual place recognition and computer vision in general. Their description occur at a small level, usually just a pixel and its local neighborhood. Its extraction is based on two steps: finding a salient area, and extracting its descriptors. The most famous features-extractor of this kind is the scale-invariant feature transform (SIFT) [1], published and patented in 1999. Various alternatives have been proposed during the years, mostly to speed up the computation, such as SURF [2], which are required when the computation needs to be real-time. Another more recent evolution of SIFT is RootSIFT [3], which creates better descriptors than SIFT while reducing the computational requirements

2.2.1.2 Global features

Global features consider the image as a whole, and compute a vector as its representation. Perhaps the most naive example of global features would be to just use the raw image itself, possibly after resizing. Global features are usually less robust to changes in point-of-view and local changes (such as occlusions), but are usually much less computationally hungry than local features. In this group there are both hand-crafted methods, such as GIST [4], and more recent learned methods. Among the learned methods, the most widely used for images are convolutional neural networks [5, 6], which can be used as a features extractor for a given image. Usually the features are the output from one or more convolutional layer. The CNN can either be trained on a different task, like classification, or directly trained for the image retrieval task.

2.2.1.3 Hybrid features

By hybrid features we refer to those types of features that don't belong to the previous two groups, and that either consider just a part of the image, or that combine multiple types of features. One example is represented by patch features, which, during the extraction, consider only a patch of the image at a time. The patch can be chosen in various ways, either based on the content (e.g. on the image saliency [7]) or in a standardized fashion, like through a fix grid or a sliding window [8]. On the other hand, combined features use a combination of local and global features to build the final descriptors of the image. One example is presented by [9], which uses global features to

restrict the number of potential positives, and local features to compute the final output

2.2.2 Features aggregation

Features aggregation is the task of aggregating features in a more convenient way. These is because the features, especially if local, can have huge dimension. Moreover, in visual place recognition, aggregation can be performed in a way to benefit the retrieval process, for example by enhancing features that are known to be useful for the task.

2.2.2.1 Feature to visual word assignment

This technique comes from the more famous bag of words (BoW) used in natural language processing, where a vector is built counting the occurrences of each word. However, in computer vision there are no such things as words, so it was common to group the features in clusters, or Voronoi cells, and consider only the center of the clusters as features [10]. In this way, a vector with the same length as the number of clusters can be built, with the counts of features belonging to each cluster. This creation of a visual vocabulary is known as bag of visual words or bag of features. However, this hard assignment of each feature to its cluster can worsen the representation of the features, and numerous ways to replace this with a "softer" assignment have been thought. An example is hamming embedding, by Jegou et al. [11], which further divides each cluster for a more precise assignment of every feature. Another subsequent work by Jegou et al. is Vector of Locally Aggregated Descriptors (VLAD) [12], in which also the distance between the feature and the center of its cluster is saved, giving a more precise representation.

2.2.2.2 Weighting scheme

Weighting consists in assigning a weight to each features, usually by giving higher weights to more discriminative features. The weights can be computed by taking into account the frequency of a certain feature in the dataset, as in [10]. Other works [13] propose to assign the weight according to their intra and inter-burstiness, which is the likelihood that a feature is repeated more than once in an image and in the dataset. Other techniques [14] simply propose to remove the least discriminative features, saving memory and reducing noise.

2.2.2.3 Multiple features aggregation

Features obtained with neural networks can be used together with local or patch features. This way its possible to gather multiple types of features into a single vector, like a bag of features. Mixing local and neural features has been done in [15], and [16] has replace local with patch features.

2.2.2.4 Pooling of deep feature maps

With the recent advances of deep learning, it has become more common to extract features with CNNs. The features can be extracted by the last convolutional layers, and can be concatenated together. To reduce the high dimensionality several types of pooling have been applied. Maximum Activations of Convolutions [17] proposes to reduce the dimensionality by aggregating each maximum of the activation maps into unidimensional vectors. Sum-Pooled Convolutional features [18] gives better results, simply summing the responses of each maps instead of finding their maximum. More recently Arandjelovic et al. [5] propose NetVLAD, a fully differentiable layer that can be plugged into a CNN and be treated like another layer of the network. This layer has trainable weights, which make it the state-of-the-art option at the moment. This later propose to mimic the architecture of NetVLAD in a deep-learning fashion, therefore calculating clusters and distances from them for each feature.

2.2.3 Similarity Research

The most common way to find the most similar vector of features to a given one (e.g. computed from a query) is by finding the closest to it by euclidean distance (usually with L2 norm) or cosine similarity. This is simple to do as long as the dimensions don't grow too much, both in terms of number of descriptors (one for each image) and of size of each descriptors. In the latter case, a dimensionality reduction is often needed, and PCA is a common technique [5, 6]. To improve results, whitening can also be applied, as in [5].

2.2.3.1 K-nearest neighbors

Brute force KNN can be used in those cases where the datasets are not too large, and computation can be done in a reasonable time. However, if the size of the dataset grows, so does the number of descriptors, and having an

exact response can take too long to be computed. In some cases, approximate k-nearest neighbors can be a better solution, in which a little loss in accuracy can give a huge reduction of computation time [19]. An example of a library which implements approximate nearest neighbors search is Faiss [20], developed by the Facebook AI Research team.

2.2.3.2 Other Machine Learning methods

Other methods have been employed which focus on better understanding the distribution of the features, and exploit this knowledge to improve the accuracy of the retrieval task. An example is given by [21], where the similarity search is viewed as a classification task, and an SVM is used. In [22] the authors avoid the heaviness of SVM by exploiting the advantages of Linear Discriminant Analysis (LDA).

2.2.4 Candidates re-ranking

The retrieved candidates can optionally undergo a post-processing phase of re-ranking, attempting in this way to sort them in a more accurate way. This is often done when the similarity search is done in an approximate way, and the re-ranking of the selected candidates can be performed in an exact manner. Another example in which re-ranking can be used, is when the features have been reduced by PCA, and the candidates can be sorted by using their distance with the query considering their full dimension, skipping therefore the PCA reduction. Re-ranking is also performed in cases where other data is known about the dataset, such as in [23], where Torii et al. exploit the location of their dataset to further improve the accuracy of the final response.

2.3 Other approaches

A lot of research has tried to approach the VPR problem with various image retrieval techniques that take into account the various problem of VPR, such as domain shift, while others tried to find solutions very different from the standard image retrieval, such as using 3D methods or transforming the localization task into classification.

2.3.1 Cross-appearance localization

Given the huge amount of interest in standard visual place recognition, some researchers also focused their work on solving the problem in different ways. This is the case of cross-domain VPR, where the database and query images belong to slightly different domains, and cross-view VPR, where the domain shift is much bigger.

2.3.1.1 Cross-domain

Cross-domain visual place recognition takes explicitly into account the differences in distribution between the database domain and the query domain. Some examples can be of database images taken as normal daylight photos, while the queries can belong to many different domains, such as photos taken at night, grey-scale images, or even painting (Figure 2.2). An example in this field is [22], in which the query is an old sketch or painting.

Figure 2.2: Cross-domain

Figure 2.3: Cross-view

2.3.1.2 Cross-view

Cross-view VPR focuses on the much more challenging task of having a database made of aerial views images (Figure 2.3). These images have the

advantage to be available for any corner of the globe. However the query images are still taken on from the ground, thus the difficulty of the task. Several works have been done in the field, both with classical [24] and deep methods [25]. Interesting work from Bansal et al. [26] relies on image rectification for ground-level query images, in a way to make them more similar to aerial-view images.

2.3.2 Localization problem as a classification task

Researchers have also approached the VPR task as a classification task, which is, instead of finding the exact position of the query image, to find the region in which it was taken. This allows VPR to take a more global approach, for example by designing a system that predicts in which continent or country the image was taken. To this end, researchers [27] propose a variety of world-wide grids, as in Figure 2.4, and design systems that classify images according to their position in the grid.

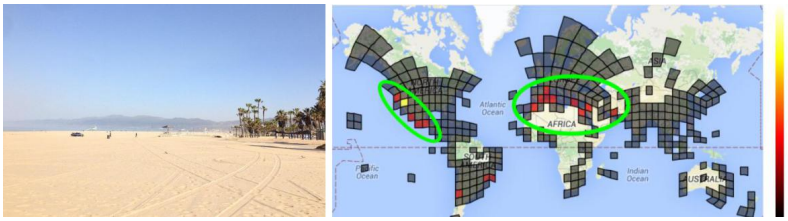


Figure 2.4: Localization as classification

2.3.3 3D-based methods

3D based methods use a database of geo-localized 3D models, which are used to find the location of the 2D query image. The databases can be built using a variety of sensors, such as RGB-D cameras, LIDARs, RADARs, etc., and are usually expensive to acquire, in terms of money cost (3D sensors are usually more expensive than cameras), time, computation and storage needs. These methods are often much slower than 2D-based methods, which makes them less scalable, but they give much higher 6-DOF accuracy [28]. This methods can be used together with 2D-based methods, as in [29], where the authors use the speed of 2D based methods to filter a limited amount of candidates from the database, and then 3D methods to have a precise estimate of the query position.

Figure 2.5: 3D based method from [30]. These methods recover the exact pose of the query. The central image is the query and the surroundings represent the 3D database

Part I

Chapter 3

Data collection

3.1 Building a dataset

After analysing the alternatives, we decided that 2D image retrieval would be the best option for our task. This because 3D datasets are too few and cover just small portion of the territory, and approaching the problem as a classification task, as in [27], would be too imprecise for our needs. Moreover, we decided not to use aerial views, because they usually rely on a stream of images, and their performance with a single image is still very low.
We then briefly explored the various options available in order to obtain a dataset with ground-level street images, that could be scalable and easy to use. The top options were Google Street View, Mapillary, Bing Streetside, Apple Maps, Open Street Cam. All these are websites or apps built on top of massive datasets, usually made of images taken by cars with a camera that drive around cities (Figure 3.1). However, the only ones covering all European cities are Google Street View and Mapillary. Of these two, the latter only has frontal and backward images, while the former has 360° spherical panoramas, which would be perfect for our case of study. Using Street View, we could build datasets covering almost any city of the western world (Figure 3.2).

