University of Passau
Faculty of Computer Science and Mathematics

Chair of Data Science
Prof. Dr. Michael Granitzer

INSA Lyon
Département Informatique

Double Master Program

Masterarbeit

Perceptual Hashing using Convolutional Neural Networks for Large Scale Reverse Image Search

Mathieu Gaillard

Date: September 2017

Supervisors: Prof. Dr. Michael Granitzer
Prof. Dr. Lionel Brunie
Dr. Elöd Egyed-Zsigmond

Erklärung zur Masterarbeit
Declaration of Authorship

Name, Vorname des
Studierenden: Gaillard, Mathieu

Universität Passau,
Fakultät für Informatik und Mathematik

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate auch als solche gekennzeichnet habe.

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

....
(Datum) (Unterschrift des Studierenden)

Supervisor Contacts:

Prof. Dr. Michael Granitzer
Chair of Data Science
Universität Passau
EMail: michael.granitzer@uni-passau.de
Web: http://mgrani.github.io

Prof. Dr. Lionel Brunie
Institut National des Sciences Appliques de Lyon
EMail: lionel.brunie@insa-lyon.fr
Web: http://liris.cnrs.fr/lionel.brunie

Dr. Elöd Egyed-Zsigmond
Institut National des Sciences Appliques de Lyon
EMail: elod.egyed-zsigmond@insa-lyon.fr
Web: http://liris.cnrs.fr/eegyedzs

Acknowledgements

I would like to thank the initiators of the double Master IFIK for their valuable efforts and commitment to the German-French collaboration : Prof. Dr. Harald Kosch for the German side and Prof. Dr. Lionel Brunie for the French side.

I would like to express my deep gratitude to my supervisors Prof. Dr. Michael Granitzer and Dr. Elöd Egyed-Zsigmond, for their enthusiastic encouragement and useful critiques of this research work.

I would also like to thank my friends from the double degree for their support and encouragement throughout my study. Especially Guillaume Kheng, Quidditch Deutscher Meister, who was also living with me in Braugasse and with whom I spend literally 99% of my time in Passau. Thanks to him I did a lot of sport, I am sure that my work benefited from that.

Finally, a special thank to Morwenna Joubin, without whom it wouldn't be possible for me to submit this document from France, in time.

Abstract

In this master thesis, we present our study on Convolutional Neural Networks Features and Perceptual Hashing for Large Scale Reverse Image Search. We especially focus our attention on robustness of such systems against common modifications (Gaussian blur, color filter, resize, compression, rotation, cropping). In a first part, we design a benchmark to evaluate the speed and accuracy of several existing techniques. These techniques have very good retrieval performances except against rotation and cropping. In a second part, we investigate the use of CNN Features for reverse image search. Experiments show that they are considerably robust against modifications. To efficiently perform a nearest neighbor search we advocate the use of hashing into short binary codes. In a third part, we propose a supervised method for learning a binary hash function that preserve similarity. This method is based on LSH with random projection and Minimal Loss Hashing, we propose a new approach to optimize the hash function in a continuous space. Experiments show that this approach is valid and promising for hashing CNN Features.

Résumé

Dans cette thèse de master, nous présentons une étude sur les caractéristiques extraites à partir de réseaux de neurones convolutifs et les hash perceptuels pour la recherche d'images inversée à grande échelle. Nous portons en particulier notre attention sur la robustesse de ces systèmes face à des modifications communes (flou Gaussien, filtre de couleur, redimensionnement, compression, rotation, rognage). Dans un premier temps, nous concevons un protocole pour comparer la vitesse et la précision de plusieurs techniques existantes. Ces techniques ont de bonnes performances, exceptées contre la rotation et le rognage. Dans un second temps, nous examinons l'utilisation des caractéristiques extraites à partir de réseaux de neurones convolutifs pour la recherche d'image inversée. Les expériences montrent qu'elles sont considérablement robustes aux modifications. Pour effectuer efficacement une recherche de plus proches voisins, nous recommandons l'utilisation de techniques de hachage binaire. Dans un troisième temps, nous proposons une méthode supervisée pour apprendre une fonction de hachage binaire qui préserve les similarités. Cette méthode est basée sur LSH avec des projections aléatoires et sur Minimal Loss Hashing. Nous proposons une nouvelle approche pour optimiser la fonction de hachage dans un espace continu. Les expériences montrent que cette approche est valide et prometteuse pour hacher les caractéristiques extraites à partir de réseaux de neurones convolutifs.

Contents

1. Introduction

1.1. Background

This study is mainly motivated by the need to find the original of an image in a large image collection given a slightly modified version of it. This problem is called Reverse Image Search (RIS) and is extensively studied because of its applications in the context of intellectual property and crime prevention. More generally, Reverse Image Search is related to Content-based Image Retrieval and Information Retrieval. Many implementations of RIS systems already exist, for example Google Images, TinEye and Microsoft PhotoDNA.

As it is easy for people to take, store and share pictures, a massive amount of images appears every day on the internet. To enable indexing, searching, processing and organizing such a quickly growing volume of images, one has to develop new efficient methods capable of dealing with large scale data.

In computer vision, one extracts high-dimensional feature vectors from images on which a nearest neighbor search is then performed to query images by similarity. With a very large collection of images, representations should be as small as possible to reduce storage costs. Additionally, a data structure should allow for sublinear-time search in the database of image representations. There already exist many approaches but, since this field is very wide, it is impossible to review all of them. This is why we will focus our attention only on methods based on perceptual hashing.

The idea of perceptual hashing is to map similar (resp. dissimilar) inputs into similar (resp. dissimilar) binary codes according to a selected distance. This approach works very well because binary codes are compact and easy to handle on computers. Furthermore, recent work showed that it is possible to search binary codes in sublinear time for uniformly distributed codes.

Recently it has been shown that the activations within the top layers of a large convolutional neural network (CNN) provide a high-level descriptor of the visual content of an image. Even when the convolutional neural network has been trained for an unrelated classification task, the retrieval performance of this approach is competitive.

1.2. Motivation

A motivating use case could be the following. Let's imagine one man who regularly corresponds with girls on an online dating site. With a service that indexes all images reachable on the internet, for instance: Google Images or TinEye, he could reverse search some anonymous profile pictures. If the person uses the same picture somewhere else on the internet, he could find it, potentially along with other information. A Facebook profile or a work website could reveal the identity of the people and possibly more private information: work place, religious background, and so on. It is also possible to find out that the profile picture of someone is actually a photo of a Hollywood star.

With this example, we can see that a reverse image search service can be helpful for common people. Obviously we don't talk about the privacy issue for the people on the internet, but by following some simple rules it is possible for them to prevent that.

1.3. Purpose of the thesis

Current implementations of Reverse Image Search can perform queries with images that are compressed, grayscaled or resized. However, other modifications are harder to deal with such as: cropping and rotation. The purpose of this thesis will be to improve the robustness of such a search engine to modifications.

Based on the observation that convolutional neural networks are promising to extract robust features from images, and based on the observation that binary codes are efficient and scalable, our idea is to proceed in two steps. In a first step, we study the robustness of the CNN features vectors to modifications, the aim being to find a convolutional neural network capable of extracting robust representations. In a second step, we study how to map high-dimensional feature vectors into binary codes. By putting together these two steps, one can create a perceptual hash function that is robust to modifications.

1.4. Outline

The thesis will be organized as follows.

Chapter 2 - Reverse Image Search : presents the general concept of Reverse Image Search.

Chapter 3 - Perceptual Hashing : presents the general concept of Perceptual Hashing.

Chapter 4 - Convolutional Neural Networks : presents the Convolutional Neural Networks and shows that they can extract excellent image representations.

Chapter 5 - Hashing for Dimensionality Reduction : presents how hashing can improve the efficiency of nearest neighbor search by reducing the dimension of features vectors.

Chapter 6 - Search in Hamming Space : presents methods for nearest neighbor search for binary codes in Hamming space.

Chapter 7 - Benchmarking : presents our benchmark for Reverse Image Search.

Chapter 8 - CNN Features Robustness : presents the results of our benchmark on features extracted with off the shelf convolutional neural networks.

Chapter 9 - CNN Features Hashing : presents our approach for learning a hash function to map CNN Features into binary codes.

Chapter 10 - Conclusion : Concludes this master thesis.

Part I.

Review

2. Reverse Image Search

2.1. Definition

To better understand the definition and context of the Reverse Image Search problem, we first give the definition of two more general problems: Image Retrieval and Content-based Image Retrieval.

An image retrieval system is a computer system for browsing, searching and retrieving images from a large database of digital images. [Wik17b] Examples of such a system are *Google Image* [1] and *Bing Image* [2], where the user writes a text query and have a list of images in return.

Content-based image retrieval (CBIR) is the application of computer vision techniques to the image retrieval problem. "Content-based" means that the search analyses the actual content of the image rather than the metadata such as keywords, tags, or descriptions associated with the image. [Wik17a] An example of such a system is *Google Image*, where the user can perform a search by image, either by URL or by uploading a file. If one submits an image of a cat, in return we will get images of visually similar cats, but not necessarily the exact same cat.

Reverse image search (RIS) is a content-based image retrieval (CBIR) query technique. The aim of a RIS system is to find the original of an image in a large image collection given a slightly modified version of it. An example of such a system is *TinEye* [3], where the user can submit an image and find out where this exact image appears on internet.

2.1.1. Usage

The main applications are listed in the FAQ [4] of *TinEye*.

- Find out where an image came from, or get more information about it
- Identify duplicate images

[1] www.google.com
[2] www.bing.com
[3] www.tineye.com
[4] www.tineye.com/faq

- Find a higher resolution version of an image
- Locate web pages that make use of an image you have created
- Discover modified or edited versions of an image
- Show that the information provided with an image is false

Following are more specific use cases: A dating site can verify that a profile is authentic. An insurance company can detect fraud. Trademark offices can identify infringing trademarks. A museum can provide a mobile application on which the user can have additional details about a painting. A brand can replace QR codes and connect a printed catalog to an ecommerce platform.

2.2. General framework

Most approaches to Reverse Image Search share a similar pattern. It consists of: firstly, a way of representing an image, and secondly, a distance (or similarity) measure between two representations. As shown in figure 2.1, RIS systems usually work in two phases: indexing and searching. During the indexing phase, representations of all the images in a collection are extracted and added to a database. The images are not necessarily stored in the database; this reduces its size. Afterward during the searching phase, a query image is presented to the system and its representation is extracted. A nearest neighbor search is then performed with the query representation, using the previously defined distance measure, to search for similar images in the database.

Reverse Image Search with large collections of images imposes two challenging constraints on the methods used. Firstly, for each image, only a small amount of data can be stored; secondly, queries must be very cheap to evaluate.

2.2.1. Image representation

The image representation is usually a p-dimensional vector of numerical features or a q-bit binary code. The feature vectors should be compact, in order to store millions of them in a database. Many algorithms are able to extract feature vectors from images based on color, texture, shape. We can classify features in two categories: low-level features, which are minor details of the image that are detected by simple algorithms: edges, corners, and so on, and high-level features, which carry more semantic and thus are better understandable by humans: objects, actions, and so on. High-level features are difficult to extract for computers because it is hard to understand the semantic of an image, this is currently an active research field. In

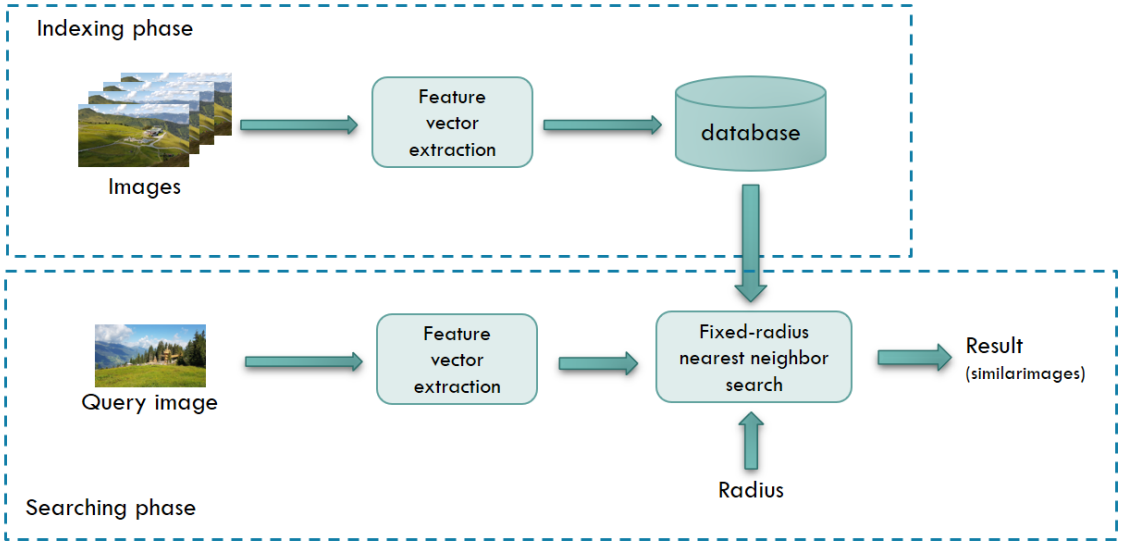

Figure 2.1.: General framework for Reverse Image Search with fixed-radius nearest neighbor search.

the following part, we will describe some algorithms to extract low-level features from images.

A very simple approach is to compute the color histogram of an image. The image representation in this case is a vector of N-bins. If there is too much different colors in the image, it is possible to subsample the color space. To compare two histograms, we can use a distance based on correlation.

Color and Edge Directivity Descriptor (CEDD) [CB08] combines, in one histogram, color and texture information. The feature vector size is up to 54 bytes per image. The similarity between two CEDD histograms is measured with Tanimoto coefficient.

Scale-invariant Feature Transform (SIFT) [Low04] extracts distinctive invariant features from images that can be used to perform reliable matching between different views of an object or scene. The features are invariant to image scale and rotation, and are shown to provide robust matching across a substantial range of affine distortion. SIFT detects key points in the image and compute small descriptor for each of them. The number of key points can vary and this is why a feature vector extracted with SIFT does not have a fixed length. To compare two images, their keypoints are matched by identifying their nearest neighbors, which can be very costly dependending on the number of keypoints.

The GIST descriptor was initially proposed in [OT01]. It is based on Gabor filters, which measure the orientations and spatial frequencies, and describes a scene with a set of perceptual dimensions (naturalness, openness, roughness, expansion,

ruggedness). The GIST description is a feature vector of dimension 960, which can be compared with an Euclidean distance [DJH+09].

2.2.2. Distance

To compare two image representations one has to choose a distance or similarity function. Depending on the representation many distances are possible.

p-dimensional vectors

If the image representation is a p-dimensional feature vector, following are distances or similarities between $x \in \mathbb{R}^p$ and $y \in \mathbb{R}^p$.

Minkowski distance is a generalization of Manhattan and Euclidean distances. Euclidean distance can be interpreted as the ordinary straight-line distance between two points in Euclidean space.

$$d_{minkowski}(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

When $p = 1$, the Minkowski distance is equivalent to the Manhattan distance. When $p = 2$, it is equivalent to the Euclidean distance. If the function is used to rank vectors according to their distance we can save time by not computing the nth root because this function is monotonically increasing.