Figure 3.1: Car used by Google Street View to take panorama images

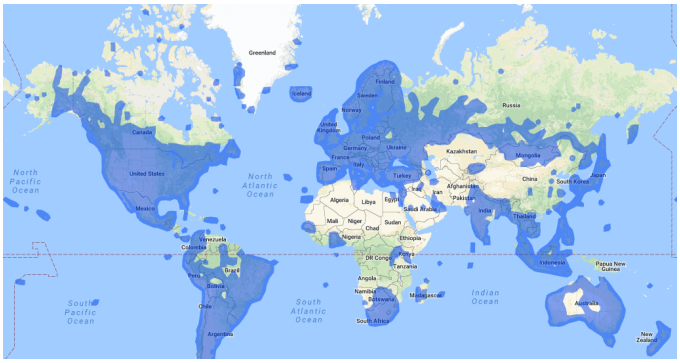


Figure 3.2: The areas in blue have been covered by Google Street View

3.2 Metadata and Google street view panoramas

The Google Street View dataset covers almost all Italian roads, and in most places the photos were taken several times throughout the years. We focused our research in the area of the city of Turin, to limit the amount of images needed for the task. The images provided by Street View are 360° equirectangular panoramas (Figure 3.3), that can be downloaded with various resolution, the highest being 6656x13312 pixels. Using Street View APIs we are able to download some metadata for each panorama, including its ID, latitude, longitude and the date when the image was taken. In this way we are able to build a large database with information about whole cities, and its related dataset of images. We made scripts in order to simplify the download of the data, which take as parameters the coordinates of the area,

and the number of processes that we want to launch, in order to speed up the download by multi-processing. In this way we are able to obtain city-wide metadata in a couple of hours, and their related panoramas in a couple of days, all without requiring high-end computers or incredibly fast internet connection.

Figure 3.3: Example of a Google Street View panorama

3.3 Google Street View Time Machine

A good algorithm for visual place recognition must be invariant to changes in viewpoint and lighting and to moderate occlusions. It should also learn to suppress confusing visual information such as clouds, vehicles and people, and to chose to either ignore vegetation or to learn a season-invariant vegetation representation. The Google Street View Time Machine can help to achieve this, as it can show the same location in different dates. The Time Machine is based on the fact that Google cars pass through the same roads multiple times over the years, and the dataset keeps accumulating images. So every passage of a Google car can be seen as a new layer of panoramas over the previous ones. On the Street View website, one can easily see the same location throughout the years, and using their APIs it is possible to download those panoramas. Figure 3.4 represents various images of the same location between 2008 and 2019. It can be noticed that the images have huge differences, and by using them to train a neural network it can learn to ignore

dynamic objects (such as vehicles), illumination changes, small point of view changes, changes in the vegetation and shadows.

(a) November 2011

(b) June 2012

(c) May 2014

(d) May 2015

(e) July 2016

(f) October 2017

(g) September 2018

(h) August 2019

Figure 3.4: Various images of the same place from 2008 to 2019

3.4 Removing the distortion

As shown in Figure 3.3, panoramas represent a 360° equirectangular projection of the surroundings of the camera. Altough equirectangular projections

are an excellent way to map the surface of a sphere to a flat image, the resulting image appears very distorted, and thus very far from any undistorted image that we would later on use as query. This can represent a challenge for a retrieval neural network, as most visual place recognition networks are designed to handle database images and query images coming from the same domain. Undistorting an equirectangular panorama can be done only for small sections of the image (the field of view has to be less than 180°) by using the gnomonic projection, also known as rectilinear projection. The gnomonic projection is a nonconformal map projection obtained by projecting points P_1 (or P_2) on the surface of sphere from a sphere's center O to point P in a plane that is tangent to a point S. In Figure 3.5, S is the south pole, but can in general be any point on the sphere. Since this projection obviously sends antipodal points P_1 and P_2 to the same point P in the plane, it can only be used to project one hemisphere at a time. In a gnomonic projection, great circles are mapped to straight lines, and it represents the image formed by a spherical lens. In the projection of Figure 3.5, the point

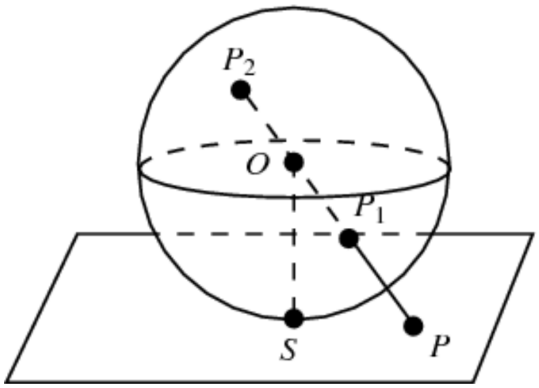


Figure 3.5: Gnomonic Projection

S is taken to have latitude and longitude $(\lambda, \phi) = (0,0)$ and hence lies on the equator. The transformation equations for the plane tangent at the point S having latitude ϕ and longitude λ for a projection with central longitude λ_0 and central latitude ϕ_1 are given by

$$x = \frac{\cos\phi \sin(\lambda - \lambda_0)}{\cos c}$$

$$y = \frac{\cos\phi_1 \sin\phi - \sin\phi_1 \cos\phi \cos(\lambda - \lambda_0)}{\cos c}$$

and c is the angular distance of the point (x, y) from the center of the projection, given by

$$\cos c = \sin \phi_1 \sin \phi + \cos \phi_1 \cos \phi \cos (\lambda - \lambda_0)$$

The inverse transformation equation are

$$\phi = \sin^{-1} \left(\cos c \sin \phi_1 + \frac{y \sin c \cos \phi_1}{\rho} \right)$$

$$\lambda = \lambda_0 + tan^{-1} \left(\frac{x \sin c}{\rho \cos \phi_1 \cos c - y \sin \phi_1 \sin c} \right)$$

where

$$\rho = \sqrt{x^2 + y^2}$$

$$c = tan^{-1} \rho$$

By using the inverse transformation equation we're able to undistort small tiles of the panorama (Figure 3.6), which we can then use for the image retrieval task.

(a) Distorted image (b) Undistorted image

Figure 3.6: Example of a tile of a panorama, distorted (a) and undistorted (b).

3.5 Cleaning the dataset

Even by keeping low resolutions for panoramas, a dataset collected with Google Street View can take huge amount of memory, and its processing will therefore need a long time and computation power. To reduce the size of the dataset, we remove parts of the panoramas that contain the least useful information: the top 4/13, and the bottom 5/13 (Figure 3.7), reducing the size to 4/13 of the original image. This helps to achieve huge memory sav-

Figure 3.7: An example of a panorama, and in the center the part that we save in the dataset

ings, but the dataset can still be of very large dimension. Just to give an example, a dataset of the Turin area, with around 1.000.000 panorama with resolution 512x3584 (removing the top and bottom parts), requires about 300GB of memory. Processing this amount of data can easily take days, and it is therefore necessary to make sure that the our dataset doesn't contain unneeded images. The focus of our research was to retrieve the position of given images which were taken in urban environments. Therefore, all the images containing non-urban scenes can be deleted from the dataset, and this can greatly reduce the amount of memory needed to store the dataset, depending on the area from which the images are downloaded. Obviously, due to the size of the dataset, this process of deleting rural panorama has to be done in an automated way. In order to do this, we decided to use semantic segmentation neural networks to estimate the number of pixels that belong to buildings for each panorama: if this number is below a threshold (we

chose 2.5% of the total amount of pixels as the threshold), it means that the photo was taken in a rural area and can therefore be deleted. The network that we used was a PSPNet [31] with a ResNet50 [32] backbone trained on the ADE20k dataset [33, 34], which showed good qualitative results on our Street View images. This method turned out to work very well for the task, and, when the geographical area selected to download images is big enough (more than 10.000km2), it helps to reduce the size of the dataset of up to 75%. Although the accuracy of PSPNet [31] is very high, its huge computational requirement made it unfeasible to be used of large scale datasets. Our solution was therefore to train a light-weight ResNet18 [32] to give an estimate of the number of pixels belonging to buildings for each panorama, changing therefore a semantic segmentation task to an inference task. To train the ResNet18 [32] we built a dataset of 288.000 panoramas using the PSPNet [31]. After a qualitative analysis we found that the ResNet18 [32] could identify the rural-area images equally well as the PSPNet [31], and was able to do this in a fraction of the time, giving us the possibility to use this method to clean even large scale datasets.

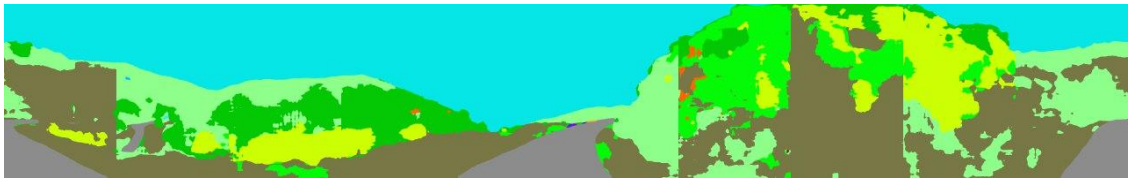
(a) Panorama with buildings

(b) Semantic segmentation of the panorama

(c) Semantic segmentation with buildings highlighted

Figure 3.8: Example of a panorama with many pixel belonging to buildings, and therefore it should be kept in the dataset

(a) Panorama without buildings



(b) Semantic segmentation of the panorama

(c) Semantic segmentation with buildings highlighted

Figure 3.9: Example of a panorama with no pixel belonging to buildings, and therefore it should be deleted

3.6 Datasets

In order to train and test our algorithms we built a few datasets, all of which contain images in the Turin area. For our experiments we used a pre-existing dataset, named Pitts30k, and all the new datasets that we created.

3.6.1 Turin1M

Turin1M is a dataset of 949542 Street View panoramas covering an area of around 100 km^2 between latitude 45.0 and 45.1 and longitude 7.6 and 7.7. This area covers most of the city of Turin, its whole center, and parts of nearby towns (Figure 3.10). The dataset is made of panoramas with resolution 512x3584 where the top and bottom part have been removed, as in Figure 3.7. The whole dataset requires about 300 GB of memory.

3.6.2 Turin30k

When Turin1M dataset is completely downloaded, we have extracted three geographically disjoint subsets (train set, val set and test set) as in Figure

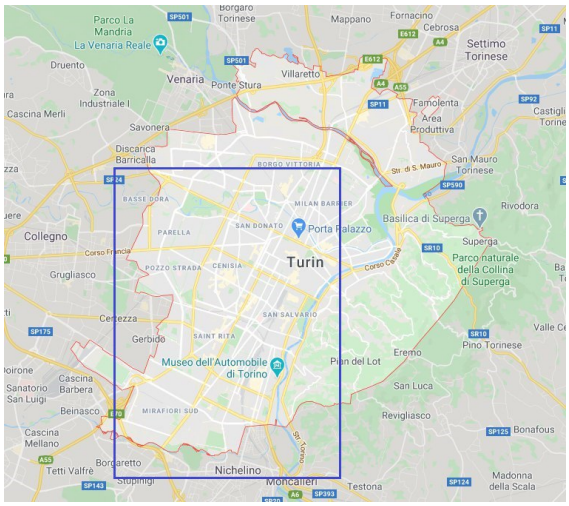


Figure 3.10: The blue rectangle represents the area of Turin1M, while the red boundary is the border of the city of Turin

3.11, in order to fine-tune the network giving a reasonable amount of data and reducing also the retrieval time. First of all, during the training part of the CNN, we have followed what is done by the authors of [5], that consists in use images of the last year (2018/2019) for the query set, and the ones of the same places but of different years for the database set. This is a good approach, because in this way the network receives the image of the same places in different instant, from different view-points and environment conditions. So, it generalizes and learns which features are useful and which are not. Further, the subsets are built providing the number of images inside database equal to the one of the query set. This means that for each query there is exact one positive corresponding database image (e.g. for 10 images of places of the database there are respectively the same 10 places of last year like queries). Since, the network will receive distorted images for the bag-of-visual-words like set and not-distorted images as queries, at test time, we have trained it in order to try to prepare the algorithm to this kind of situation. Then, what is done consists essentially in train using the cropped panos that are downloaded by means of unofficial API, so the distorted images, as set for searching the similar images w.r.t. the queries, while the queries are retrieved using the official API, that returns images with straight lines.
Anyway, we have built three different subset, following the previous rules, that are compose as described in Table 3.1.
In order to consider the whole Turin1M dataset, passing to a large-scale situation, a possible solution is to immediately filter the images that contain

Dataset	Database	Query set
Turin30k_train	10,000	10,000
Turin30k_val	10,000	10,000
Turin30k_test	10,000	10,000

Table 3.1: Number of images in subsets Turin30k.

somethings similar to the queries, discarding all the rests. In this sense, the retrieval area should be reduced together with the needed amount of time. An example could be using such detection/semantic segmentation algorithms to produce annotations applied to filter over the entire dataset. Through this methods, we can retrieve what objects are represented in a photo, discarding all the dynamic ones, detecting texts, vegetation, car models, urban cleanings and so on. In this way, all the images that don't contain the annotation that are extracted by the query, should be ignored at least in a first moment. Overall, we have postponed this kind of approaches to the future works.

Figure 3.11: The three rectangles represents the area selected to build the sets of training, validation and test

3.6.3 Turin81

The main idea of this thesis is to develop a software able to receive images from a third domain, like a mobile phone, and visualize the results produced by the algorithm.
Then, it is not trivial collect a set of geo-localized images from an external device w.r.t. the one of the database. Overall, just to make some indicative tests, we have created a set of 81 images, called Turin81 (Figure 3.13), around the city-centre of Turin captured with an iPhone 7 which has the GPS turned on. In this way, the information about the coordinates are stored in the EXIF metadata attached to the photo. Retrieving a consistent number of this kind of images remains a big problem. Some images of Turin81 are shown in the Figure 3.12.

Figure 3.12: Example of Turin81 images

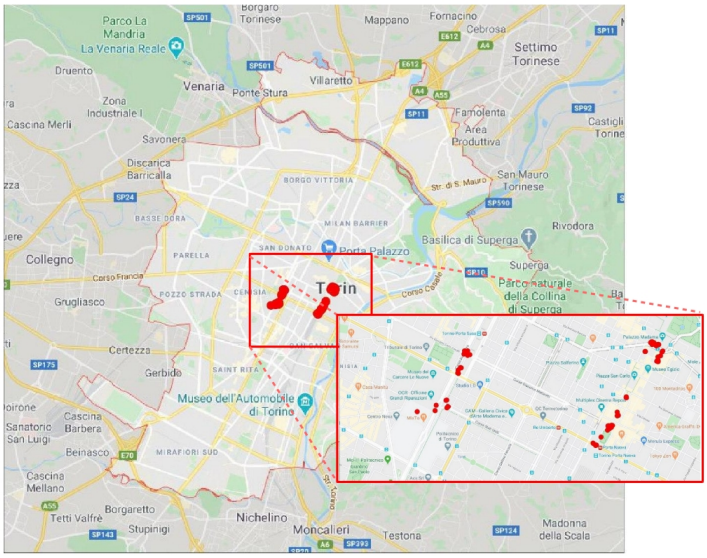


Figure 3.13: The red dots represent the location of the photos in Turin81

3.6.4 Turin30k_undistorted

It contains the same sets of Turin30k, but with the database images that are processed in our algorithm to remove the distortion (Section 3.4). This is used to perform the same tests and evaluate the domain shift given by the distortion in our task.

3.6.5 Turin30k_inpainted

One of the biggest problem of VPR are dynamic objects, like vehicles or pedestrians, that might occlude parts of the images. Moreover, a common image retrieval system, not optimized for the VPR task, might focus its similarity search on these dynamic obejct (e.g. the query has a blue car, then find the database images with blue cars), which obviously doesn't suit well image retrieval for VPR.
To solve this problem, we propose a technique to automatically "erase" these dynamic objects from photos. This is done in two steps: localizing the dynamic objects, and "painting" them over. In order to localize the dynamic objects, semantic segmentation comes to great help, as it gives a good approximation of the pixels that belong to each semantic class. To this end, we used the same PSPNet [31] that we used in 3.5, which showed good results on our datasets. Once the segmentation step was finished, and we found all the dynamic objects in the images, it was time to find a way to erase them.

To this end, a technique called inpainting comes to great help.

3.6.5.1 Inpainting over dynamic objects

Inpainting is a method that, given a masked image, which is an image where some pixels have been removed, tries to find a good estimate of the original values of those pixels. Usually inpainting with deep learning is done with a encoder-decoder architecture, in which the encoder takes as input the masked image and produces features, and the decoder takes as input the features and tries to reconstruct the image removing the hole. Obviously, the network needs to learn useful features from a vast amount of data, in order to predict the values of the missing parts. The inpainting approach that we choose was from the paper Image Inpainting for Irregular Holes Using Partial Convolutions [35], which claims to be the first to demonstrate the efficacy of training image-inpainting models on irregularly shaped holes. A U-Net-like architecture [36] (Figure 3.14 is used, in which each block of the decoder takes as input not only the output of the previous block, but also the output of the corresponding block of the encoder. One of the main challenges in

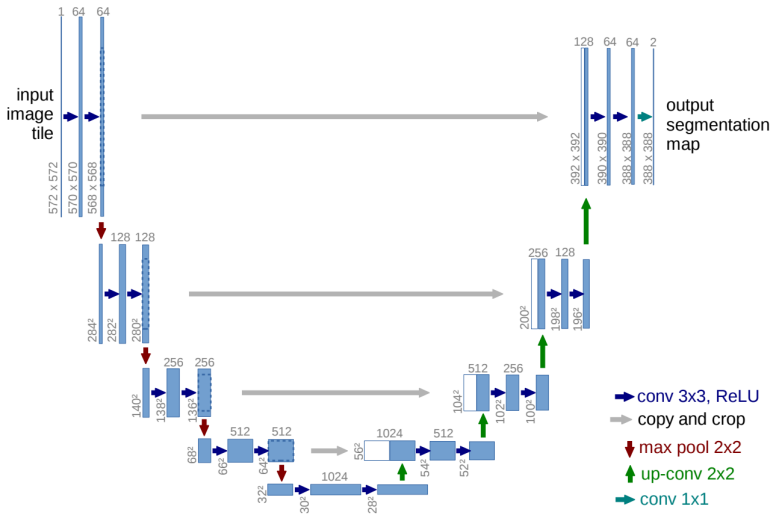


Figure 3.14: U-Net architecture

inpainting is to find a suitable loss. A simple approach would be to use an L1 or L2 loss, calculating the pixels-wise distance between the ground-truth image, and the image outputted by the network. Unfortunately, minimizing the L1 or L2 loss does not produce realistic results, as the output is usually very blurry.

To solve this problem [35] proposes to use multiple losses:

- $$L_{hole} = \frac{1}{N_{i_{gt}}} ||(1 - M) \odot (I_{out} - I_{gt}||_1)$$

  where M is the binary mask, I_{out} is the prediction, I_{gt} is the ground truth image, $N_{i_{gt}}$ denotes the number of elements in I_{gt} (C*H*W)

- $$L_{valid} = \frac{1}{N_{i_{gt}}} ||M \odot (I_{out} - I_{gt})||_1$$

- $$L_{perceptual} = \sum_{p=0}^{P-1} \frac{||\Psi_p^{I_{out}} - \Psi_p^{I_{gt}}||_1}{N_{\Psi_p^{I_{gt}}}} + \sum_{p=0}^{P-1} \frac{||\Psi_p^{I_{comp}} - \Psi_p^{I_{gt}}||_1}{N_{\Psi_p^{I_{gt}}}}$$

  where $\Psi_p^{I_*}$ is the output of I_* from the pth layer of the VGG-16 [37] pre-trained on ImageNet (the layers utilized are pool1, pool2, pool3), I_{comp} is the raw output image $Iout$, but with the non-hole pixels directly set to ground truth, $N_{\Psi_p^{I_{gt}}}$ is the number of elements in $\Psi_p^{I_{gt}}$

- $$L_{style_{out}} = \sum_{p=0}^{P-1} \frac{1}{C_p C_p} ||K_p((\Psi_p^{I_{out}})^T(\Psi_p^{I_{out}}) - (\Psi_p^{I_{gt}})^T(\Psi_p^{I_{gt}})$$

  $$L_{style_{comp}} = \sum_{p=0}^{P-1} \frac{1}{C_p C_p} ||K_p((\Psi_p^{I_{comp}})^T(\Psi_p^{I_{comp}}) - (\Psi_p^{I_{gt}})^T(\Psi_p^{I_{gt}})$$

- $$L_{total-variation} = \sum_{(i,j)\in R,(i,j+1)\in R} \frac{||I_{comp}^{i,j+1} - I_{comp}^{i,j}||_1}{N_{I_{comp}}} + \sum_{(i,j)\in R,(i,j+1)\in R} \frac{||I_{comp}^{i+1,j} - I_{comp}^{i,j}||_1}{N_{I_{comp}}}$$

  where R is the region of 1-pixel dilation of the region.