Cosine similarity measures the cosine of the angle between two vectors. The cosine similarity is equal to 1 if the angle between vectors is 0. It is equal to 0 if the angle between vectors is $\pi/2$.

$$s_{cosine}(x, y) = \cos \theta(x, y) = \frac{x \cdot y}{\|x\|_2 \|y\|_2}$$

$$d_{cosine}(x, y) = \frac{\cos^{-1} s_{cosine}(x, y)}{\pi}$$

For example, the cosine similarity is used in information retrieval. In this field, documents are often represented as vectors containing the number of occurrences of terms. The cosine similarity is meaningful to measure the similarity of two documents regarding their subjects. In facts, what is important is not the number of time a term appears, but whether it appears or not.

Cosine similarity is related to Euclidean distance if the vectors are normalized to unit length.

$$\|x-y\|^2 = (x-y)\cdot(x-y) = x\cdot x+y\cdot y-2x\cdot y = \|x\|^2+\|y\|^2-2(\|x\|^2\|y\|^2\cos\theta(x,y))$$

Because $\|x\| = \|y\| = 1$, we can conclude:

$$\|x-y\|^2 = 2(1-\cos\theta(x,y))$$

Binary codes

If the image representation is a q-bit binary code, following are distances or similarities between $a \in [0,1]^q$ and $b \in [0,1]^q$.

Hamming distance is the number of positions at which the bits are different. It measures the edit distance between two binary codes if the only allowed operation to transform the code is to flip a bit.

$$d_{hamming}(a,b) = \sum_{i=1}^{n}(a_i \oplus b_i)$$

On computers, this distance is straightforward to compute because it is the number of ones (population count) in the XOR of two binary codes. Since the population count and XOR are two basic operations in modern CPUs, the computation of Hamming distance is very efficient. Following is an implementation in C with two 64-bit binary codes using the POPCOUNT .

```
int hamming_distance(uint64_t a, uint64_t b)
{
    return __builtin_popcountll(a ^ b);
}
```

Simple Matching Coefficient (SMC) is the number of matching bits divided by the length of the binary codes. It is useful when both a value of 0 and 1 for a bit carry equal information. SMC is used for comparing the similarity of two binary codes, to measure a distance one can use the Simple Matching Distance (SMD).

$$SMC = \frac{\text{number of matching bits}}{\text{number of bits}} = \frac{M_{00}+M_{11}}{M_{00}+M_{01}+M_{10}+M_{11}}$$

where:

M_{00} is the number of positions where a and b both have a value of 0.

M_{01} is the number of positions where a has a value of 0 whereas b has a value of 1.

M_{10} is the number of positions where a has a value of 1 whereas b has a value of 0.

M_{11} is the number of positions where a and b both have a value of 1.

$$\begin{aligned} SMD &= 1 - SMC \\ &= 1 - \frac{M_{00} + M_{11}}{M_{00} + M_{01} + M_{10} + M_{11}} \\ &= \frac{M_{00} + M_{01} + M_{10} + M_{11} - M_{00} - M_{11}}{M_{00} + M_{01} + M_{10} + M_{11}} \\ &= \frac{M_{01} + M_{10}}{M_{00} + M_{01} + M_{10} + M_{11}} \end{aligned}$$

SMD is related to Hamming distance because $M_{01} + M_{10}$ is equal to the number of positions at which the bits are different, that is the Hamming distance.

$$d_{SMD}(a, b) = \frac{d_{hamming}(a, b)}{q}$$

Jaccard Similarity Coefficient is the number of positions where the two binary codes share a 1 divided by the number of positions where at least one binary codes has a value of 1. To measure a distance, one can use the Jaccard distance. The Jaccard Similarity Coefficient is very similar to the Simple Matching Coefficient, the only difference is that it does not take in account the positions where the two binary codes share a 0. It is useful for asymmetric binary data where a value of 0 and 1 for a bit does not carry equal information, for example a market basket data.

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

M_{00} is the number of positions where a and b both have a value of 0.

M_{01} is the number of positions where a has a value of 0 whereas b has a value of 1.

M_{10} is the number of positions where a has a value of 1 whereas b has a value of 0.

M_{11} is the number of positions where a and b both have a value of 1.

$$d_{Jaccard}(a, b) = 1 - J = \frac{M_{01} + M_{10}}{M_{01} + M_{10} + M_{11}} = \frac{d_{hamming}(a, b)}{q - M_{00}}$$

It is possible to compute the term M_{00} efficiently. It is the number of ones (population count) in $\neg(a \parallel b)$.

2.2.3. Nearest Neighbor Search

Nearest Neighbor Search (NNS) is the problem of finding in a database all the items whose distances to a query item are the smallest. Two variants are interesting for CBIR and RIS: k-nearest neighbors search and fixed-radius near neighbors.

K-nearest neighbors search aims to find the k nearest neighbors of a given query point. Usually the results are ordered by decreasing similarity (i.e. increasing distance). When applied to CBIR it is interesting in order to explore an image collection, because if the first results are not interesting, one can just look at the following results.

Fixed-radius near neighbors aims to find all points that are within a radius of a given query point. Even if it is possible to sort the results according to the distance, the results should be considered as a set of unranked images. When applied to CBIR, this method is interesting for identifying content because only relevant images are returned, thus the result is composed of a varying number of images. The determination of an adequate radius, in accordance with the actual application, is critical. Information retrieval research has shown that precision and recall follow an inverse relationship [DJLW08]. If the radius is too low, the precision is better at the expense of the recall, and vice versa. Depending on the application, one can want to favor precision, to authenticate content because there is fewer false-positives, or recall, to identify content because the user can deal with false positives.

Suppose our dataset is composed of p-dimensional feature vectors in a Euclidean space $D \equiv \{x_i\}_{i=1}^{n}$ where $x_i \in \mathbb{R}^p$. Let $z \in \mathbb{R}^p$ be a query feature vector, the one-nearest neighbor of the query z is defined as:

$$KNN_1(z) = \underset{1 \leq i \leq n}{\arg\min} \|z - x_i\|^2$$

Let $r \in \mathbb{R}^+$ be a radius, the fixed-radius nearest neighbors of the query z are defined as:

$$FRNN_r(z) = \{y \in D \mid \|y - z\| \leq r\}$$

For p-dimensional feature vectors, there exist algorithms for exact nearest neighbor search such as the k-d tree, R-tree or MVP tree. But unfortunately, because of the curse of dimensionality, they are not efficient for high dimensional data (more than 20 dimensions). For this reason, Approximate Nearest Neighbor Search (ANNS) is gaining more interest because it allows for faster searching time with only small actual errors. In any case, we can refine a list of approximate nearest neighbors by pruning the items with the actual distance on the original features. The Locality Sensitive Hashing (LSH) framework is one approach to Approximate Nearest Neighbor Search and is detailed later in this part.

3. Perceptual Hashing

3.1. Definition

A perceptual hash function is a type of hash function that has the property to return analogous outputs if inputs are similar. This allows one to make meaningful comparisons between hashes in order to measure the similarity between the source data. The definition of a hash function according to [MVO96] is:

> A hash function is a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash-values.

In the case of a perceptual hash function, four more properties should be present according to [Zau10]:

> Let H denote a hash function which takes one media object (e.g. an image) as input and produces a binary string of length l. Let x denote a particular media object and $\widetilde{x}$ denote a modified version of this media object which is "perceptually similar" to x. Let y denote a media object that is "perceptually different" from x. Let x' and y' denote hash values. $0,1^l$ represents binary strings of length l. Then the four desirable properties of a perceptual hash are identified as follows.
>
> A uniform distribution of hash-values; the hash-value should be unpredictable.
>
> $$\mathbb{P}(H(x) = x') \approx \frac{1}{2^l} \quad \forall x' \in \{0,1\}^l$$
>
> Pairwise independence for perceptually different media objects.
>
> $$\mathbb{P}(H(x) = x' | H(y) = y') \approx \mathbb{P}(H(x) = x') \quad \forall x', y' \in \{0,1\}^l$$
>
> Invariance for perceptually similar media objects.
>
> $$\mathbb{P}(H(x) = H(\widetilde{x})) \approx 1$$

Distinction of perceptually different media objects. It should be impossible to construct a perceptually different media object that has the same hash-value as another media object.

$$\mathbb{P}(H(x) = H(y)) \approx 0$$

Most of the time, to achieve these properties, the perceptual hash function extracts some features of media objects that are invariant under slight modifications. For example, knowing how a compression algorithm works, it is possible to find some invariant features and then design a perceptual hash based on them. Some examples of perceptual hash functions for images are detailed later in this section.

3.2. Usage

Perceptual hash functions can be used in the context of CBIR or RIS for representing images but they are also useful in other contexts. Due to the properties inherited from the hash functions they can be used to authenticate media objects even if they are slightly modified, for example lossy compressed. The image creator can generate a perceptual hash from his original work. Then, as explained in [Zau10], he has two options.

On the one hand it is possible to sign the perceptual hash with a private key in order to have a digital signature that is robust to slight modifications, for example JPEG compression. This measure protects the receiver of the media object because he can check its authenticity.

On the other hand, it is possible to use the perceptual hash for digital watermarking. This measure protects the content author. For example, a different watermark can be applied on different images. Then if an illegal copy is found the copyright owner can infer who is responsible for the data leak.

3.3. Implementations

In this section, we present the three examples of implementations of perceptual hash functions, which take images and output binary codes, from [Zau10].

The Discrete Cosine Transformation (DCT) based perceptual hash function takes advantage of the property that low-frequency DCT coefficients are mostly stable under image modifications to construct a 64 bits binary code, which are compared with a Hamming distance. See our implementation in appendix A

The Marr-Hildreth (MH) operator, also denoted as the Laplacian of Gaussian (LoG), is a special case of a discrete Laplace filter. It is an edge and contour detection based image feature extractor. The MH operator generates vectors encoded on 576 bits.

The Radial Variance hash is based on the Radon transform that is the integral transform which consists of the integral of a function over a straight line. It is robust against various image processing steps (e.g. compression) and more robust than the DCT and MH based perceptual hash functions against geometrical transformations (e.g. rotation up to 2 degrees).

4. Convolutional Neural Networks

4.1. Definition

A Convolutional Neural Network (CNN) is a class of feed-forward neural network. CNN are mainly applied in the fields of computer vision and natural language processing. It has been first presented in [LBBH98]. Recent progresses have led to a gain in interest in this method and also in a broader field called Deep Learning, which is described in [LBH15]. This section is mainly based on this latter article. It has been a great discover because this class of network is able to automatically learn the features to extract from a dataset. Whereas traditionally, the previous works in computer vision were based on hand-engineered features. This difference is essential because the key success factor of CNN is the amount of data and computation power available instead of field knowledge.

CNN are inspired by our visual cortex, in particular, the connectivity pattern between layers of artificial neurons is restricted to a region as it is in our visual system with receptive fields. With this special connectivity pattern, CNN take advantage of natural signals, which are often locally highly correlated. Another property of natural signals is the fact that the local statistics of images are invariant to location. If a motif can appear in one location of an image, it could also appear somewhere else.

CNN are built on top of four key ideas: local connection, shared weights, pooling and use of many layers. The architecture of a typical CNN is composed of these layers; in a first part: convolution layer, ReLU, pooling layer and in a second part: fully connected layer, loss layer.

The convolution layer applies a set of discrete convolutions on the output of the previous layer. Each convolution is done using a kernel (also called: mask, filter bank) and produces a 2D feature map. The feature maps contain high activations if their convolutions have detected an interesting motif. The layer's parameters are the weights in the kernels. This layer can also be seen as a layer with local connections, so that a neuron is only connected to neurons of the previous layer in the same region. Local connections give the ability to the layer to detect local patterns from the previous layer. Moreover, the weights of the local connections are shared among all neurons in the layer, this means that the same local pattern can be detected in the same way anywhere in the previous layer.

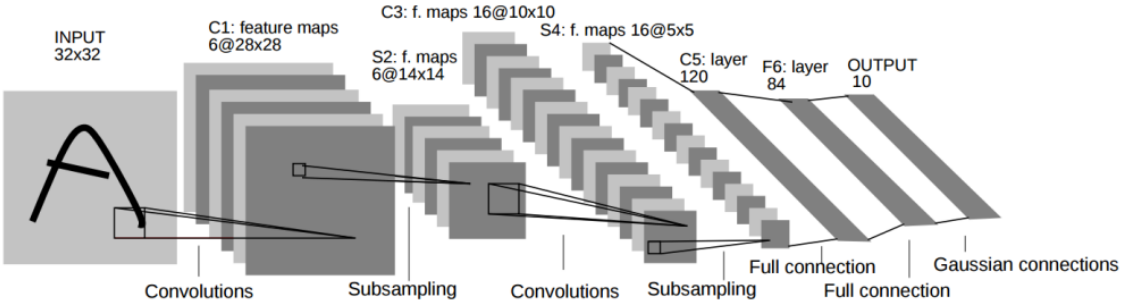

Figure 4.1.: Architecture of LeNet-5, a basic Convolutional Neural Network

The ReLU (Rectified Linear Units) applies a non-linear function $f(x) = \max(0, x)$ to the output of the previous convolution layer. Other non-linear activation function are available: hyperbolic tangent or sigmoid function for example. The ReLU is the most common because the training of the neural network is faster with it [KSH12].

The pooling layer splits the output of the previous layer into a grid, then on each cell of the grid a pooling function is applied and outputs only a single value. Thus, the pooling layer is a down-sampling step, the resolution of the grid can vary and affects the sampling rate. This layer merges multiple semantically similar features into one. Due to the down-sampling, the position of the feature is not accurately preserved, therefore an invariance to small shifts or distortions is spawned. In fact, the position of a feature is not as important as its relative position to others features. A typical pooling function outputs the maximum value of a local patch.

Repeating these layers several times in this order: convolution, ReLU, pooling, one can exploit the fact that high level features are composed of lower level features. The first group of layers can detect small details such as edges and contours. The second layer can detect higher level features such as motifs, and so on. This process is repeated so that firstly, edges are merged into motifs, motifs into parts and finally parts into objects. The architecture of a very basic CNN is shown in figure 4.1.

After this first part composed of convolution layers, ReLU, pooling layers, a second part of the neural network is composed of fully-connected layers and a loss layer. This second part is a classical feed-forward neural network used to classify the features coming from the first part. The training of CNN is possible using the backpropagation algorithm. Because of the local connectivity and the shared weights, there is less parameters in the model thus the network is less prone to overfitting and generalizes better.

4.2. Usage

Convolutional neural networks are mostly used in supervised learning for classification tasks. One of the first successful commercial application was to recognise handwritten digits on bank checks [LBBH98]. In this task, the system is given a normalized, grayscale, 28-by-28 pixels image of a digit and outputs the class of the image: zero, one, two, and so on until nine. The same technique can be used for Optical Character Recognition (OCR). By using a sliding window on an image, it is possible to use a CNN to recognize if a face is depicted at a certain position [GD04]. It is possible to use CNN for image classification [KSH12], for example the ImageNet dataset is composed of roughly 1.2 million images divided into 1,000 classes, the goal being to predict the class of a given image. Scene labeling consists in labeling each pixel in an image with the category of the object it belongs to. CNN have been successfully applied to this task [FCNL13]. It is useful for example to self-driving cars and more generally to autonomous robots. When used with one dimensional convolution, CNN are able to process acoustic signals for automatic speech recognition [SPKL15].

4.3. Feature extraction

Recent work showed that the representation learned by the CNN is a good descriptor for image retrieval. With the AlexNet network [KSH12], one can use the feature activations induced by an image at the last hidden layer to represent an image. Experiments has shown that semantically similar images get a similar feature vector in a Euclidean space, even if the images are not close in L2. The only drawback is that the representation is a real-valued vector of dimension 4096. Therefore, the computation of the distance between two vectors is expensive, moreover the storage cost could also be an issue. To tackle this issue, the authors suggest a dimensionality reduction with an auto encoder.

In a later paper [BSCL14], the performance of feature vectors extracted with CNN in the context of image retrieval is studied more in depth. The conclusion is that CNN performs well for image retrieval, even if the network is trained on an unrelated classification task. The performances can be improved with a fine tuning on a dataset similar to the retrieval dataset. For example, by using a neural network trained on ImageNet to retrieve landscape images, the performances are good. Unsurprisingly, after a retraining on a dataset composed of landscape images the performances become even better. The compression with PCA is also investigated, and the retrieval performance is not too much affected when compared to other state of the art descriptors. Image features extraction with CNN, for all these reasons, seems to be very promising for image retrieval.

4.4. Models

In this section, three models used for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) are presented. All of them are implemented in the Keras [C+15] library and are used later in this thesis.

The VGG network [SZ14] was created in 2014 for ILSVRC. The authors investigated the effect of depth on the accuracy. For that purpose, they use a simple convolutional neural network with very small (3x3) convolution filters with stride and pad of 1, 2x2 maxpooling with stride 2, and push the depth to 16-19 layers, which was a lot at the time. VGG16 and VGG19 are detailed in table 4.1.

Table 4.1.: Configurations of VGG16 and VGG19 networks. The convolutional layer parameters are denoted as "'conv(receptive field size)-(number of channels)"'. The ReLU activation is not shown for brevity.

VGG16	VGG19
input (224x224 RBG image)	
conv3-64	conv3-64
conv3-64	conv3-64
maxpool	
conv3-128	conv3-128
conv3-128	conv3-128
maxpool	
conv3-256	conv3-256
conv3-256	conv3-256
conv3-256	conv3-256
	conv3-256
maxpool	
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
	conv3-512
maxpool	
conv3-512	conv3-512
conv3-512	conv3-512
conv3-512	conv3-512
	conv3-512
maxpool	
FC-4096	
FC-4096	
FC-1000	
soft-max	

The InceptionV3 network [SVI+15] was created in 2015. Instead of stacking convolutional and pooling layers sequentially on top of each others, some layers are in parallel and their results are merged periodically. As shown in figure 4.2 from Google Research Blog [1], the key idea of this network is to stack Inception modules, which are composed of convolutional and pooling layers. This can be easily seen in figure 4.2. By using fewer weights than previous neural networks, for instance about 30x fewer parameters than VGG19, the computational cost of InceptionV3 is suitable for big-data or mobile scenarios.

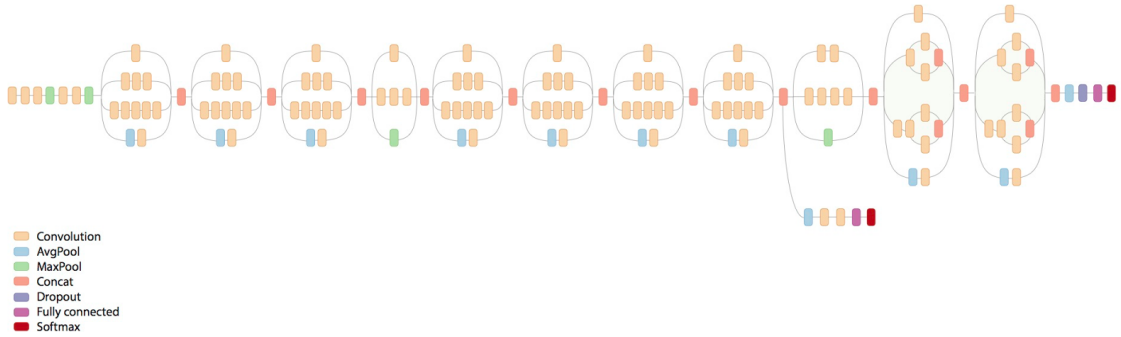

Figure 4.2.: Architecture of the InceptionV3 network.

The ResNet50 network [HZRS15] was created in 2015 for ILSVRC. The authors use a 50 layers Residual Network, whose key idea is to introduce shortcuts connections into a sequential network. As networks are becoming deeper, the learning is more difficult because of the vanishing/exploding gradient problem. In particular, with the network depth increasing, accuracy gets saturated and then degrades rapidly. The idea behind shortcuts in Residual Networks is to by-pass a set of layers if they are not useful to improve the accuracy of the whole network. Thus, in theory, it is possible to stack as many building blocks as possible without affecting the accuracy, because unnecessary blocks can be skipped. Figure 4.3 shows a Residual Network with 34 layers

[1] https://research.googleblog.com/2016/03/train-your-own-image-classifier-with.html

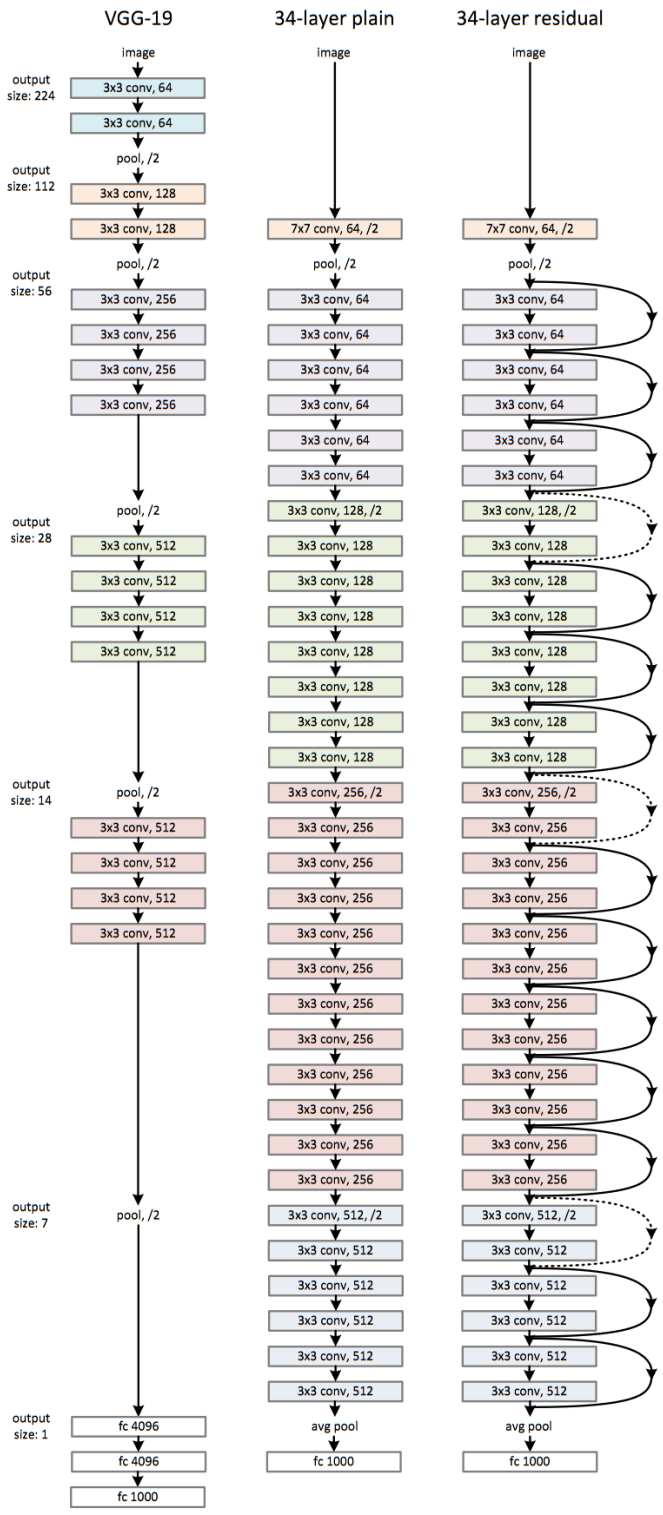

Figure 4.3.: **Left**: the VGG19 model as a reference. **Middle**: a plain network with 34 parameters layers. **Right**: a Residual Network with 34 parameter layers. The dotted shorcuts increase dimensions.

5. Hashing for Dimensionality Reduction

5.1. Hashing into binary codes

As detailed in chapter 3, images can be represented with small binary codes (also known as hash codes), which can be compared with a Hamming distance. Because binary codes are small and easy to handle on computers, millions of them can be stored in a database. Furthermore, the whole database can fit in memory, which leads to less IO operations. As explained in chapter 2, the Hamming distance can be computed very quickly even on commodity hardware thanks to the POPCOUNT instruction.

To extract a representation of an image in the form of a binary code, we can use a perceptual hash function. Formally a perceptual hash function maps high-dimensional inputs $x \in \mathbb{R}^p$, into binary codes, $y \in \mathbb{H} \equiv \{0,1\}^q$, and is defined as: $y = h(x)$. A suitable hash function should map similar (resp. dissimilar) images into similar (resp. dissimilar) binary codes according to a selected distance.

In the next two sections, we review several methods classified in two categories: data independent approaches, which make no prior assumption about the data distribution, and machine learning approaches, which take advantage of the data distribution to optimize the hash function. For each method we will describe the hash function and its associated similarity measure. The hash function determines the partitioning of the input space, which is important to consider. A linear hash function might have more difficulties to preserve similarities of a data set when compared to a non-linear hash function. Non-linear hash functions can more finely discriminate points that are not linearly separable. Moreover non-linear hash functions can discover the internal model of the input space to be able to represent it in a more efficient manner. Hence the binary codes can be shorter without necessarily being less efficient.

There exist different types of hash function: linear transformations, which will be described later, linear transformations followed by a cosine transform, functions based on kernels, functions based on multilayer neural networks, and so on.

One popular hash function is a linear transformation $h_W(x) = thr(Wx)$ where $W \in \mathbb{R}^{q \times p}$ and $thr(\cdot)$ is an element-wise Heaviside function. Figure 5.1 shows the partitioning of the space for this particular function. Each row of W denotes the

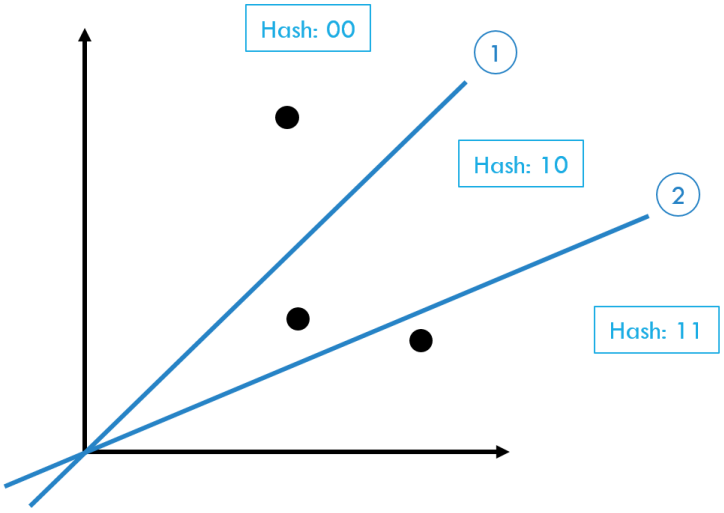

Figure 5.1.: Partitioning of the space for a hash function based on a linear transformation

normal of a hyperplane in the input space and determines the value of one bit; 1 is assigned to points on one side of the hyperplane and 0 to points on the other side.

The most popular similarity measure is based on Hamming distance. Other options are possible: Weighted Hamming distance, Jaccard coefficient, precomputed lookup table, and so on.

Depending on the application and the dimensionality of the inputs and outputs, the complexity of the computation of a hash function can be unacceptable. For example, the computation of q-bit binary codes for a p-dimensional input using a linear transform has a complexity of $O(qp)$. More expressive hash functions can have a higher complexity. Thus, it is important to carefully select the hash function according to the application.

5.2. Data independent approaches

In this section, we present the data independent approaches, which makes no prior assumption about the data distribution. The most popular approach is based on Locality Sensitive Hashing (LSH) [WSSJ14]. The general idea of LSH is to hash similar input items to the same binary code with higher probability than dissimilar input items. In other words, the goal of LSH is to maximize collisions rather than to avoid them. LSH comes with theoretical guarantees that a specific metric is increasingly well-preserved as the code length increases.

There are different LSH schemes that can preserve different distances: l_p distance, angular distance, Hamming distance, and so on. We will present LSH with random

projection which preserves the angular distance.

5.2.1. Random projection

Locality Sensitive Hashing based on random projection [Cha02] is designed for the angular distance. Thus, it preserves the cosine similarity. The angular distance between two vectors can be computed with

$$s_{cosine}(x, y) = \cos\theta(x, y) = \frac{x \cdot y}{\|x\|_2 \|y\|_2}$$

$$d_{cosine}(x, y) = \frac{\cos^{-1} s_{cosine}(x, y)}{\pi}$$

The hash function is defined as $h_W(x) = thr(Wx)$. Each coefficient of W is randomly chosen from a Gaussian distribution. Then the probability of collision is

$$\mathbb{P}(h_W(x) = h_W(y)) = 1 - distance = 1 - \frac{\theta(x, y)}{\pi}$$

It is possible to choose the length of binary codes according to the number of input vectors to hash. According to [WSSJ14], for n vectors, take $O(\log^2 n)$.

5.3. Machine Learning approaches

In this section, we present the machine learning approach, which takes advantage of the data distribution to optimize the hash function. Two different approaches are detailed, Semantic Hashing is based on an autoencoder, Minimal Loss Hashing and Triplet Loss Ranking, are based on the minimization of a loss function.

In addition to the two main components, hash function and similarity measure, machine learning solutions have an optimization criterion, which defines the objective that the algorithm pursue. A basic optimization criterion is the order-preserving criterion where the results of the ANN search are compared to an external reference. The similarity alignment criterion minimizes the difference between the similarities computed in output and input space. The coding consistent criterion encourages similar (resp. dissimilar) codes when the inputs are similar (resp. dissimilar) and penalizes dissimilar (resp. similar) codes when the inputs are similar (resp. dissimilar). The coding balance criterion aims to uniformly distribute the input vectors among the hash buckets.

In approaches based on machine learning, the parameters of the hash function are optimized using the input data. As usually in machine learning, it is possible to learn the model on a training data set and then use it with other data. A better way to proceed is to use all data to train the model. In this case, over-fitting is not an issue because there is no need to generalize to new data.

While presenting the approaches, we only consider the hash function, the similarity measure and the optimization criterion, without spending too much time on the learning algorithm, assuming that a good solution is found.

5.3.1. Semantic Hashing

Semantic Hashing [SH07],[SH09] is an unsupervised learning approach based on Neural Networks, in fact no similarity information is used. A multilayer autoencoder is used to discover a lower dimensional representation of the input vectors. Its architecture is shown in figure 5.2. The code layer, which is in the middle of the autoencoder, is forced to use a small number of binary variables (e.g. 32). The model is first pretrained as a stack of RBM's, then it is unrolled to create a multilayer autoencoder, which is fine-tuned by minimizing the root mean squared reconstruction error with backpropagation. To get the final binary code, the activations of the units in the code layer are simply thresholded.