The total loss is therefore computed as

$$L_{total} = L_{valid} + 6L_{hole} + 0.005L_{perceptual} + 120(L_{style_{out}} + L_{style_{comp}}) + 0.1L_{total-variation}$$

The final total loss makes the output images accurate enough to be indistinguishable from the ground truth for small masks. However, in our case the masks often have bigger size, and the output can be less realistic, but hopefully good enough to improve the performances of the network. As shown in Figure 3.15, this technique shows pretty good results on our dataset. The results this dataset are reported in 4.2.6.

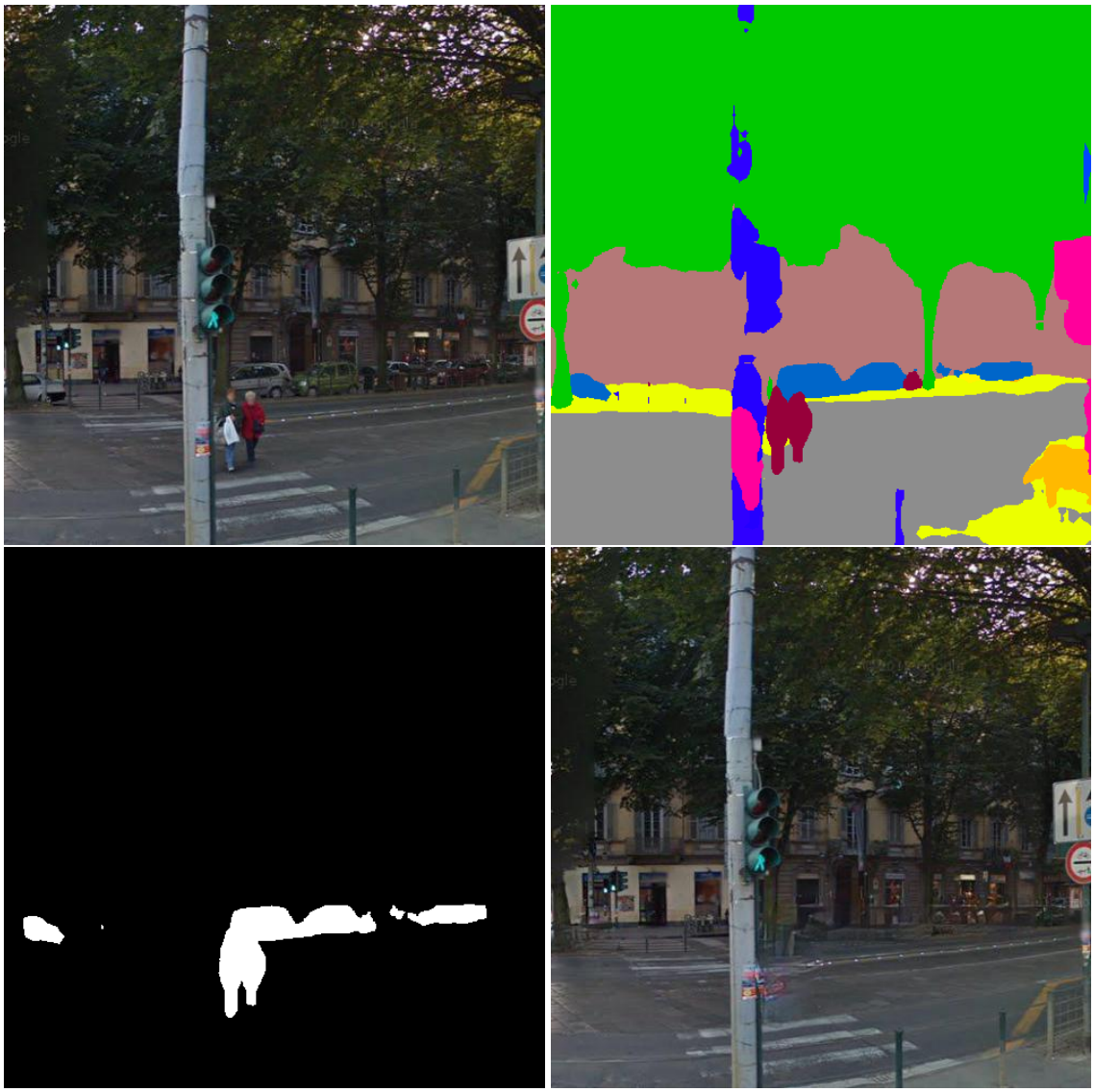


Figure 3.15: Example of the pipeline to create inpainted datasets on Turin30k: on the top-left the input image, top-right the segmented image, bottom-left the mask of dynamic objects, bottom-right the inpainted image

3.6.6 Pitts30k

Pitts30k is a pre-existing dataset created by the authors of NetVLAD [5] in order to test their algorithm. We also used this dataset to compare our results to theirs. Pitts30k is a dataset of 30000 database images and 21840 query images, divided in train, val e test set. The images are taken in the city of Pittsburgh, Pennsylvania. Just like our Turin30k dataset, Pitts30k

has been built with Google Street View images.

3.6.7 Pitts30k_inpainted

For the sake of completeness, we decided to apply to Pitts30k the same segmentation plus inpainting that we used on Turin30k (Figure 3.16. The results this dataset are reported in 4.2.5.

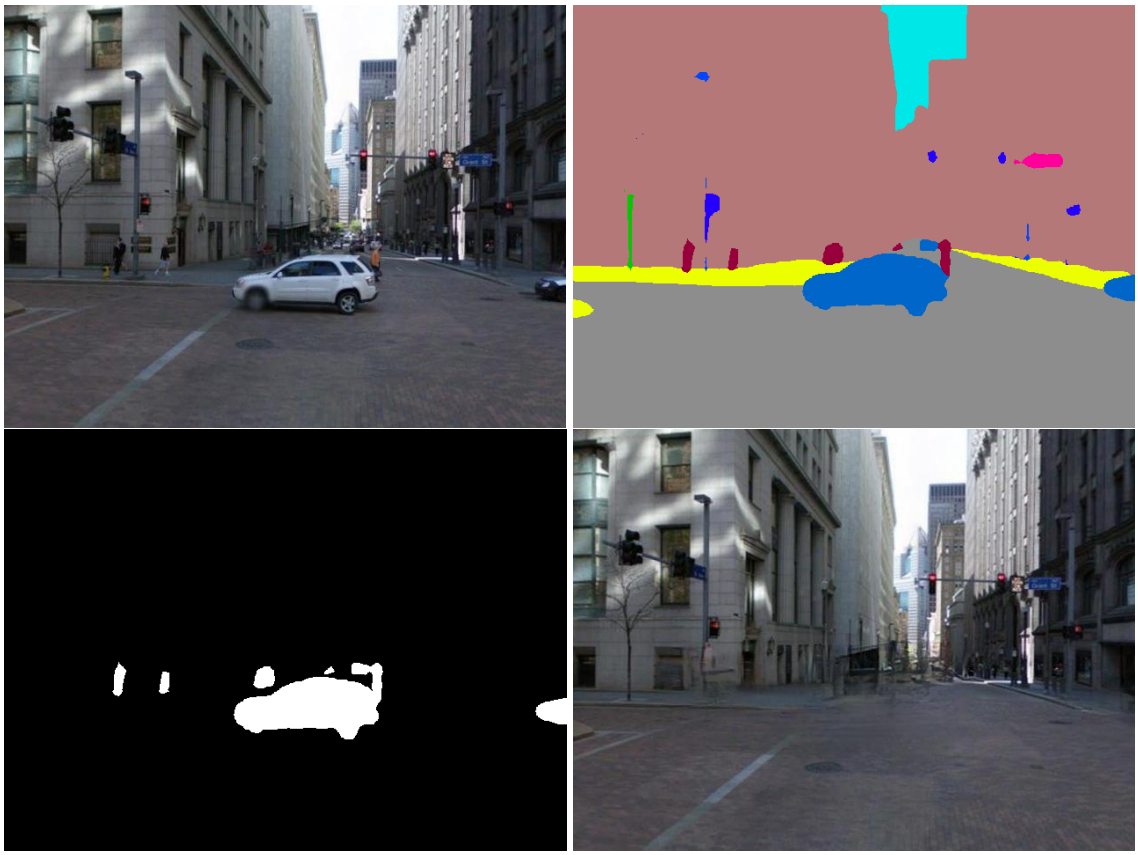


Figure 3.16: Example of the pipeline to create inpainted datasets on Pitts30k: on the top-left the input image, top-right the segmented image, bottom-left the mask of dynamic objects, bottom-right the inpainted image

Part II

Chapter 4

Architectures and Experiments

4.1 Architectures

To perform the tests on the datasets that we created we used a number of architectures, starting from current state-of-the-art NetVLAD.