With this framework, semantically similar documents are mapped to similar binary codes. Thus, we can use a Hamming distance to compare two binary codes. To perform an approximate nearest neighbor search, rather small codes (e.g. 20 bits) are used and a hash table lookup method is performed with a small radius (e.g. 4 bits). To improve the precision, it is possible to use a two-stage method. At first, a list of candidates is obtained with 28 bits codes, it is then pruned with 256 bits codes. For the code layer, two technical solutions are available.

On the one hand, logistic units are trained with deterministic Gaussian noise. The noise doesn't change during the training, and because there are more examples than parameters in the model, it is forced to generalize rather than to tailor the parameters to the beforehand fixed noise. The presence of noise forces the input of the activation function to be large and negative for some training cases and large and positive for others. Consequently, the noise has only a small impact on the output of the layer.

On the other hand, an easier approach discovered later [KH11]. Instead of logistic units, binary stochastic units are used in the code layer and no noise is added. A binary stochastic unit outputs either 1 or 0 otherwise depending on a probability given by a logistic function.

$$z_i = b_i + \sum_j s_i w_{ij}$$

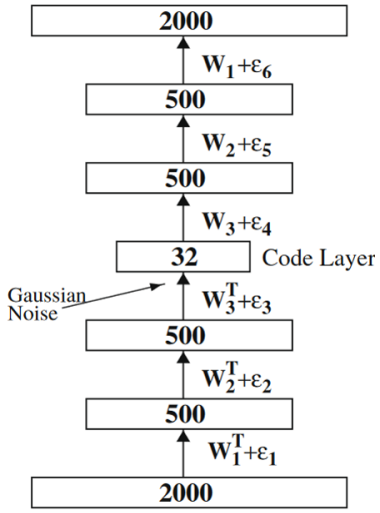

Figure 5.2.: Autoencoder in Semantic Hashing

$$\mathbb{P}(s_i = 1) = \frac{1}{1 + \mathrm{e}^{-z_i}} \qquad \mathbb{P}(s_i = 0) = 1 - \mathbb{P}(s_i = 1)$$

To train a binary stochastic unit we pretend during the backpropagation, that the output value comes from a normal logistic unit and it gives a smooth gradient for the backpropagation.

5.3.2. Minimal Loss Hashing

Minimal Loss Hashing (MLH) [NF11],[Nor16] is a supervised approach based on a hinge-like loss function. A linear hash function is used $h_W(x) = thr(Wx)$, it is also possible to use it with other hash functions as long as they are differentiable with respect to their parameters, so that we can compute the Jacobian. The Hamming distance is used to compare the output binary codes.

The learning is based on a training data set of labeled pairs (x_i, x_i', s_i) where x_i and x_i' are p-dimensional centered training points and s_i is a similarity label, whose value is 1 if x_i and x_i' are similar and 0 if dissimilar. To preserve a specific metric one can compute the similarity label by thresholding the pairwise distance. Alternatively, to preserve the semantic similarity, one can assign a similarity label based on the class label.

For each sample in the training data set, a loss function $l_{pair} : \{0,1\}^q \times \{0,1\}^q \times \{0,1\} \rightarrow \mathbb{R}^+$ assesses the quality of the mapping by assigning a cost to a pair of binary codes and a similarity label. By minimizing the loss over all training examples, we can learn the parameters of the hash function.

$$L(w) = \sum_{i=1}^{n} l_{pair}(h_W(x_i), h_W(x_i'), s_i)$$

To optimize the parameters of the hash function, the key point of MLH is to use a coding consistent criterion based on a hinge-like loss function. A hyper-parameter ρ is a threshold of distance in Hamming space, that defines the boundary between similar and dissimilar codes. If the distance between two codes is less than (resp. more than) ρ we can consider them as similar (resp. dissimilar). In addition, the λ parameter adjusts the penalty incurred for dissimilar pairs when they are too close in comparison to the penalty incurred for similar pairs when they are too far from each other. Although it is a slow process, a good value for these hyper parameters can be found with a validation set. Let $\|h-h'\|_H$ be the Hamming distance between binary codes h and h'. The pairwise loss based on the hinge function is defined as:

$$l_{hinge}(h, h', s) = \begin{cases} \max(\|h - h'\|_H - \rho + 1, 0) & \text{for } s = 1 \\ \lambda \max(\rho - \|h - h'\|_H + 1, 0) & \text{for } s = 0 \end{cases}$$

Minimal Loss Hashing builds a convex-concave upper bound on it. Then a stochastic gradient-based approach is used to minimize the loss function. The learning algorithm is online, scales well to large data sets and large code lengths.

5.3.3. Triplet Ranking Loss

Triplet Ranking Loss (TRL) [NFS12],[Nor16] is a supervised approach that generalizes Minimal Loss Hashing. While certainly good at learning a metric structure (e.g. Euclidean, Cosine) of the input data, the goal of this approach is rather to preserve the semantic similarity. When no pairwise similarity is available, one could rely on a generic distance (e.g. Euclidean) but this often produce unsatisfactory results. TRL, on the other hand, is based on the relative similarity between a triplet of points. Thus, no pairwise similarity label is required.

A linear hash function is used $h_W(x) = thr(Wx)$, as with MLH it is possible to use other hash functions as long as they are differentiable with respect to their parameters. The Hamming distance is used to compare the output binary codes.

The learning is based on a training data set of triplets of p-dimensional centered training points (x, x^+, x^-) such that the pair (x, x^+) is more similar than the pair (x, x^-). To build a training data set that preserves semantic similarity one can pick two points from the same class and one point from another class.

As with MLH, for each sample in the training data set, a loss function $l_{rank} : \{0,1\}^q \times \{0,1\}^q \times \{0,1\}^q$ assesses the quality of the mapping by assigning a cost

to a triplet of binary codes. The loss function should penalize the examples where $h_W(x_i)$ is closer to $h_W(x_i^-)$ than to $h_W(x_i^+)$ in Hamming distance. By minimizing the loss over all training examples, we can learn the parameters of the hash function.

$$L(w) = \sum_{i=1}^{n} l_{rank}(h_W(x_i), h_W(x_i^+), h_W(x_i^-))$$

To optimize the parameters of the hash function, TRL uses a hinge-like loss function. On the contrary of MLH, there is no hyper parameter, which makes the learning easier because there is no need to find them a good value with a validation set. The loss is zero when $h_W(x_i)$ is closer to $h_W(x_i^+)$ than to $h_W(x_i^-)$ in Hamming distance. Let $\|h - h'\|_H$ be the Hamming distance between binary codes h and h'. In this case, the loss is zero when $\|h - h^+\|_H$ is at least one bit smaller than $\|h - h^-\|_H$. The triplet loss based on the hinge function is defined as:

$$l_{rank}(h, h^+, h^-) = \max(\|h - h^+\|_H - \|h - h^-\|_H + 1, 0)$$

The same optimization technique as MLH is used. A convex-concave upper bound is built on the loss function. Then a stochastic gradient-based approach is used to minimize the loss function. TRL is more flexible than the pairwise hinge loss and is shown to produce superior hash functions. A simple kNN classifier on the learned binary codes is competitive with state-of-the-art classifiers on CIFAR and MNIST.

5.4. Comparison

In this section, we compare the methods in term of hash functions, partitioning of the input space and retrieval performance. We don't take in account the optimization technique of the machine learning approaches, assuming that a good solution is found.

Fundamentally, data independent approaches should be less efficient than machine learning approaches because they don't take in account the data distribution. This means that on the same instance, depending on the data distribution, the machine learning solutions should generally yield better results. The binary codes generated by data independent approaches should be longer than their equivalent from machine learning approaches for the same retrieval accuracy. The data independent solutions are nevertheless more general and come with theoretical guarantees that a specific metric is increasingly well preserved as the code length increases. Moreover, one has to be very careful when training a machine learning approach because there is many pitfalls: trying to generalize from a bad sampled training data set, making an error during the preprocessing, choosing wrong values for the hyper parameters,

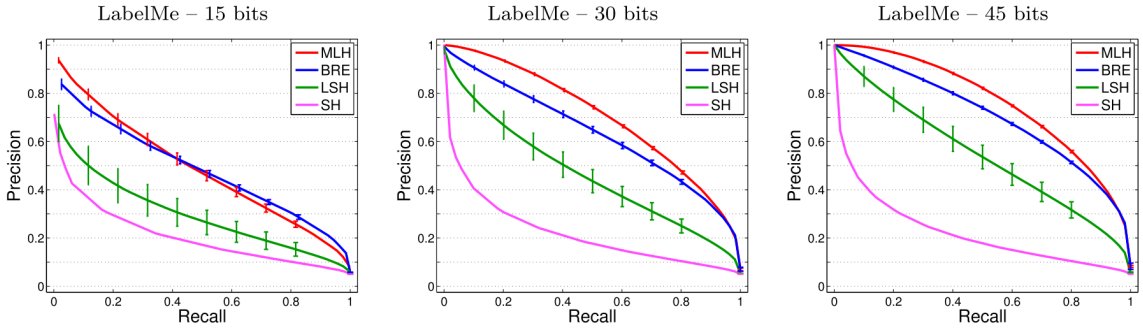

Figure 5.3.: Precision-recall curves on LabelMe for different methods for different code lengths. [NF11]

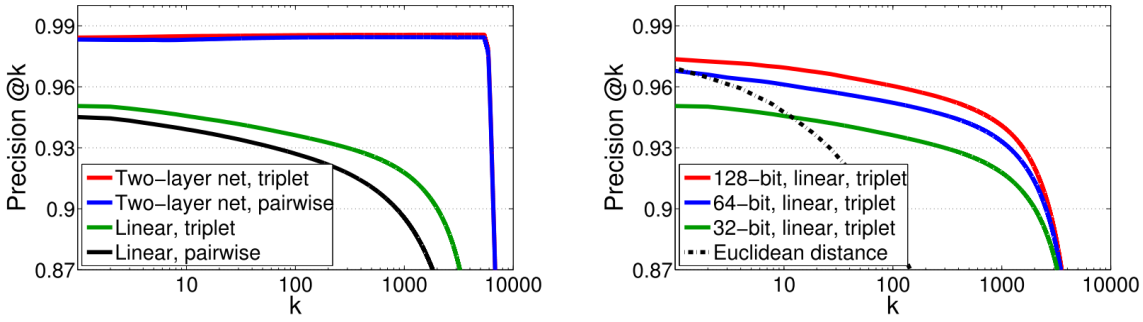

Figure 5.4.: MNIST precision@k: (left) four methods with 32-bit codes; (right) three code lengths with triplet loss. [NFS12]

choosing, a priori, a hash function that is not appropriate, training only one model and not benchmarking it against other approaches, and so on.

Based on experiments conducted by the authors of the approaches we can have an idea on their relative performances.

A first experiment, conducted by the authors of Minimal Loss Hashing (MLH) [NF11], compares it with LSH and other methods. Six data sets are used: *Photo-tourism*, a collection of images represented as 128D SIFT features, *LabelMe* and *Peekaboom*, collections of images represented as 512D Gist descriptors. *MNIST*, a collection of 784D grayscale images of handwritten digits, *Nursery* with 8D features, a synthetic data set comprising uniformly sampled points from a 10D hypercube. All methods used identical training and test sets. The similarity labels are based on a thresholded Euclidean distance so that each point has, on average, 50 neighbors. On each data set, input vectors are mean-centered. On all but the 10D Uniform data set, each datum is normalized. Some methods improve with dimensionality reduction so a PCA is applied to retain 40 dimensions (except for 10D Uniform and 8D Nursery). Average and standard deviation of precision/recall are computed on 10 executions of all methods, except for Spectral Hashing (SH). In this review we only present the precision-recall curves on LabelMe for different code lengths, in figure 5.3. Other experiments are conducted in the original paper: precision using

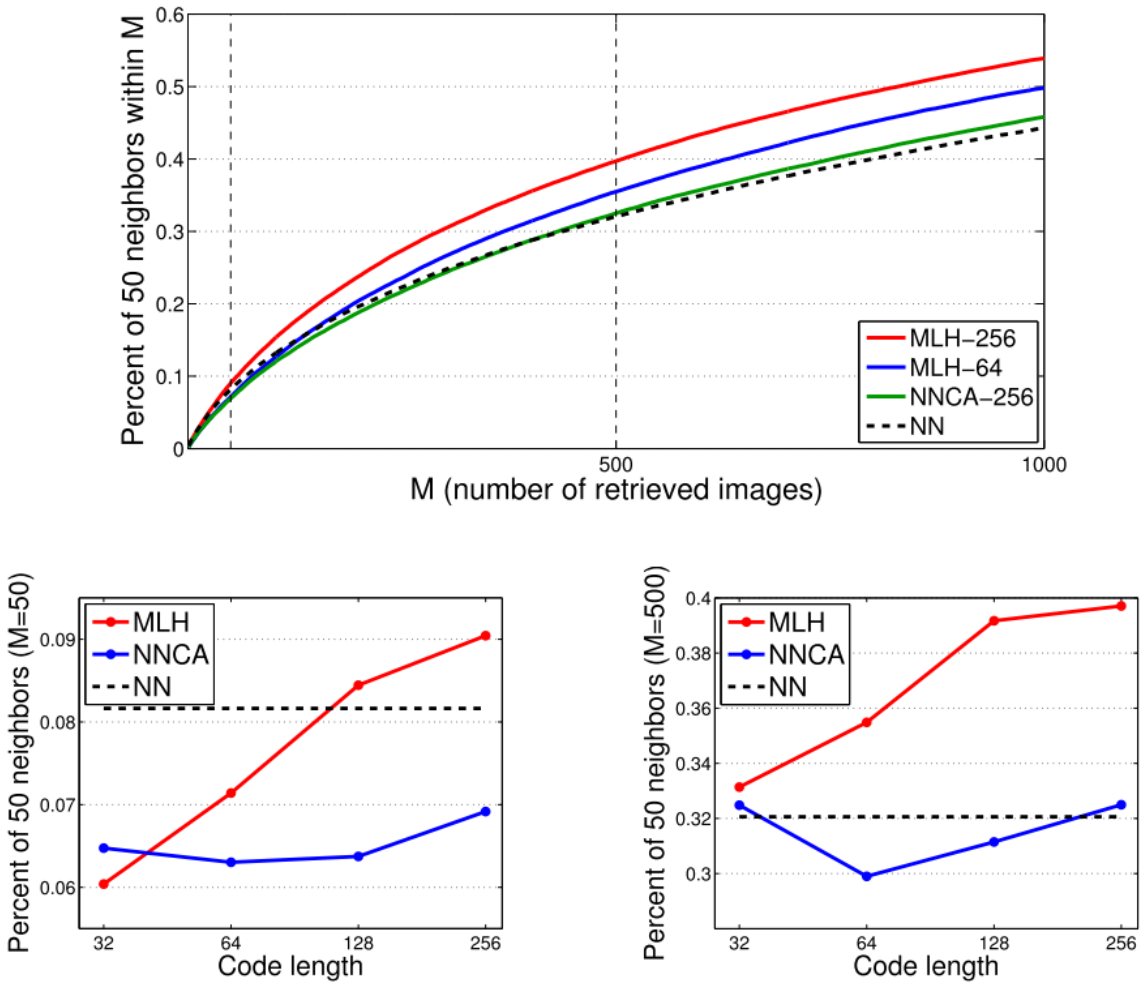

Figure 5.5.: (top) Percentage of 50 ground-truth neighbors as a function of number of images retrieved ($0 \leq M \leq 1000$) for MLH with 64, 256 bits, and for NNCA with 256 bits. (bottom) Percentage of 50 neighbors retrieved as a function of code length for $M = 50$ and $M = 500$. [NF11]

a Hamming radius of 3 bits for different methods as a function of code lengths, precision-recall curves for different methods for different code lengths. In almost all cases, the performance of MLH is clearly superior to all others (including LSH).