4.1.1 NetVLAD

Today, the state-of-the-art is achieved by NetVLAD [5] network, that is properly developed for the VPR task solved via image retrieval and is inspired by the VLAD representation [38]. The network is composed by a backbone, which is the VGG16 [37] for the best result, or AlexNet [39]. It is truncated at the last convolutional layer (conv5), where the features related to the entire image are extracted. At this point, a novel trainable VLAD layer is used to compact the features in a fixed length vector representation. The architecture is shown in the Figure 4.1. In [5] the Triplet Loss (Figure 4.2) is

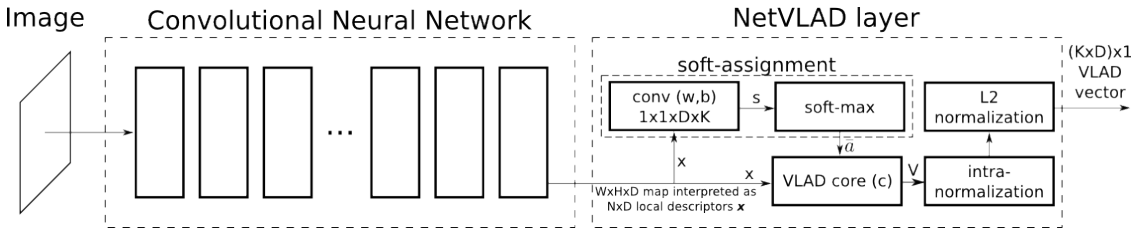


Figure 4.1: NetVLAD architecture [5]

used, in a revisited form properly for the VPR task. They [5] call their loss 'Weakly supervised triplet ranking loss', that we can summarize in this way:

- Take a test query q;
- Between all its potential positives p_i^q, take the best matching potential positive $p_{i^*}^q = \underset{p_i^q}{\operatorname{argmin}}\, d_\theta(q, p_i^q)$;
- Take its negative samples $\{n_j^q\}$;
- Compute the distance between the query and positive $d(q, \{p_{i^*}^q\})$ and the ones between the query and all the negatives $d(q, \{n_j^q\})$;
- The goal becomes: $d(q, \{p_{i^*}^q\}) < d(q, \{n_j^q\})\ \forall j$, so it's a ranking loss between each training triplet $(q, \{p_i^q\}, \{n_j^q\})$;
- So, their [5] loss is defined in this way:

$$L_\theta = \sum_j l\left(\min_i d_\theta^2(q, p_i^q) + m - d_\theta^2(q, n_j^q)\right)$$

  where l is the hinge loss and m is a constant figuring the margin. In this way all the negatives that have distance greater by a margin than the distance between the best matching potential positive and the query, will have loss 0. While when the margin is violated, the loss will be proportional to the amount of violation.

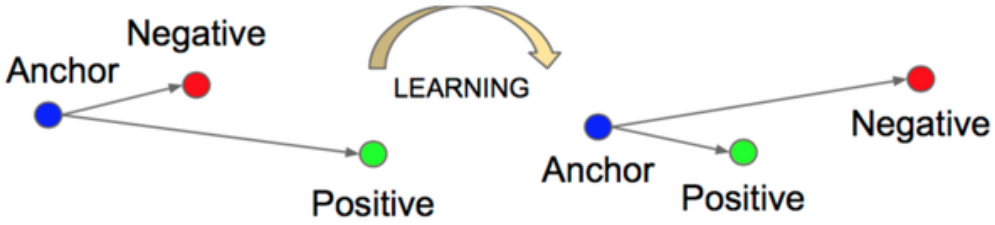


Figure 4.2: Triplet Loss: minimize the distance between the anchor and the positive sample, while maximize the distance between the same anchor and the negative sample.

Overall, the best advantage is related to the fact that NetVLAD [5] is end-to-end trainable and is developed for the specific VPR task, so it is very robust on localizing urban photos.

4.1.1.1 Training process

In order for the triplet loss to be performed, it is necessary to compute the nearest (in the features space) positive and the 10 nearest negatives for each query. Obviously finding the exact nearest positive and negatives would require to compute the features for all database images at each iteration, making the training process incredibly slow. To speed up the training, the authors proposed to compute a cache of all the features, to be used for a number of iterations, and then be recomputed again. This number is called cache refresh rate, and it's best value is between 1000 and 500.
The code that we used starts from a version downloaded from a GitHub repository [40], that contains a pytorch version of NetVLAD [5] that was originally developed in Matlab. This code receives a set of arguments in order to run the algorithm for training, testing or clustering. Furthermore, from command line, it's possible also decide the pre-trained network to use, the number of epochs to train, which dataset to test, the path to resume a checkpoint and many others settings.
To train completely the network, two steps are needed:

- Run the algorithm in clustering mode:
  - Build model: it is compose by the pre-trained backbone (VGG16 [37] or AlexNet [39]) and a L2 normalization layer;
  - Take the dataset used to train: in this phase only the database set is used;
  - Create a dataloader sampling randomly from the database set and extract the descriptors;
  - Perform a K-means algorithm to cluster the extracted descriptors for a given K (from NetVLAD [5] paper K=64);
  - Store the cluster centroids.
- Run the algorithm in train mode:
  - Build model: the network is composed by the encoder (same backbone as before) and the pooling layer NetVLAD [5], that is initialized with the centroids previously extracted;
  - Take the dataset used to train: there are two classes for the dataset, the first one is for the database images, it contains the path for the images, how many database and query images there are, the coordinates and some others information. While the one for the queries,

contains all the stuff so the code to analyze each query and find its positive and negatives. This research is performed in two steps, the first one is a KNN with radius, used to find the positives images inside the 25 meters from the query coordinate, and the second step is a KNN based on the features, to keep the most positive between the ones previously found. This last KNN is executed inside the inherit ___getitem___ function of the dataset class, where the features of the database images are already passed. For the negative samples instead, the code considers all the samples out of 25 meters and keep randomly 1000 samples. Then, a KNN based on the query and negatives features is computed, extracting only 10 negative samples that are within margin w.r.t. the positive.

- Start the training: for a certain number of epochs it splits the queries set in subsets, for each of them the algorithm extracts the descriptors of the database saving that results in a cache variable. The dimension of the cache is given by the argument cacheBatchSize (default value is 24). The use and the advantages of the cache is explained in [5]. At this point, the dataloader of the considered subset of queries is created. The dataloader of queries set returns tensors corresponding to the query, its positive and its negatives. Then, the Triplet Loss is calculated and the backpropagation algorithm is performed. All this stuff is repeated for all the subsets of queries set.
- Each time an epoch ends, an evaluation is performed on a specified set (val set by default). A checkpoint is saved for each epoch and whether the accuracy is improved a further checkpoint called 'model_best' is stored.
- The metrics used to evaluate the algorithm is the recall@N. It consists in give us how many predictions are correct after N ones. The value used to compare an epoch and evaluate if is the best one is the recall@5.

The training can be also restored from a checkpoint. In this case the option 'resume=path_of_checkpoint' should be defined when the program is launched. Then, all the flags, the optimizer parameters and so on, are resumed from the checkpoint.

- Run the algorithm in test mode:
  - Specify the checkpoint path in the resume option;

- The model is built and the weights of the resumed checkpoint are loaded;
- The specified dataset is loaded, both database and queries set. There is only one dataloader, because the samples are all collected in an sorted list, then the counter of database and queries images, give us the split of that two sets.
- The batches pass through the network extracting the descriptors and search between the descriptor vector is executed. To improve the efficiency of this step, the faiss [20] python library is used.
- The recall@N with N = [1, 5, 10, 20] is displayed.

4.1.1.2 Setup

All our trainings on NetVLAD have been done with the following configurations:

- Backbone: VGG16 [37] (pre-trained on ImageNet) cropped at the last convolutional layer;
- $Loss_{NetVLAD}$: Triplet ranking loss [5];
- Trainable layers: whole backbone + netvlad core;
- Number of clusters K: 64;
- Optimizer: SGD;
- Learning rate: 0.0001;
- Momentum: 0.9;
- Weight decay: 0.001;
- Scheduler step: 5 epochs;
- Scheduler decay: 0.1;
- Batch size: 4 tuples (each tuple contains the query, the positive and at most 10 negatives);
- Cache refresh rate: 1000.

4.1.2 ResNetVLAD

The complexity of NetVLAD, in addition to the burden of calculating the positives and negatives for each iteration, makes it a very heavy network to train. Training NetVLAD can easily take days, even on the latest GPUs. This is also due to the VGG16 architecture, which was created in 2014, and, given its huge number of parameters (138M), its slow to train. More recent CNNs, like the popular Resnet [32] in all its variants, can achieve similar or better results while requiring a fraction of the time to train. For the aforementioned reasons, we decided to substitute the backbone of NetVLAD with a Resnet18, which achieves similar results to the VGG16 on the ImageNet challenge. However, instead of taking features from the last convolutional block, as with the VGG16, we decided to extract them from the one before the last. This helped us to speed up the training time, from few days to few hours, giving us the possibility to perform a greater number of tests.

4.1.2.1 Setup

All our trainings on ResNetVLAD have been done with the following configurations:

- Backbone: ResNet18 [32] (pre-trained on ImageNet [41]) cropped at the 4th convolutional block, and freezing all layers before the last one;
- $Loss_{NetVLAD}$: Triplet ranking loss [5];
- Trainable layers: last res-block layer of ResNet18 + pooling;
- Number of clusters K: 64;
- Optimizer: Adam;
- Learning rate: 0.00001;
- Scheduler: no;
- Batch size: 4 tuples (each tuple contains the query, the positive and at most 10 negatives);
- Cache refresh rate: 1000.

4.1.3 DyNetVLAD

Although NetVLAD is end-to-end trainable, preliminary tests showed that the clusters's initialization has a great impact on the final results of the trained neural network. To exploit this information, we built a novel implementation, where the architecture is the same as in the original paper, but the clusters are not only learned, but also recomputed dynamically along the training, from which the name DyNetVLAD. The training process differs from NetVLAD in that once convergence is reached, instead of ending the training process, new clusters are computed, and the NetVLAD core parameters are reset from scratch. The new clusters are computed from the weights of the backbone, which in the meantime has adjusted its weights, and the backbone's weights are kept once the new clusters are computed. This method showed better results on both datasets that we used for training and testing, by better adapting to them, even though the training time is usually much longer than in NetVLAD. Preliminary tests with a similar approach, in which the clusters are recomputed after any epoch which shows improvement (instead of after reaching convergence) showed worse results.

4.1.3.1 Setup

All our trainings on DyNetVLAD have been done with the following configurations:

- Backbone: ResNet18 [32] (pre-trained on ImageNet [41]) cropped at the 4th convolutional block, and freezing all layers before the last one;
- $Loss_{NetVLAD}$: Triplet ranking loss [5];
- Trainable layers: last res-block layer of ResNet18 + pooling;
- Number of clusters K: 64;
- Optimizer: Adam;
- Learning rate: 0.00001;
- Scheduler: no;
- Batch size: 4 tuples (each tuple contains the query, the positive and at most 10 negatives);
- Cache refresh rate: 1000.

4.1.4 ResNetVLAD + autoencoder

One big problem in our project is that the query images (taken with a phone) and the database images belong to different domains. Moreover we have few query images, so we cannot use them to train the model. Furthermore in this experiments we have used distorted images in the database to compare the result with the ones obtained in the previous experiment, especially the one over Turin1M. So, the idea is to inject more information as possible to the network about our query test dataset, without feeding it any human-annotated label. Summarizing, the focus on the very important features (e.g. the ones of the buildings) is already partly done as in NetVLAD [5], using the Google Time Machine, then we use the distorted images as database set also during the training and finally to learn more about this features, a self-supervised task is used. When the number of test query images will be higher, a domain adaptation process will be performed directly using those images.
Self-supervised learning is a technique in which the labels that are used to train the neural networks are automatically generated in the code, and take therefore the name of pseudo-label. This makes creating labels a trivial task, which, unlike human-annotated labels, does not require a long annotating process. The easiest example of self-supervised learning is perhaps the rotation task, where an unlabeled image is used, rotated of a random angle multiple of 90°, and then the network has to predict the correct rotation of the image. This process helps the network to learn features which are useful to the self-supervised task, and most likely they will be useful also for the main task. Moreover self-supervised learning is used for domain adaptation, where a network is trained to solve the main task on a domain, and a self-supervised task on another domain, on which we will at the end test the network.
To this end, we added a self-supervised branch to the main ResNetVLAD architecture. There are multiple options available for choosing a self-supervised task, and we opted for an autoencoder. An autoencoder is a network whose task is to output the image, as it was given in input. A standard autoencoder is made by an encoder, which transforms the image into features, and a decoder, which transforms the features into an image. The loss used in our case is L2, which during preliminary tests showed better results than L1 in reconstructing the input image.

4.1.4.1 Setup

The training of ResNetVLAD + autoencoder has been done with the following configurations:

- Encoder for NetVLAD: ResNet18 [37] (pre-trained on ImageNet) cropped at the last convolutional layer;
- Encoder for autoencoder: same as encoder for NetVLAD, plus the last convolutional layer;
- Decoder for autoencoder: equivalent layers as ResNet18 inverted (scaling up instead of scaling down);
- $Loss_{NetVLAD}$: Triplet ranking loss [5];
- $Loss_{autoencoder}$: L2 [5];
- Trainable layers: whole encoder + NetVLAD core + whole decoder;
- Number of clusters K: 64;
- Optimizer: Adam;
- Learning rate: 0.00001;
- Scheduler: no;
- $Batch_sizeNetVLAD$: 4 tuples (each tuple contains the query, the positive and at most 10 negatives);
- $Batch_sizeautoencoder$: 16;
- Cache refresh rate: 1000.

4.2 Experiments

All the experiments report a table, in which is shown the recall of the algorithm on the val and test dataset. When testing a model, for each query we retrieve the 20 database images that are the nearest in the features space to the query. These 20 images are sorted, and based on them we calculate the recall. R@1 (standing for recall at 1) is the percentage of query for which the first database image retrieved was correct. R@5 is equivalent, but considering the first five images, and so on. R_avg is the average between R@1, R@5,

R@10, R@20. For each experiment it's shown on which dataset the model was trained and tested. The algorithms are trained and tested on computer with:

- GPU: NVIDIA GeForce GTX Titan X 12GB
- RAM: 64GB
- CPU(s): 12 x Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz

4.2.1 NetVLAD on Pitts30k

We started our experiments by trying our pytorch NetVLAD model on Pitts30k, as its authors did, to make sure that we could get the same results as theirs. Our tests showed (Table 4.1) that our configurations were indeed correct, as we could obtain the same results as the paper.
Setup:

- Trained on: Pitts30k, train set
- Tested on: Pitts30k, val and test set
- Reached convergence after 25 epochs, in 37h30m

Dataset	R@1	R@5	R@10	R@20	R_avg
Pitts30k Val	85.2	94.6	97.0	98.3	93.8
Pitts30k Test	81.9	91.2	93.2	95.7	90.5

Table 4.1: Results of NetVLAD trained on Pitts30k

4.2.2 ResNetVLAD on Pitts30k

We trained our ResNetVLAD model on Pitts30k, to make sure that it would be faster and as accurate. Our tests confirmed this hypothesis. Moreover, the results on the Turin30k dataset shows that there is a considerable domain shift between the two datasets.
Setup:

- Trained on: Pitts30k, train set
- Tested on: Pitts30k, val and test set; Turin30k, val and test set

- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 3 epochs, in 2h18m

Dataset	R@1	R@5	R@10	R@20	R_avg
Pitts30k Val	87.5	95.3	96.9	98.1	94.4
Pitts30k Test	85.5	92.4	94.3	95.9	92.0
Turin30k Val	54.5	76.1	83.4	88.9	75.7
Turin30k Test	54.6	73.6	80.0	85.8	73.5

Table 4.2: Results of ResNetVLAD trained on Pitts30k

4.2.3 ResNetVLAD on Turin30k

Aware of the fact that there must be a reasonable domain shift between the datasets of Pitts30k and Turin30k, we trained our model on Turin30k. The results are still lower w.r.t. those on Pitts30k, meaning that Turin30k is a more challenging dataset. The training time on this dataset is also considerably longer, meaning that the features needed to it are more distant to the pre-existing ones, of the pre-training on ImageNet.
Setup:

- Trained on: Turin30k, train set
- Tested on: Turin30k, val and test set; Pitts30k, val and test set
- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 7 epochs, in 7h44m

Dataset	R@1	R@5	R@10	R@20	R_avg
Turin30k Val	76.0	90.7	94.2	96.5	89.3
Turin30k Test	76.7	91.3	94.8	97.0	89.9
Pitts30k Val	67.4	80.5	84.3	87.6	79.9
Pitts30k Test	73.8	85.1	88.9	92.2	85.0

Table 4.3: Results of ResNetVLAD trained on Turin30k

4.2.4 ResNetVLAD on Turin30k_undistorted

To check if the distortion would actually be one of the causes for the lower performances of ResNetVLAD on Turin30k compared to Pitts30k, we had to train and test the model on the undistorted dataset. Our results show that removing the distortion can bring a slight improvement over the distorted dataset. Moreover, the ResNetVLAD trained on Turin30k_undistorted gives higher results when tested on Pitts30k rather than ResNetVLAD trained on Turin30k, meaning that the domain gap between Pitts30k and Turin30k_undistorted is smaller than the one between Pitts30k and Turin30k.
Setup:

- Trained on: Turin30k, train set
- Tested on: Turin30k, val and test set; Pitts30k, val and test set
- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 9 epochs, in 9h28m

Dataset	R@1	R@5	R@10	R@20	R_avg
Turin30k_undistorted Val	77.0	91.2	94.3	96.4	89.7
Turin30k_undistorted Test	77.4	91.5	94.9	97.1	90.2
Turin30k Val	74.5	90.5	94.0	96.3	88.9
Turin30k Test	75.4	90.6	94.4	96.8	89.3

Table 4.4: Results of ResNetVLAD trained on Turin30k_undistorted

4.2.5 ResNetVLAD on Pitts30k_inpainted

The Pitts30k_inpainted dataset has exactly the same images of Pitts30k, to which segmentation and inpainting have been applied. Our results on this dataset showed a slight but constant improvement, meaning that inpainting does indeed have a positive effect on this dataset.
Setup:

- Trained on: Pitts30k_inpainted, train set
- Tested on: Pitts30k_inpainted, val and test set
- The recalls, epochs and times are an average over 3 runs

- Reached convergence after 2 epochs, in 2h00m

Dataset	R@1	R@5	R@10	R@20	R_avg
Pitts30k_inpainted Val	87.8	95.4	97.1	98.2	94.6
Pitts30k_inpainted Test	86.1	92.6	94.5	95.9	92.3

Table 4.5: Results of ResNetVLAD trained on Pitts30k_inpainted

4.2.6 ResNetVLAD on Turin30k_inpainted

The Turin30k_inpainted dataset has exactly the same images of Turin30k, to which segmentation and inpainting have been applied. Our results on this dataset do not show any improvement.

- Trained on: Turin30k_inpainted, train set
- Tested on: Turin30k_inpainted, val and test set
- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 7 epochs, in 7h14m

Dataset	R@1	R@5	R@10	R@20	R_avg
Turin30k_inpainted Val	76.0	90.7	94.3	96.5	89.4
Turin30k_inpainted Test	76.7	90.8	94.5	96.8	89.7

Table 4.6: Results of ResNetVLAD trained on Turin30k_inpainted

4.2.7 DyNetVLAD on Pitts30k

We then trained our novel implementation on the Pitts30k dataset, to see if it could achieve reasonable improvements. The results clearly show a considerable improvement.
Setup:

- Trained on: Pitts30k, train set
- Tested on: Pitts30k, val and test set; Turin30k, val and test set

- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 5 epochs, in 3h54m

Dataset	R@1	R@5	R@10	R@20	R_avg
Pitts30k Val	88.1	95.6	97.4	98.4	94.9
Pitts30k Test	86.4	93.1	94.9	96.4	92.7
Turin30k Val	57.0	77.7	84.6	89.9	77.3
Turin30k Test	58.3	76.9	82.9	88.2	76.6

Table 4.7: Results of DyNetVLAD trained on Pitts30k

4.2.8 DyNetVLAD on Turin30k

We then trained DyNetVLAD also on the Turin30k dataset, to see if it could achieve reasonable improvements also on this dataset. The results clearly show a considerable improvement.
Setup:

- Trained on: Turin30k, train set
- Tested on: Turin30k, val and test set; Pitts30k, val and test set
- The recalls, epochs and times are an average over 3 runs
- Reached convergence after 20 epochs, in 21h56m

Dataset	R@1	R@5	R@10	R@20	R_avg
Turin30k Val	77.4	91.6	94.9	97.0	90.2
Turin30k Test	78.2	92.1	95.6	97.6	90.9
Pitts30k Val	68.8	81.4	85.4	88.5	81.1
Pitts30k Test	72.7	84.2	87.8	91.1	83.9