A second experiment, conducted by the authors of Triplet Loss Ranking (TRL) [NFS12], compares it with MLH. Two data sets are used, MNIST and CIFAR-10. Similarity labels are derived from class labels: items from the same class are similar. In this review we only present the precision@k (fraction of same-class items from a kNN retrieval) on the MNIST data set for different hash functions, different optimization criteria and for different values of k. The results are plotted in figure 5.4. The hash function is either a linear transformation or a two-layer neural network. The optimization method is either MLH or TRL. K varies from 1 to 10,000. A

Euclidean nearest neighbor search is also included as baseline because it provides an upper bound on the performance of methods that preserve Euclidean distances (e.g. LSH). The conclusion is that TRL yields slightly better performances than MLH.

A third experiment, conducted by the authors of Minimal Loss Hashing (MLH) [NF11], compares it with NNCA, which is based on Semantic Hashing. The experiment makes use of the 22K LabelMe data set with a semantic pairwise affinity matrix provided by humans. As a consequence, the similarity is based on semantic content rather than on Euclidean distance. At the time this experiment was conducted, NNCA was considered as the superior method for this task. MLH and NNCA are trained, using varying code lengths, on 512D Gist features using the semantic labels. A nearest neighbor baseline using cosine similarity is added because it is a bound for LSH as it mimics Euclidean distance, which is worse than cosine similarity in Gist space. As shown in figure 5.5, MLH and NNCA exhibit similar performance for 32-bit codes, but for longer codes MLH is superior.

According to these experiments, we have a rough idea on what are the relative performances of the previously described methods. MLH is clearly superior to LSH, Semantic Hashing appears to be less efficient than MLH. It is hard to compare Semantic Hashing with LSH because no experiment compares them directly. TRL is slightly better than MLH. The conclusion is however not perfectly reliable because the methods are not all compared at the same time. Moreover, it is based on experiments conducted by the authors of the methods.

MLH and TRL are the two best methods reviewed in this paper. Even if TRL is a generalization of MLH and appears to be slightly better, the two approaches have different use cases. On the one hand, MLH can preserve a metric structure with a pairwise similarity and ρ allows one to search by range for similar images. On the other hand, TRL is well suited to the preservation of semantic similarity and since ranking is kept, one can perform a kNN to search for similar images.

6. Search in Hamming Space

6.1. Exhaustive Sequential Search

Because binary codes are storage efficient and comparisons are fast, a basic approach is to perform an exhaustive search by computing the distances to all codes in the database. This can be executed on processors and graphics processing units, and distributed on multiple computers.

Processors are well suited when all binary codes in the database fit in the cache. Indeed, the popcount instruction is so fast that the search is essentially limited by the throughput of the RAM. Moreover, if the search is executed on multiples cores cache miss problems can arise. Thus, memory bandwidth and cache are both performance factors. Therefore, processors are efficient for exhaustive sequential search up to about 5 million binary codes.

Graphics processing units (GPU) have higher computational power and memory bandwidth than processors. Moreover, the popcount instruction is also available. A certain number of binary codes is necessary to reach the best performance. Therefore, GPU are efficient for exhaustive sequential search starting from about 5 million binary codes.

6.2. Hash Table

The hash table lookup consists in creating a hash table on the binary codes. To perform a nearest neighbor search, we can retrieve the content of the hash buckets in the vicinity of a query code. The complexity, in this case, only depends on the length of binary codes and the search radius. For binary codes of q bits, the number of distinct hash buckets to examine is [Nor16]

$$V(q, r) = \sum_{z=0}^{r} \binom{q}{z}$$

This approach is suited only for small distances because the number of hash buckets within a Hamming ball around a query grows almost exponentially with the search radius. For example, with 32 bits codes and a radius of 8 bits, about 15 million

hash buckets are to explore, which is absurd if there are less than 15 million binary codes in the database, because an exhaustive sequential search would be faster. When binary codes are longer than 32 bits, which is often the case, the hash table approach is no more efficient.

A recent approach based on the idea of hash table lookup is Multi-Index Hashing (MIH) [NPF12], [Nor16], [GV16]. It consists in building multiple hash tables on binary code sub-strings and yield to sub-linear run-time behavior for uniformly distributed codes.

6.3. Comparison

In this section, we compare the performances of 3 methods: exhaustive sequential search with CPU and GPU, and Multi-Index Hashing. We implemented exhaustive sequential search on CPU with OpenMP [1] and on GPU with Thrust [2]. We use the implementation of Multi-Index Hashing from [NPF12] [3].

We compared the methods using the following protocol. A list of 64 bits binary codes is randomly generated with a uniform distribution. For each method, 10,000 fixed-radius nearest neighbor searches (with a radius fixed to 8) are performed with the first codes in the previously generated list. We measure the time to perform all queries along with the time to load data. We also check that all 3 methods are returning the same results.

Our benchmark is available on GitHub [4]. To get an idea of the actual performances we recommend running it directly on the concerned hardware. On commodity hardware, a Intel i5 Ivy Bridge and a GTX 750 Ti, for a large number of hashes (at least 50M) the MIH solution is the most efficient, followed by the GPU and finally the CPU (up to about 5M hashes). Depending on the hardware these results can vary. We compared the three methods on a server equipped with two Intel Xeon E5-2630 v3 and a Tesla K80 (we used only 1 of the 2 GPU), sequential search on the CPUs seems to be always better than GPU. MIH was better than sequential search on the CPU starting from about 100M hashes.

Sometimes, raw performance is not as important as power efficiency. In this case, one might want to compare the price-performance ratio. MIH only runs on one CPU core and is therefore the most efficient if the database contains enough hashes. Sequential search on CPU and GPU use all available cores and consume a lot of power, this is why they are less energy efficient.

[1] http://www.openmp.org
[2] https://developer.nvidia.com/thrust
[3] https://github.com/norouzi/mih
[4] https://github.com/mgaillard/HashSearch

Part II.

Contribution

7. Benchmarking

In a previous work [GE17], we designed a new framework to benchmark reverse image search engines. Our approach is based on the evaluation measures of information retrieval systems described in [MRS08]. We model the reverse image search engine as an information retrieval system that returns an unranked set of documents for a query. If many documents are retrieved, the user is in charge of choosing the one that best suit his needs.

7.1. Metrics

7.1.1. Effectiveness

To process a query, the reverse image search engine order the indexed images by relevance. Results are retrieved using a fixed-radius nearest neighbor search. Thus, each image, whether relevant for the query or not, can be retrieved or not. This notion can be made clear by examining the following contingency table 7.1.

Table 7.1.: Contindency table for information retrieval from [MRS08]

	Relevant	Non-relevant
Retrieved	True positive (tp)	False positive (fp)
Not retrieved	False negative (fn)	True negative (tn)

The effectiveness of the system is measured with the following metrics: Precision (P) is the fraction of retrieved documents that are relevant,

$$precision = \mathbb{P}(\text{relevant}|\text{retrieved}) = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = \frac{tp}{tp + fp}$$

Recall (R) is the fraction of relevant documents that are retrieved.

$$recall = \mathbb{P}(\text{retrieved}|\text{relevant}) = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = \frac{tp}{tp + fn}$$

A single measure that trades off precision versus recall is the F-measure, which is the weighted harmonic mean of precision and recall:

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \qquad F_{\beta=1} = \frac{2PR}{P + R}$$

It is possible to change the weights in the harmonic mean of the F measure in order to tune it. This is done by changing the β parameter. Values of $\beta < 1$ emphasize precision, while values of $\beta > 1$ emphasize recall. This is important in order to benchmark the system in accordance with its application.

7.1.2. Performance

It is important to measure the time the system need for indexing and searching. Indeed, the system should be able to index millions of images and search across them as fast as possible. The complexity of both the indexing and searching phases depends on the number of images and is not necessarily linear. Therefore, we propose to measure the indexing and searching time for a certain number of images.

7.2. Protocol

In order to compare the effectiveness and the speed of different image search methods, we created a comparison protocol. We choose 25,000 images from the MIR-FLICKR dataset[1].

To be able to calculate automatically the precision and recall of the results, we applied 6 small modifications on each image, that gave us a dataset with 175 000 images. We measured the index and search speed as well as the results precision and recall.

The modifications (illustrated on Figure 7.1) applied on the images are:

1. Gaussian blur ($r = 4$, $\Sigma = 2$)
2. Black and white transformation
3. Resize to half height and width
4. JPEG compression with $quality = 10$
5. Clockwise rotation by 5 degrees
6. Crop by 10% at the right side of the image.

Figure 7.1.: Illustration of the image modifications.

These modifications represent the basic cases of small changes images usually undergo over the web. The benchmark is programmed so that it is easy to change the set of modifications.

Our benchmark protocol for a generic reverse image search engine is detailed in this section.

1. Select $N + M$ images that are representative to an application with no duplicated images. In our case $N = 24,000$ and $M = 1,000$ (the first 1,000 images from the dataset in alphabetical order)
2. Split them into 2 sets of N base images and M non-indexed images.
3. Select K modifications and from the N base images, generate K new image sets containing $K \times N$ modified images. In our case $K = 6$, and the modifications are those enumerated above.
4. Index the N base images and the $K \times N$ modified images according to the image representation method.
5. Make search queries with:
   a) The M images from the non-indexed image set,

[1]http://press.liacs.nl/mirflickr

b) The N images from the base image set.

c) The $K \times N$ images from the modified sets.

6. Analyse the search results and compute the mean precision, the mean recall and the mean F-measure of all queries.

a) For the M images of the non-indexed set, there should be no relevant result image. Thus, the result should be empty.

b) When querying with one of the other $(K+1) \times N$ already indexed images, the relevant results are the $K + 1$ images that are the modified version of the query image. Thus, the result of each query should contain K+1 images that come from the same base image as the query image.

Because our system use a fixed-radius nearest neighbor search, we model it as an information retrieval system that returns an unranked set of documents. To determine the best radius, we repeat this protocol for several different radii.

It is possible to use the protocol in two different ways. Firstly, by applying all modifications to have an overview of the performances of a method. Thus, it is possible to compare two methods according to a complete workload. Secondly, by applying a single modification, for example to identify the strength and weakness of a method according to some modifications.

We implemented the protocol [2] as Linux shell commands and C++, using online available libraries.

7.3. Results

We evaluate the methods described in chapter 3: DCT, MH and RV based perceptual hash functions from [Zau10].

7.3.1. Retrieval performance against single modification

We index the N base images and then make K search queries each with one of the N modified images. We compute the F-measure for various radius and take the maximum.

The functions are robust against Gaussian blur ($r = 4$, $\Sigma = 2$), JPEG compression (quality 10%), grayscale filter, and scale to half the size. However the functions are not robust against crop (10% on the right) and rotate (5 degrees clockwise) modifications as illustrated on Figure 7.2.

[2] See our implementation on: https://github.com/mgaillard/pHashRis

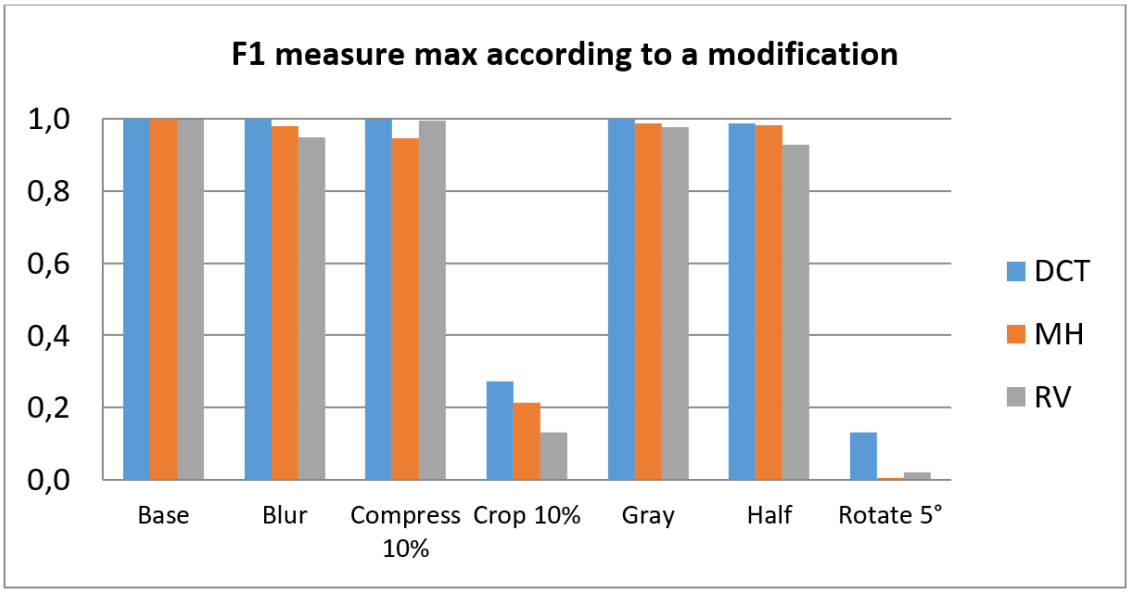

Figure 7.2.: Maximum F1-measure of DCT, MH and RV perceptual hash functions against single modifications.

We also tested, the evolution of our accuracy measures with different degrees of the modifications. We noticed that the different hash methods were quite sensible to rotations (above 2 degrees) or to cropping (above 5%), but resisted very well to compression, blur or resize.

7.3.2. Retrieval performance against all modifications

To evaluate the speed and retrieval performance of a reverse image search engine, we run the protocol described in the previous section. The results are in figures 7.4, 7.5, 7.6. We can see that all functions have equivalent retrieval performances with a maximum F-measure of about 60

The results (table 7.3) showed that the DCT based hash function is clearly faster than the Marr-Hildert (MH) Operator and Radial Variance (RV) based hash functions. It is also more accurate against the 6 chosen modifications. We evaluate the retrieval performance, calculating the mean precision, recall and F-measure of the returned results. The calculation is repeated for different radius values. The radius here is a normalized distance value, below which two images are considered as similar.

The performance is evaluated through the index and search speed. Table 7.3 presents the measurements done for 125,000 images on an OVH dedicated virtual machine equipped with an Intel Xeon Haswell 8 cores at 3.1 GHz and 30 GB of RAM.

	Index 125k images	125k queries on 125k images
DCT	3 min 55 sec	3 min 59 sec
MH	31 min 13 sec	32 min 20 sec
RV	1 min 48 sec	3h 47 min 31 sec

Figure 7.3.: indexing and search time measures for the DCT, MH and RV based perceptual hash functions

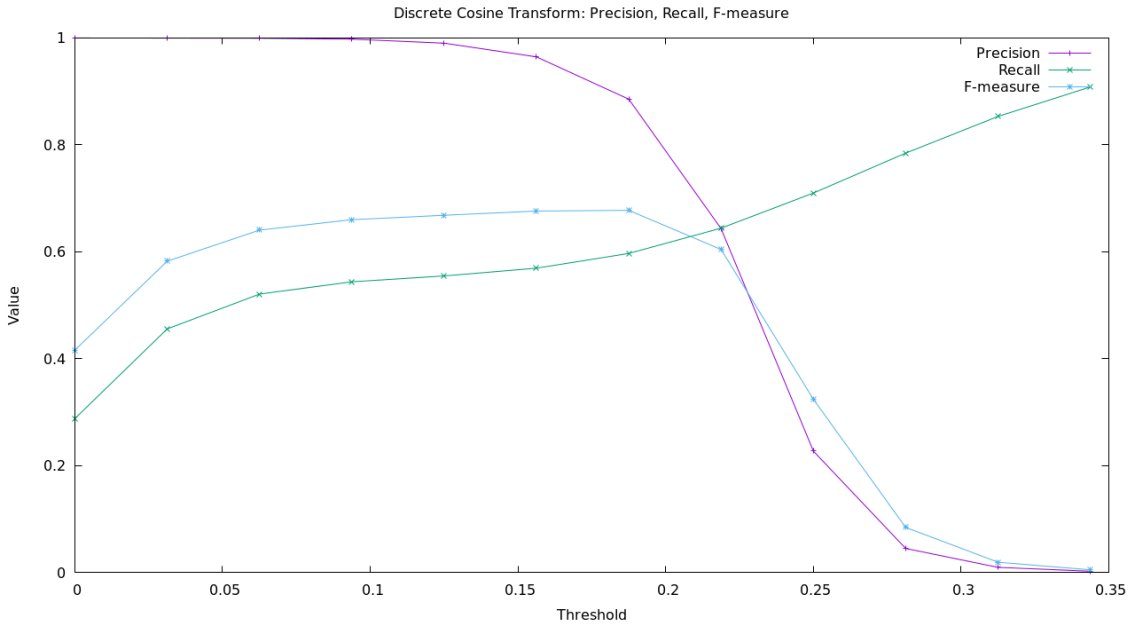

Figure 7.4.: Precision/Recall/F-measure curves of the DCT based perceptual hash function search results according to different radius values

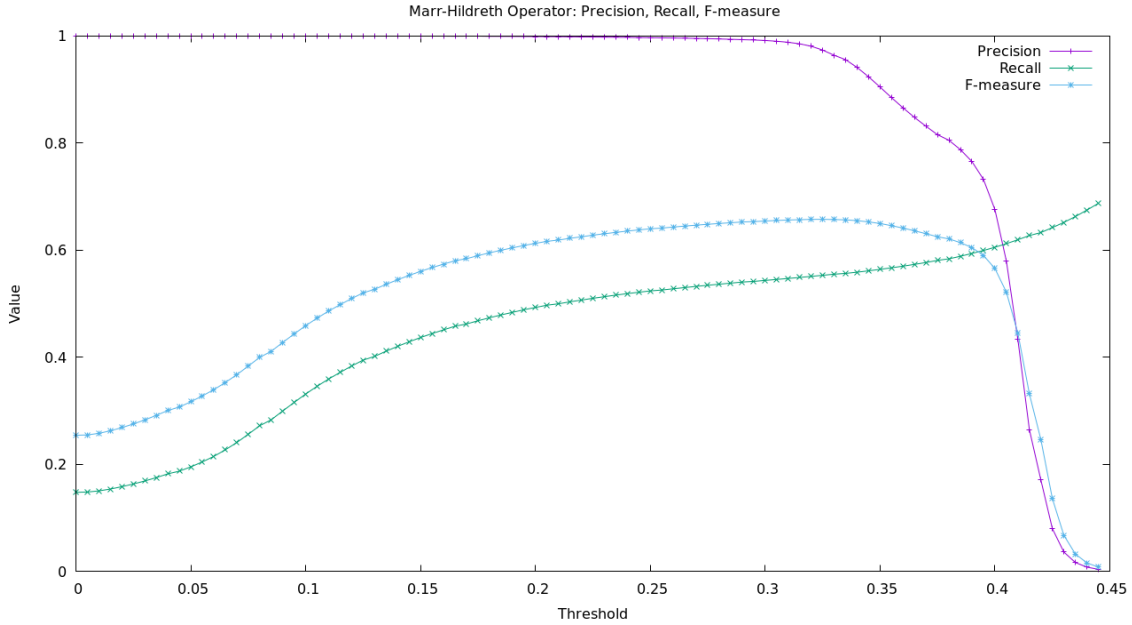

Figure 7.5.: Precision/Recall/F-measure curves of the Marr-Hilderth based perceptual hash function search results according to different radius values

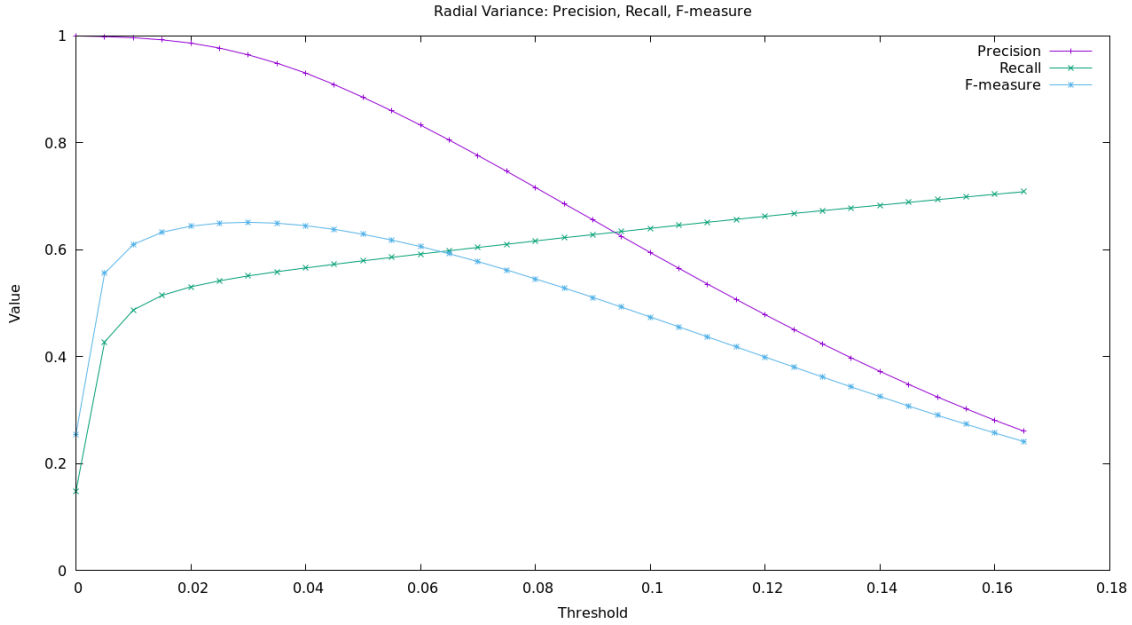

Figure 7.6.: Precision/Recall/F-measure curves of the Radial Variance based perceptual hash function search results according to different radius values

8. CNN Features Robustness

8.1. Introduction

In a previous chapter 4, we advocate the use of convolutional neural networks to extract features from images. In this chapter, we evaluate the robustness of CNN features against modifications. In a first time, instead of trying to learn an image representation from scratch, we assess the robustness of existing networks trained on an unrelated classification task. The aim being to find an already good representation, according to a specific distance, whose dimensionality would be reduced later. This step is important because we make the hypothesis that an already robust image representation will still stay more robust after a dimensionality reduction than a less robust image representation. Despite breaking our main problem into two sub problems (features extraction and dimensionality reduction) is certainly not optimal, depending on the dimensionality reduction method, it is possible to fine tune the image representation. It is for example possible with methods based on neural networks such as: Minimal Loss Hashing and Triplet Ranking Loss. By optimizing at the same time the image representation and the dimensionality reduction, the retrieval performance should probably be better.

8.2. Protocol

The protocol is inspired from our previous benchmark for Reverse Image Search described in chapter 7. It is based on a set of images that are modified according to some transformations. We then consider a base image and its modified versions as similar amongst themselves, and all other pairs of images as dissimilar. All images are queried and the average precision, recall and Fmeasure are computed over all queries.

8.2.1. Preparation

Following are the steps to prepare images for the benchmark.

1. Select N images that are representative to an application with no duplicated images. In our case, N is equal to either 200, 2500 or 25,000.

2. Select K modifications and from the N base images, generate K new image sets containing $K \times N$ modified images. In our case $K = 6$, and the modifications are those enumerated in section 7.2: blur, grayscale, resize, JPEG compression, rotation and cropping.

3. With a convolutional neural network, extract the features from all images. More details about this step are given in section 8.3.

8.2.2. Distribution of distances

To get a rough idea of the robustness of CNN features against modifications, we analyze the distribution of distances between similar and dissimilar images. It is also useful to know the distribution of distances for the RIS benchmark. In fact, we can infer an approximate value for the best radius separating similar and dissimilar images.

1. For each modification, compute the distance between each base image and its modified version. Take this distance in account in the computation of the minimum, first quartile, median, last quartile and maximum distance between the base and the modified image set. Draw a box plot with these data.

2. For each pair of different images in the base image set, compute the distance between them. Take this distance in account in the computation of the minimum, first quartile, median, last quartile and maximum distance between dissimilar images. Draw a box plot with these data.

The resulting box plots can be seen in the Result section 8.5 of this chapter. Ideally, the distribution of distances should not overlap. In other words, the distances between similar images must be strictly smaller than the distances between dissimilar images.

8.2.3. RIS benchmark

To get a precise idea of the retrieval performance using CNN features, we implement the same benchmark as in chapter 7. We compute all distances between all pairs of images and apply a threshold below which images are considered as similar. Then we compute the mean precision, the mean recall and the mean F-measure. The protocol is repeated for various threshold, the threshold that maximizes the F-measure can be considered as the best radius for the reverse image search system.

1. Make search queries with the $(K + 1) \times N$ images from the base and modified sets.

2. Analyse the search results and compute the mean precision, the mean recall and the mean F-measure of all queries.

    a) The relevant results are the $K + 1$ images that are the modified version of the query image. Thus, the result of each query should contain K+1 images that come from the same base image as the query image.

The resulting plots can be seen in the Result section 8.5 of this chapter. More detailed results for *VGG16_block5_pool_max* with Cosine distance are available in appendix B. Ideally, the maximum F-measure should be near to 1.

8.3. Models

In this section, we describe all models compared in the benchmark. To extract the features, we use a pretrained neural network to predict the label of an image. The feature vector consists of the activations of the units of one specific layer. If the layer is a convolutional or pooling layer, the results are 2D features maps. In this case, the activations are flattened using a global average or maximum pooling. We compare the models described in section 4.4: *VGG16*, *VGG19*, *ResNet50*, *InceptionV3*. We also evaluate *Xception*, which is also a model available in Keras. If needed the feature vectors can be normalized to unit length in Euclidean distance. To compare the feature vectors, two distances are used: Euclidean and Cosine.

In the following sections, we describe, for all models, all layers that we compared. The name of layers are the same as in the Keras library. For the models, the naming convention we use in the program is: [network]_[layer]. If the feature vector is normalized with L2 we add *_norm_l2* at the end. For example, *ResNet50_flatten_1_norm_l2* is extracted from the *flatten_1* layer of the *ResNet50* network and is normalized as unit length in Euclidean distance.

8.3.1. VGG

In VGG16 and VGG19, available layers ordered by increasing depth are:

- block{i}_pool_avg for $i \in \{3, 4, 5\}$
- block{i}_pool_max for $i \in \{3, 4, 5\}$
- flatten
- fc1
- fc2
- predictions

8.3.2. ResNet50

In ResNet50, available layers ordered by increasing depth are:

- activation_{i}_avg for $i \in \{4, 7, 10, 13, 16, 19, 22, 28, 31, 34, 37, 40, 43, 46\}$
- activation_{i}_max for $i \in \{4, 7, 10, 13, 16, 19, 22, 28, 31, 34, 37, 40, 43, 46\}$
- avg_pool_avg
- avg_pool_max
- flatten_1
- predictions

8.3.3. InceptionV3

In InceptionV3, available layers ordered by increasing depth are:

- mixed{i}_avg for $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- mixed{i}_max for $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- predictions

8.3.4. Xception

In Xception, available layers ordered by increasing depth are:

- add_{i}_avg for $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- add_{i}_max for $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- block14_sepconv1_act_avg
- block14_sepconv1_act_max
- block14_sepconv2_act_avg
- block14_sepconv2_act_max
- predictions

8.4. Implementation

Our implementation is on GitHub [1]. It consists of three main programs and a bash script to orchestrate their executions.

The first program is a Python script that extracts features from images using the neural network library Keras [C+15]. The feature vectors can be saved in HDF5 files, which are then read during the execution of the second and third programs.

The second program is the implementation of the distance benchmark. It computes statistics on the distribution of distances between similar and dissimilar images. Those are first written in a file and then used to generate a plot with gnuplot.

The third program is the RIS benchmark. It is written in C++ with OpenMP, to fasten the computation of distances in high dimensional spaces thanks to multi-threading and SIMD. This benchmark computes the mean precision, recall and F-measure of a RIS system based on the previously extracted features. The results are first written in a file and then used to generate a plot with gnuplot.

The workflow of the bash script is:

1. Images are modified according to the K modifications with ImageMagick [2].
2. Features are extracted from the $(K + 1) \times N$ images according to all possible models.
3. The distance benchmark is executed for all features according to Euclidean and Cosine distance.
4. The RIS benchmark is executed for all features according to Euclidean and Cosine distance.

8.5. Results

Because the benchmark is compute intensive, to find the best model we proceed in three steps. Firstly, we compare all models with all distances but only with 200 images. We retain only the models whose F-measure is above 90%. Secondly, we repeat this process with 2,500 images. The best models with 2500 images ordered by decreasing F-measure are:

- *VGG16_flatten* with Cosine distance; $Fmeasure = 0,952$
- *VGG19_flatten* with Cosine distance; $Fmeasure = 0,949$
- *ResNet50_activation_46_max* with Cosine distance; $Fmeasure = 0,948$

[1] https://github.com/mgaillard/CNNFeaturesRobustness
[2] www.imagemagick.org

- *ResNet50_activation_43_max* with Cosine distance; $Fmeasure = 0,941$
- *VGG16_block5_pool_max* with Cosine distance; $Fmeasure = 0,931$
- *VGG19_block5_pool_max* with Cosine distance; $Fmeasure = 0,929$
- *InceptionV3_mixed9_max* with Cosine distance; $Fmeasure = 0,918$
- *InceptionV3_mixed9_avg* with Cosine distance; $Fmeasure = 0,911$

Finally, we compare these models with all the 25,000 images from the MIRFLICKR collection.

8.5.1. Distribution of distances

In figure 8.1, we present the distribution of distances obtained with *VGG16_block5_pool_max* with Cosine distance. Only one model is presented because this benchmark is not as precise as the RIS benchmark, and is mainly used to get an idea of the value of the best radius. Moreover, all models have similar distribution of distances.