Table 4.8: Results of DyNetVLAD trained on Turin30k

4.2.9 ResNetVLAD + autoencoder on Pitts30k

We trained and tested the ResNetVLAD with autoencoder on the standard Pitts30k, to have results that we could compare to the standard NetVLAD [5] paper. However, the results show that the input image reconstruction through an autoencoder with this configuration is not a suitable self-supervised task for ResNetVLAD. We therefore did not continue the experiments on the other datasets.
Setup:

- Trained on: Pitts30k, train set
- Tested on: Pitts30k, val and test set
- Reached convergence after 5 epochs, in 7h24m

Dataset	R@1	R@5	R@10	R@20	R_avg
Pitts30k Val	72.2	86.6	91.0	94.4	86.1
Pitts30k Test	69.1	84.6	89.2	93.0	84.0

Table 4.9: Results of ResNetVLAD + autoencoder trained on Pitts30k

4.2.10 NetVLAD on Turin81/Turin1M

Finally, in order to achieve our thesis' goal, we had to find a way to make our algorithms scalable to whole cities, and prove that they could work even on such large datasets. For the final, city-wide test we decided to use the classical implementation of NetVLAD, trained on Pitts30k

4.2.10.1 Algorithm

The whole algorithm needs to be processed fast enough to be used through the GUI. Therefore, the classical pattern of online processing would be way too slow for our purpose (it might take days for a city).
One of the biggest challenges was due to the fact that even if each panorama was about 400 KB, its related vector of features, calculated with NetVLAD, was 1284 KB, and all together they would require huge amounts of memory. This is because the vector of features is a vector of 32768 (2^{15}) floats of 4 bytes, and we take 10 crops from each panorama, and we therefore have 10

vectors of features. In order to calculate the euclidean distance between a vector of features of a query and all the vector of features of the database images, it is necessary to load the features of the whole database each time that we have a new query. This process could take days in a city-wide dataset. As an example, in our Turin1M dataset the images need about 300 GB of storage, while its vector of features need around 1200 GB. Just loading sequentially in memory such a huge dataset, without doing any computation on it, requires almost a day.
To overcome this memory obstacle, we decided to apply PCA on all the vectors of features, to reduce the dimension from 32768 to a smaller number, to make sure that these vectors could fit in RAM without losing too much accuracy. After trying different values for the final dimension of the vectors, we decided 256 to be the best one: using PCA 256 the recall5 on the Pittsburgh dataset drops from 94.8% to 91.8%, but the dimension of the vectors of features could be reduced by 128 times. The memory required to store the features for the Turin1M dataset could therefore be reduced from 1200 GB to about 10 GB, which made them easy and fast to load to RAM. But even with the whole dataset loaded in RAM, finding the nearest neighbours with traditional methods would require too much time, as its complexity varies between O(nd+kn) and O(ndk), depending on the algorithmic choices, where n is the size of the database set, d the size of the query set, and k is KNN's hyperparameter. To speed up the KNN search, we found a great library developed by Facebook AI Research team named Faiss [20].
Faiss is a library for efficient similarity search and clustering of dense vectors. It is written in C++ with complete wrappers for Python/numpy. Faiss [20] has various implementation of KNN search, many of which give approximated results, and which can be performed on GPU. Moreover, Faiss [20] has a great tool for clustering, which can also be performed on GPU. The clustering can be of great help because given a vector of features from a query, we can perform the KNN only on the features of the database images which are closest to the features of the query. To find the closest ones, we just take those that belong to the same cluster to the query features, or to the neighboring ones. This helps to avoid computation on the farthest vectors of features, which obviously would not improve the recall, and would greatly slow down the algorithm.

Step	Duration	Offline/Online	Disk usage
Download metadata	3 hours	Offline	300 MB
Download panoramas	1 day	Offline	300 GB
Features computation	2 days	Offline	1200 GB
PCA-256 computation	2 days	Offline	10 GB
Faiss clustering	2 hours	Offline	10 GB
Retrieval	$<$ 1 sec*	Online	NA

Table 4.10: We show the duration of each step for the retrieval on Turin1M. *The retrieval time is calculated for one query

Once the Faiss [20] index is built on the Turin1M dataset, the model can be tested through the GUI or by our dedicated step in the pipeline. This pipeline step is useful to test a set of images in a supervised environment, while with the GUI we can test one image at a time, without knowing the recall but watching the database images predicted. In order to process and test big images like the ones produced by a smartphone, we have decided to perform a 5-crop during the pre-processing. Each crop has square shape equal to the 90% of the smallest side of its original image. Then each crop is resized to a smaller dimension (224x224 pixels) and finally tested. The final result is produced by a voting performed on the produced predictions of each single crop.
Summarizing, the test on each query is performed in this way:

- produce 5 crops;
- extract a certain number of predictions for each crop called *preds_per_crop*;
- the predictions that will be pooled for the entire query is *preds_per_query*, where *preds_per_query* $<$ *preds_per_crop*);
- assign to all the *preds_per_crop* different weights: the first *preds_per_query* have weight A and to all the rest (*preds_per_crop* - *preds_per_query*) a weight B, where $A > B$. In this way the first preds_per_query predictions of each crop will have higher importance w.r.t. the others, so the most common predictions between the 5 crops will be extracted;
- only the *top_per_query* (at most *preds_per_query*) predictions are used to show the results and calculate the recalls.

4.2.10.2 Setup

- Model: NetVLAD [5] with VGG16 [37] pre-trained on Pitts30k;
- Database: Turin1M
- Query set: Turin81
- PCA: 256
- Precision: 100
- Predictions per crop: 1000
- Predictions per query: 100
- Top per query (max recall): 100
- Weights for voting: $3 \cdot preds\_per\_query + 2 \cdot (preds\_per\_crop - preds\_per\_query)$

4.2.10.3 Results

Finally, the final results showed us that our thesis' goal was mostly achieved. The whole system takes less than a second to find the location of a query, and 86% of the queries are correctly located within the first 20 predictions.

Set	R@1	R@5	R@20	R@50	R@100
$Turin81$	40.7	61.7	86.4	87.7	91.4

Table 4.11: Test of Turin81 over Turin1M

4.2.10.4 GUI

Once, the dataset is built, we want to make easy for a novice user execute two kind of operation: access to db (not really implemented for the thesis goal) and visualize the algorithm results, for an uploaded photo. In order to allow this, we have developed a web-like interface, the main page is shown in the Figure 4.3. The software allows to choose a city, from the available (actually Turin), set some filter, for example it's possible to select only a specific area of the selected city, through the map of the right, and finally explore the database visualizing all the image matching the filter, or upload a user photo and run the deep learning algorithm. At this point the software

remands the user to a new page, where the results are shown. Further, for what concerns the DL software-feature, the user can navigate on the map, that will show markers in the places predicted (Figure 4.4).
The website is developed in HTML, CSS, BOOTSTRAP and JavaScript for the client side, while in php for the server one. The maps shown are implemented by means of the official Google API for JS, so they includes all the features provided by Google Maps, like navigate through the streets with Google Street View.
Actually, the filter that works is the one on the geographical area. Once Torino is selected between the cities, the map is updated and a drawable rectangle allows the user to select a specified zone. The corresponding coordinates are passed to the algorithm and are applied only at the end, so are shown only the predicted images that matches the user specified coordinates. The number of images returned by the algorithm is at most 20.
The improvement for future is to apply the filter before the retrieval is performed.

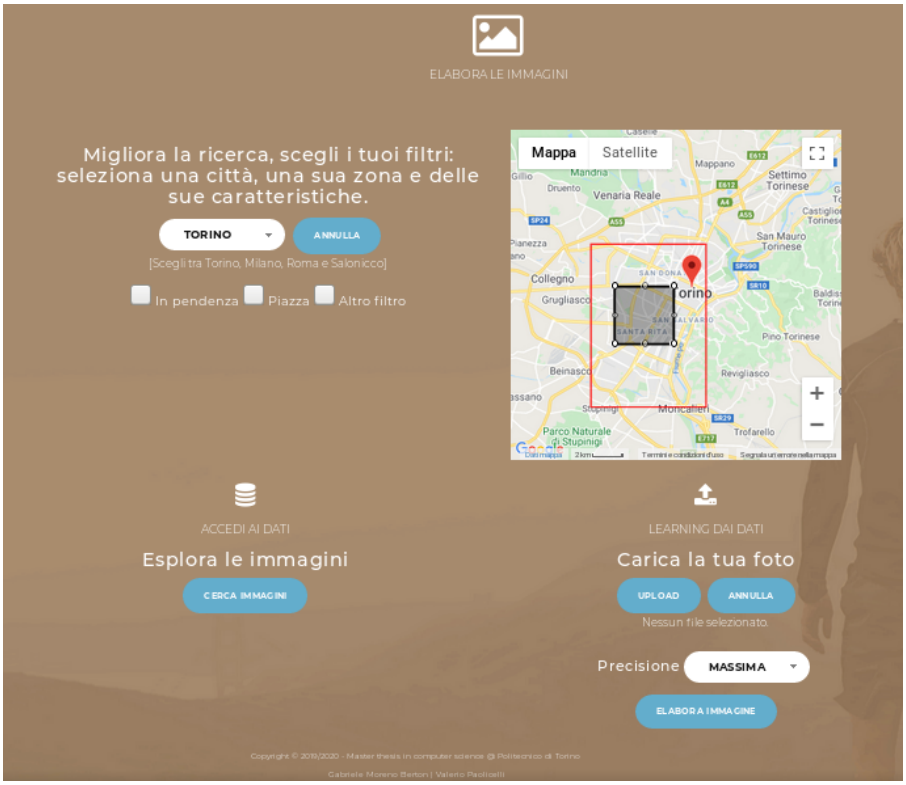


Figure 4.3: Main page of web site

Figure 4.4: Result-page of web site: on top there is the user-uploaded image, while in the red circles all the correct predictions. The map at the bottom shows the positions of the predictions.

4.3 Comparison and Discussion

To facilitate the comparison between the various methods used we present a comparison with the results collected from the previous sections. Results on Turin30k are shown in Table 4.12, while results on Pitts30k are shown in Table 4.13. For clearer visualization, only the average value of the recall is shown (average between recall 1, 5, 10 and 20).
On the Turin30k dataset we can see that removing the distortion can give a slight improvement, and DyNetVLAD performs better than NetVLAD. Inpainting does not seem to give any acceptable improvement, as the recall is higher on the val test but lower on the test set.