We can see that distances between base and modified images are statistically smaller than the distances between dissimilar images. The interquartile ranges are especially not overlapping, which means that it is possible to differentiate the majority of similar and dissimilar images. The median distance between dissimilar images is about 0.6. If one uses this value as radius for the nearest neighbor search, approximately half of all images would be retrieved. Depending on the modification, the distribution of distances is not the same. Distances between base images and images that are resized to 50% or cropped by 10% on the right are smaller than for other modifications. Consequently, we can say that features extracted with *VGG16_block5_pool_max* are more robust to resize and cropping than JPEG compression, grayscale filter, Gaussian blur and rotation.

8.5.2. RIS benchmark

Retrieval performance against all modifications

The results of the benchmark with all modifications are shown in table 8.1. Each line represents the maximum F-measure of each model along with the corresponding radius, precision and recall.

We can see that despite the fact that the F-measure decreases by about 3 points with 25,000 images compared to 2,500 images, the F-measures are still very good. For comparison, on the same benchmark, the DCT based perceptual hash function yields to a F-measure of about 0.6 (cf. section 7.3).

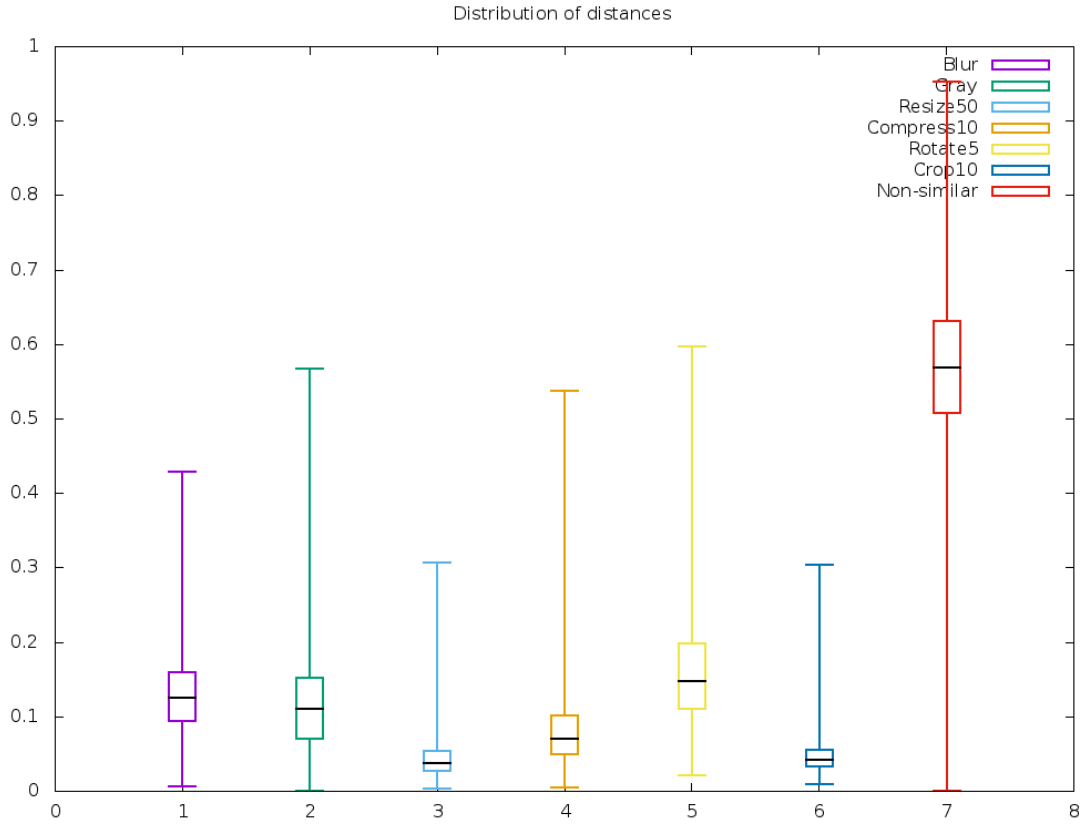

Figure 8.1.: Distribution of distances according to a modification obtained with *VGG16_block5_pool_max* with Cosine distance

Table 8.1.: Benchmark with all modifications on 25,000 images: maximum Fmeasure

Model	Dimensions	Distance	Radius	Precision	Recall	Fmeasure
VGG16_flatten	25088	cosine	0,51	0,954	0,943	0,936
VGG19_flatten	25088	cosine	0,50	0,955	0,934	0,931
ResNet50_activation_46_max	2048	cosine	0,17	0,968	0,913	0,928
ResNet50_activation_43_max	2048	cosine	0,14	0,952	0,919	0,921
VGG16_block5_pool_max	512	cosine	0,23	0,947	0,898	0,904
VGG19_block5_pool_max	512	cosine	0,23	0,948	0,892	0,902
InceptionV3_mixed9_max	2048	cosine	0,20	0,926	0,892	0,881
InceptionV3_mixed9_avg	2048	cosine	0,20	0,924	0,865	0,866

	Base	Blur	Gray	Resize	Compr	Rotate	Crop
VGG16_flatten	1,000	0,985	0,975	0,998	0,983	0,930	0,998
VGG19_flatten	1,000	0,984	0,971	0,998	0,982	0,920	0,998
ResNet50_activation_46_max	1,000	0,983	0,964	0,998	0,968	0,928	0,998
ResNet50_activation_43_max	1,000	0,984	0,955	0,998	0,966	0,927	0,999
VGG16_block5_pool_max	1,000	0,955	0,926	0,996	0,966	0,848	0,995
VGG19_block5_pool_max	1,000	0,954	0,926	0,996	0,966	0,846	0,995
InceptionV3_mixed9_max	1,000	0,910	0,953	0,994	0,917	0,857	0,994
InceptionV3_mixed9_avg	1,000	0,906	0,914	0,993	0,892	0,866	0,999

Table 8.2.: Maximum Fmeasure of CNN models against single modifications. Bar chart in figure 8.2.

Retrieval performance against single modification

The results of the benchmark with single modifications are shown in table 8.2 and in figure 8.2. Each bar represents the maximum F-measure of each model against a single modification. The radius at which the F-measure is maximum can vary depending on the model and the modification. Thus, this figure gives an idea of the best retrieval accuracy against a single modification. This is why individually the F-measure is better than in the benchmark against all modifications. Because in this latter, the radius is the same for all modifications.

We can see that all models work well and their F-measures are at least above 0.8. All models are very robust against resize to half-size and cropping 10% on the right, with a F-measure greater than 0.99. The robustness against Gaussian blur, grayscale filter and JPEG compression is good, with a F-measure greater than 0.9. Rotation is the harder modification, with a F-measure of about 0.85 for *VGG16_block5_pool_max*, *VGG19_block5_pool_max*, *InceptionV3_mixed9_max* and *InceptionV3_mixed9_avg*, and about 0.92 for the others. Models based on *ResNet50* yields to the best results for all modifications.

8.5.3. Conclusion

The objective of this chapter is to find an image representation that is robust against modifications. In chapter 7, we compare different perceptual hash functions, which can be used as image representation, and show that they are not robust against cropping and rotation. In this chapter, we compare features extracted with off the shelf convolutional neural networks and find a set of models that have very good retrieval performances against modifications. Especially, these are quite robust against rotation and very robust against cropping.

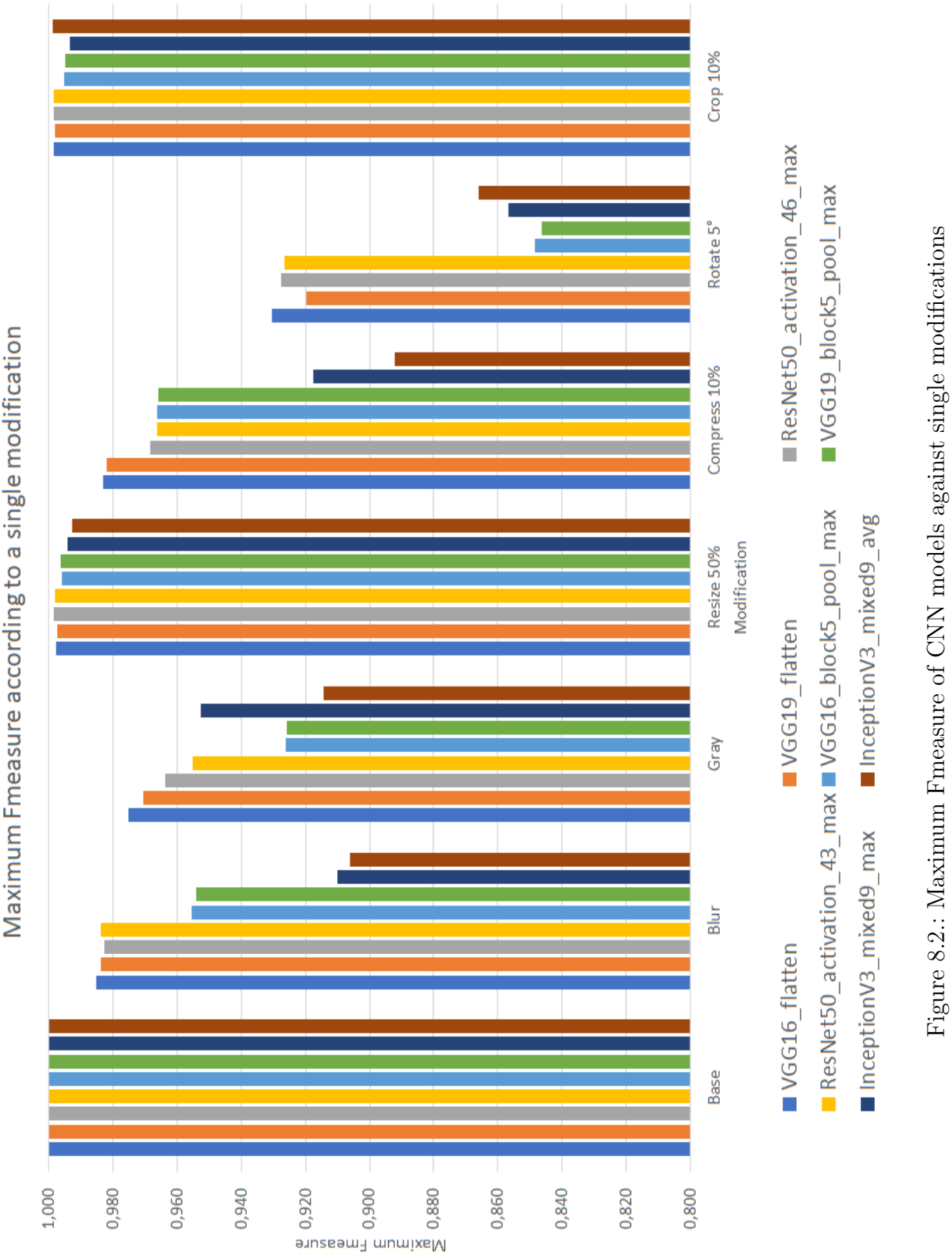

Figure 8.2.: Maximum Fmeasure of CNN models against single modifications

Robustness is certainly caused by preprocessing and data augmentation during training. Because the input of the convolutional neural network has always a constant size, images are first resized, thus features are robust against scaling. During training, it is common to generate more samples by transforming original images. For example, during the training of VGG [SZ14], to obtain the fixed size input images, training images are randomly cropped from rescaled training images. Therefore, the neural network is forced to learn a representation that is robust against cropping. More generally, by choosing the modifications made during the data augmentation step, one could give the neural network the ability to be robust against these modifications.

Two different distances are used to compare feature vectors: Euclidean and Cosine. However, it appears that cosine distance yields better results than Euclidean distance. We suppose that it is because, with convolutional neural networks, a neuron is activated with the presence of a feature. The level of presence of the feature is not as important as the fact that it is present.

Of course, CNN features are very robust against modifications but they are not suited for large scale applications. Indeed, the dimensionality of CNN feature vectors is very high. For more details, see the dimensions column of table 8.1. As explained in chapter 2, it is currently impossible to index p-dimensional feature vectors when p is greater than 20. For a large-scale application, one should reduce the dimension of the CNN feature vectors. In the next chapter, we present a method to map feature vectors into binary codes that preserve similarity.

9. CNN Features Hashing

9.1. Introduction

In a previous chapter 5, we advocate the use of small binary codes and hashing for nearest neighbor search. In this chapter, we present a new approach for hashing into binary codes inspired from LSH with random projection and Minimal Loss Hashing, which are detailed in 5.

Our approach is based on the same space partitioning as LSH with random projection and on the same cost function as Minimal Loss Hashing, it is therefore a supervised learning approach. The space is partitioned with a set of hyperplanes, each of them being responsible for the value of one bit in the binary code. If a point is on one side of a hyperplane, the associated bit takes a value of 1 and if the point is on the other side, the bit takes a value of 0. The difference between Minimal Loss Hashing and our approach is the optimization method. Minimal Loss Hashing builds a convex-concave upper bound on the loss function and a stochastic gradient-based method is used to minimize the bound function. On our side, we optimize the cost function in a continuous space. During the training, instead of binary codes, our algorithm generates q-dimensional vectors whose elements are in $[0, 1]$. Because codes are continuous, optimization is easier. Once training is done, q-dimensional vectors are binarized with a threshold to obtain binary codes.

9.2. Formulation

The task is to find a hash function that maps p-dimensional vectors, $x \in \mathbb{R}^p$, into q-bit binary codes, $h \in \mathbb{H} \equiv \{0, 1\}^q$, while preserving similarity:

$$b : \mathbb{R}^p \mapsto \{0, 1\}^q$$

The family of hash function that we consider is a thresholded linear transformation, parametrized by $W \in \mathbb{R}^{q \times p}$:

$$b_W(x) = thr(Wx)$$

where $thr(\cdot)$ is an element-wise Heaviside function. The space is partitioned with a set of hyperplanes, each of them being responsible for the value of one bit in the binary code. If a point is on one side of a hyperplane, the associated bit takes a value of 1 and if the point is on the other side, the bit takes a value of 0. For more details, see chapter 5.

The objective is to optimize the values of elements in W so that the notion of similarity is as far as possible preserved in Hamming space.

9.3. Our approach

The previously defined hash function is hard to optimize because its output is discontinuous due to the $thr(\cdot)$ function. To address this problem, we make binary codes continuous, $c \in [0,1]^q$, during training by ignoring the $thr(\cdot)$ function and applying an element-wise sigmoid function. The sigmoid function is used to hold the results of the projections between 0 and 1.

$$c_{k;W}(x) = sigmoid_k(Wx) = \frac{1}{1+e^{-kWx}} \text{ with } k \in \mathbb{R}_+$$

If the result of $c_{k;W}(x)$ is binarized with a threshold of 0.5, it is equivalent to $b_W(x)$. To extend the Hamming distance on binary codes to continuous codes, we use an element-wise L1 norm.

$$d_{continuous}(c,c') = \|c-c'\|_C = \sum_{i=1}^{q} |c_i - c'_i| \text{ with } c,c' \in [0,1]^q$$

We can see that if the two continuous codes are composed only of binary values, $d_{continuous}$ is equivalent to Hamming distance on binary codes. It is also important to note that if two elements have values of 0.5, the distance between them will be exactly zero. This can be annoying because the optimization can fall in a situation where element values are 0.5 (See figure 9.1). To prevent the optimization from doing that, we can add a regularization term to the distance that penalizes elements that are not near to 0 or 1.