Method	R_avg Val	R_avg Test
ResNetVLAD on Turin30k (Baseline)	89.3	89.9
ResNetVLAD on Turin30k_undistorted	**89.7**	**90.2**
ResNetVLAD on Turin30k_inpainted	**89.4**	89.7
DyNetVLAD on Turin30k	**90.2**	**90.9**

Table 4.12: Comparison of various methods on Turin30k. In bold are all the results that work better than the baseline. They are all in bold (and not just the best one) because the methods can be applied together (e.g. undistort + inpainting + DyNetVLAD)

On the Pitts30k dataset we can see that DyNetVLAD performs better than NetVLAD, and unlike in the Turin30k dataset, inpainting can also give a slight improvement.

Method	R_avg Val	R_avg Test
ResNetVLAD on Pitts30k (Baseline)	94.4	92.0
ResNetVLAD on Pitts30k_inpainted	**94.6**	**92.3**
DyNetVLAD on Pitts30k	**94.9**	**92.6**
NetVLAD + autoencoder on Pitts30k	86.1	84.0

Table 4.13: Comparison of various methods on Pitts30k. In bold are all the results that work better than the baseline. They are all in bold (and not just the best one) because the methods can be applied together (e.g. inpainting + DyNetVLAD)

Chapter 5

Conclusions and future works

In this thesis we have focused on the visual place recognition (VPR) task, implementing deep learning algorithms and developing software to make easier the dataset creation related to Google Street View images and the geolocalization of photo uploaded by a user. As explained in this thesis, our pipeline allows to download metadata and images for all the places covered by Google Street View and perform offline many expensive operations, reducing the amount of time needed for the actual retrieval test. In our future works we'll work on more challenging datasets, for example where the database and query images belong to further domains such as day/night or winter/summer. Moreover, we have already started removing of distortion of the Turin1M dataset, and from the first tests, it seems to have slightly improved the results. The most interesting works that should be immediately tried from our point of views are:

- attention module in the network, to focus on the parts of the image that are deemed as more important to the retrieval task;
- reduce the number of features used during the VPR task, with deep learning methods, trying to improve the accuracy;
- annotating the images with relevant static objects that might help the retrieval, such as streetlight poles, benches or statues;
- try different domain adaptation and domain generalization methods, in order to make the neural network perform well even on more diverse datasets.

Bibliography

[1] David Lowe. «Distinctive Image Features from Scale-Invariant Keypoints». In: *International Journal of Computer Vision* 60 (Nov. 2004), pp. 91–. DOI: 10.1023/B:VISI.0000029664.99615.94.

[2] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. «Speeded-up robust features (SURF)». In: *Computer Vision and Image Understanding* 110 (June 2008), pp. 346–359. DOI: 10.1016/j.cviu.2007.09.014.

[3] Relja Arandjelovic and Andrew Zisserman. «All about VLAD». In: June 2013, pp. 1578–1585. DOI: 10.1109/CVPR.2013.207.

[4] Aude Oliva and Antonio Torralba. «Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope.» In: *International Journal of Computer Vision* 42.3 (2001), pp. 145–175. URL: http://dblp.uni-trier.de/db/journals/ijcv/ijcv42.html#OlivaT01.

[5] Relja Arandjelovic, Petr Gronát, Akihiko Torii, Tomás Pajdla, and Josef Sivic. «NetVLAD: CNN Architecture for Weakly Supervised Place Recognition.» In: *CVPR*. IEEE Computer Society, 2016, pp. 5297–5307. ISBN: 978-1-4673-8851-1. URL: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2016.html#ArandjelovicGTP16.

[6] Albert Gordo, Jon Almazan, Jerome Revaud, and Diane Larlus. *End-to-end Learning of Deep Visual Representations for Image Retrieval*. 2016. eprint: arXiv:1610.07940.

[7] Jiri Matas, Ondrej Chum, Martin Urban, and Tomas Pajdla. «Robust Wide Baseline Stereo from Maximally Stable Extremal Regions». In: *Image and Vision Computing* 22 (Sept. 2004), pp. 761–767. DOI: 10.1016/j.imavis.2004.02.006.

[8] Navneet Dalal and Bill Triggs. «Histograms of Oriented Gradients for Human Detection». In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005)* 2 (June 2005).

[9] Charbel Azzi, Daniel Asmar, Adel Fakih, and John Zelek. «Filtering 3D Keypoints Using GIST For Accurate Image-Based Localization». In: Jan. 2016, pp. 127.1–127.12. DOI: 10.5244/C.30.127.

[10] J. Sivic and A. Zisserman. «Video Google: A Text Retrieval Approach to Object Matching in Videos». In: vol. 2. Nov. 2003, 1470–1477 vol.2. DOI: 10.1109/ICCV.2003.1238663.

[11] Hervé Jégou, Matthijs Douze, Cordelia Schmid, Theme COG, and Equipe-Projet Lear. «Hamming Embedding and Weak Geometry Consistency for Large Scale Image Search - Extended version». In: (Oct. 2008).

[12] Hervé Jégou, Matthijs Douze, Jorge Sánchez, Patrick Perez, and Cordelia Schmid. «Aggregating Local Image Descriptors into Compact Codes». In: *IEEE transactions on pattern analysis and machine intelligence* 34 (Dec. 2011). DOI: 10.1109/TPAMI.2011.235.

[13] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. «On the burstiness of visual elements». In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '09)* (June 2009). DOI: 10.1109/CVPRW.2009.5206609.

[14] Relja Arandjelović and Andrew Zisserman. «DisLocation: Scalable Descriptor Distinctiveness for Location Recognition». In: Nov. 2014, pp. 188–204. ISBN: 978-3-319-16816-6. DOI: 10.1007/978-3-319-16817-3_13.

[15] Ke Yan, Yaowei Wang, Dawei Liang, Tiejun Huang, and Yonghong Tian. «CNN vs. SIFT for Image Retrieval: Alternative or Complementary?» In: Oct. 2016, pp. 407–411. DOI: 10.1145/2964284.2967252.

[16] Pilailuck Panphattarasap and Andrew Calway. *Visual place recognition using landmark distribution descriptors*. 2016. eprint: arXiv:1608.04274.

[17] Ali Sharif Razavian, Josephine Sullivan, Stefan Carlsson, and Atsuto Maki. *Visual Instance Retrieval with Deep Convolutional Networks*. 2014. eprint: arXiv:1412.6574.

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

[19] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. «Object retrieval with large vocabularies and fast spatial matching». In: June 2007. DOI: 10.1109/CVPR.2007.383172.

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

[21] Colin Mcmanus, Ben Upcroft, and Paul Newmann. «Scene Signatures: Localised and Point-less Features for Localisation». In: July 2014. DOI: 10.15607/RSS.2014.X.023.

[22] M. Aubry, B. Russell, and J. Sivic. «Painting-to-3D Model Alignment Via Discriminative Visual Elements». In: *ACM Transactions on Graphics* (2013). Pre-print, accepted for publication.

[23] Akihiko Torii, Josef Sivic, and Tomas Pajdla. «Visual localization by linear combination of image descriptors». In: Nov. 2011, pp. 102–109. DOI: 10.1109/ICCVW.2011.6130230.

[24] Tsung-Yi Lin, Serge Belongie, and James Hays. «Cross-View Image Geolocalization». In: June 2013, pp. 891–898. DOI: 10.1109/CVPR.2013.120.

[25] Nam Vo and James Hays. *Localizing and Orienting Street Views Using Overhead Imagery*. 2016. eprint: arXiv:1608.00161.

[26] Mayank Bansal, Harpreet Sawhney, Hui Cheng, and Kostas Daniilidis. «Geo-localization of street views with aerial image databases». In: Nov. 2011, pp. 1125–1128. DOI: 10.1145/2072298.2071954.

[27] Tobias Weyand, Ilya Kostrikov, and James Philbin. «PlaNet - Photo Geolocation with Convolutional Neural Networks». In: (2016). DOI: 10.1007/978-3-319-46484-8_3. eprint: arXiv:1602.05314.

[28] Torsten Sattler, Will Maddern, Carl Toft, Akihiko Torii, Lars Hammarstrand, Erik Stenborg, Daniel Safari, Masatoshi Okutomi, Marc Pollefeys, Josef Sivic, Fredrik Kahl, and Tomas Pajdla. *Benchmarking 6DOF Outdoor Visual Localization in Changing Conditions*. 2017. eprint: arXiv:1707.09092.

[29] Paul-Edouard Sarlin, Cesar Cadena, Roland Siegwart, and Marcin Dymczyk. *From Coarse to Fine: Robust Hierarchical Localization at Large Scale*. 2018. eprint: arXiv:1812.03506.

[30] Youji Feng, Lixin Fan, and Yihong Wu. «Fast Localization in Large Scale Environments Using Supervised Indexing of Binary Features». In: *IEEE Transactions on Image Processing* 25 (Nov. 2015), pp. 1–1. DOI: 10.1109/TIP.2015.2500030.

[31] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. *Pyramid Scene Parsing Network*. 2016. eprint: arXiv:1612.01105.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. eprint: arXiv:1512.03385.

[33] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. «Semantic understanding of scenes through the ade20k dataset». In: *arXiv preprint arXiv:1608.05442* (2016).

[34] Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. «Scene Parsing through ADE20K Dataset». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017.

[35] Guilin Liu, Fitsum Reda, Kevin Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. «Image Inpainting for Irregular Holes Using Partial Convolutions». In: (Apr. 2018).

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. «U-Net: Convolutional Networks for Biomedical Image Segmentation». In: (May 2015).

[37] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[38] Cordelia Schmid Hervé Jégou Matthijs Douze and Patrick Pérez. «Aggregating local descriptors into acompact image representation». In: *CVPR 2010 - 23rd IEEE Conference on Computer Vision and PatternRecognition, Jun 2010, San Francisco, United States* (2010). URL: https://hal.inria.fr/inria-00548637.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[40] Nanne. *pytorch-NetVlad*. URL: https://github.com/Nanne/pytorch-NetVlad.

[41] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. «Places: A 10 million Image Database for Scene Recognition». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).