Dataset

The learning is based on a training data set of labeled pairs (x_i, x'_i, s_i) where x_i and x'_i are p-dimensional centered training points and s_i is a similarity label, whose value is 1 if x_i and x'_i are similar and 0 if dissimilar. To preserve a specific metric, one can compute the similarity label by thresholding the pairwise distance. Alternatively, to

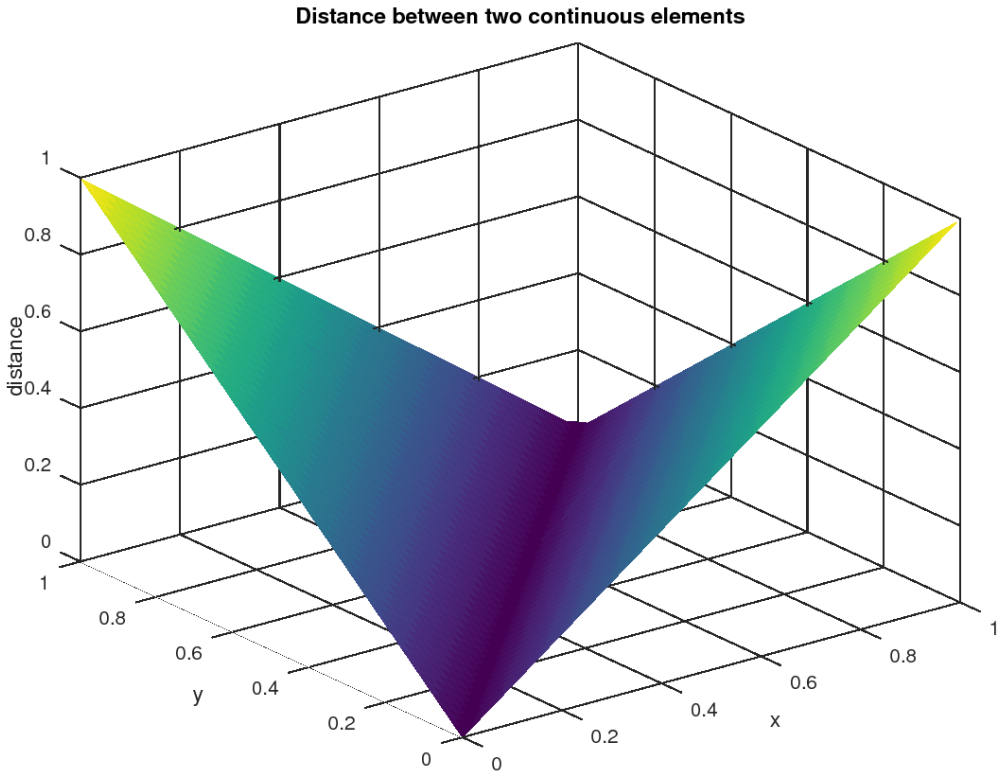

Figure 9.1.: Surface of the distance function between two continuous elements.

preserve the semantic similarity, one can assign a similarity label based on the class label. It is important to note that, if we want an exhaustive set of pairs, the number of pairs grows quadratically with the number of input vectors. With n images, there are $\frac{n(n-1)}{2}$ pairs (2-combination) of input vectors.

$$D \equiv \{x_i, x'_i, s_i\}_{i=1}^{N} \text{ with } x_i, x'_i \in \mathbb{R}^p; s_i \in \{0, 1\}$$

Preprocessing

In our implementation, it is possible to preprocess input vectors before training. Available preprocessing steps are: PCA, mean-centering and normalization to unit length, always executed in this order. Usually, only mean-centering and normalization are activated. PCA is simply implemented for experiments, to evaluate the impact of decorrelation and dimensionality reduction before hashing.

Loss function

To optimize the values of the parameters of the hash function, we define a cost function that is minimal when similarity is preserved. The cost function is inspired from Minimal Loss Hashing [NF11]. For each pair of input vectors in the training data set, a loss function $l_{pair} : [0, 1]^q \times [0, 1]^q \times \{0, 1\} \mapsto \mathbb{R}^+$ assesses the quality of the mapping by assigning a cost to a pair of continuous codes and a similarity label. The cost function penalizes similar (resp. dissimilar) samples that are mapped onto continuous codes that are dissimilar (resp. similar). By minimizing the loss over all training examples, we can learn the best parameters of the hash function.

We introduce the hyperparameter ρ, which is a threshold of distance in Hamming space, that defines the boundary between similar and dissimilar codes. If the distance between two codes is less than (resp. more than) ρ we can consider them as similar (resp. dissimilar). Although it is a slow process, a good value for this hyper parameter can be found with a validation set.

In addition, the λ parameter adjusts the penalty incurred for dissimilar pairs when they are too close in comparison to the penalty incurred for similar pairs when they are too far from each other. In the case that there are significantly less pairs of similar images than pairs of dissimilar images, the optimization method can find a solution where all codes are different because it satisfies the majority of constraints. To mitigate this effect, λ is equal to the number of similar pairs divided by the number of dissimilar pairs.

$$\lambda = \frac{\sum_{i=1}^{N} s_i}{N - \sum_{i=1}^{N} s_i}$$

Let $\|c - c'\|_C$ be the continuous Hamming distance between continuous codes c and c'. The pairwise loss based on the hinge function is defined as:

$$l_{pair}(c, c', s) = \begin{cases} \max(\|c - c'\|_C - \rho, 0) & \text{for } s = 1 \\ \lambda \max(\rho - \|c - c'\|_C + 1, 0) & \text{for } s = 0 \end{cases}$$

$$\begin{aligned} L_{continuous}(W) &= \sum_{i=1}^{N} l_{pair}(c_{k;W}(x_i), c_{k;W}(x'_i), s_i) \\ &= \sum_{i=1}^{N} s_i \max(0, \|c_i - c'_i\|_C - \rho) + (1 - s_i)\lambda \max(0, \rho - \|c_i - c'_i\|_C + 1) \end{aligned}$$

It is possible to find local optima where some elements of continuous codes are near to 0.5, which is not desirable when the codes are later thresholded. In this case the cost function is artificially better than it should be. To monitor the training without this bias, it is possible to compute the real cost function that takes binary codes instead of continuous codes. If the value of the continuous cost function is significantly less than the value of the real cost function, there might be elements that are not near to 0 or 1.

$$\begin{aligned} L_{real}(W) &= \sum_{i=1}^{N} l_{pair}(b_W(x_i), b_W(x'_i), s_i) \\ &= \sum_{i=1}^{N} s_i \max(0, \|h_i - h'_i\|_H - \rho) + (1 - s_i)\lambda \max(0, \rho - \|h_i - h'_i\|_H + 1) \end{aligned}$$

Regularization

To prevent the situation where elements of continuous codes take values near to 0.5, we add a regularization term that forces all elements of all codes to take values near to 0 or 1. The regularization function should output 0 if an element is exactly 0 or 1, and r if it is 0.5. Moreover, the function should be smooth to ease the calculation of the gradient.

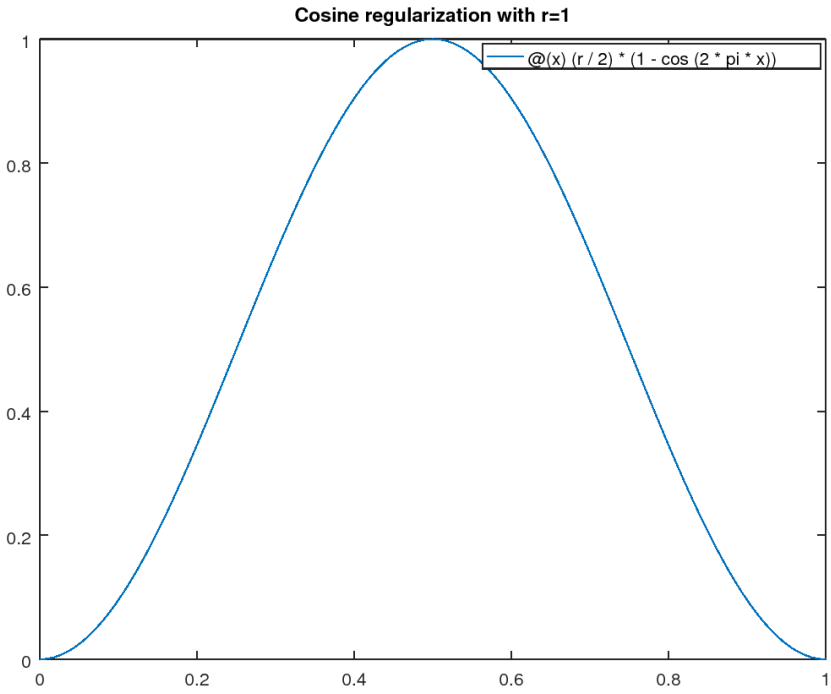

Figure 9.2.: Cosinus regularization with $r = 1$

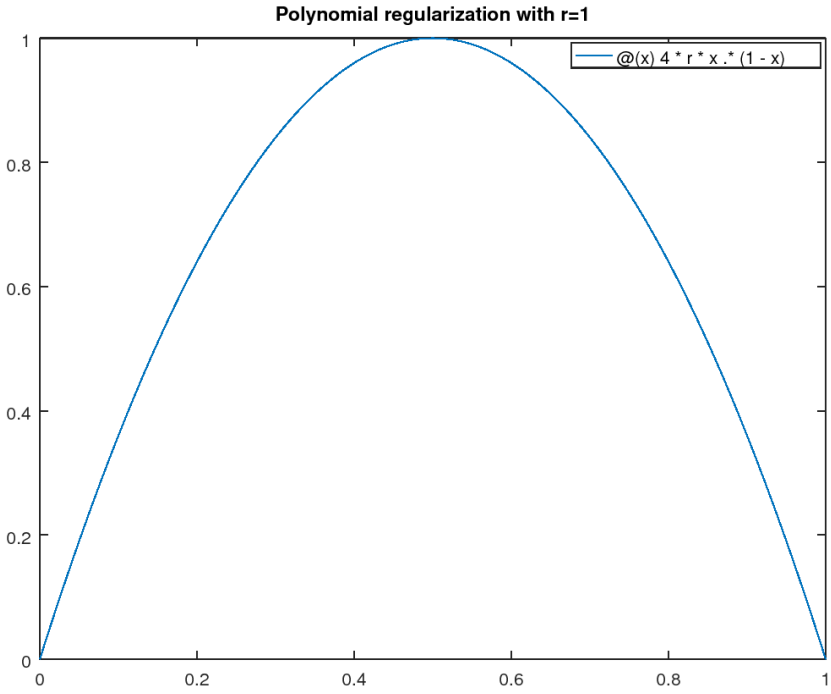

Figure 9.3.: Polynomial regularization with $r = 1$

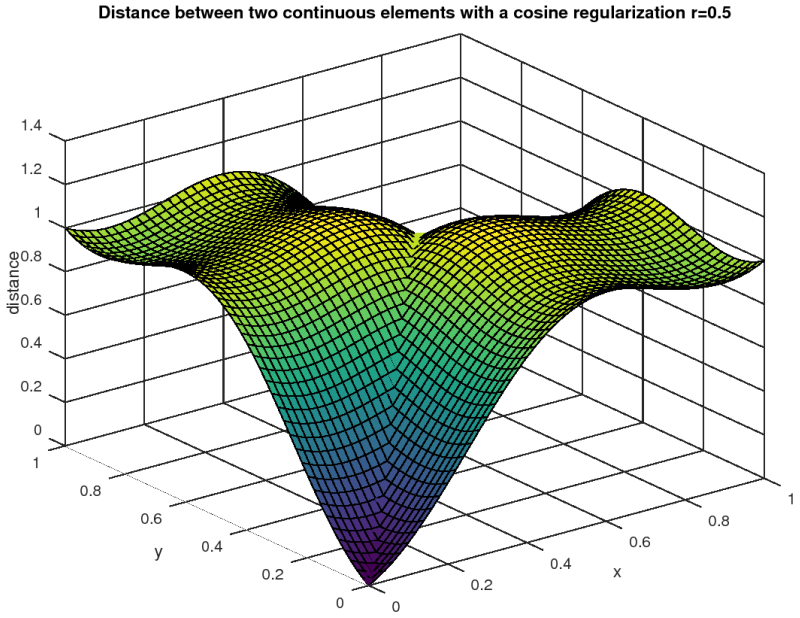

Figure 9.4.: Distance between two continuous bits with a cosine regularization $r = 0.5$

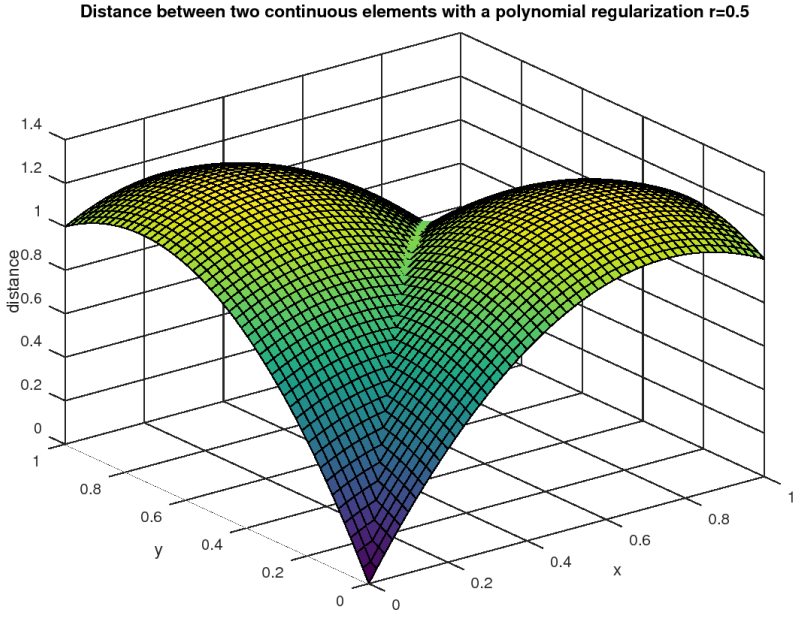

Figure 9.5.: Distance between two continuous bits with a polynomial regularization $r = 0.5$

We implemented two types of regularization: cosine (see figure 9.2) and polynomial (see figure 9.3).

$$R_{cos}(x) = \frac{r}{2}(1 - \cos 2\pi x) \text{ with } r \in \mathbb{R}_+$$

$$R_{poly}(x) = 4rx(1 - x) \text{ with } r \in \mathbb{R}_+$$

Cosine regularization has a null derivative near to 0 and 1 whereas polynomial regularization is steeper in these two regions.

Effect of regularization on the distance function between two continuous bits can be seen in figure 9.4 and 9.5. We can see that elements near to 0.5 are penalized, therefore the distance is more than 1. The line between $(0, 0)$ and $(1, 1)$ is distinguishable from the rest of the surface, this originate from the L1 norm in the not regularized distance.

9.3.1. Training

To train the model, we compute the gradient of the cost function and use a gradient descent based method to minimize the cost.

Gradient of the cost function

To compute the gradient of the cost function, we first compute the Jacobian of $c_{k;W}(x)$, the sigmoid function.

$$c_{k;W}(x) = sigmoid_k(Wx) = \frac{1}{1 + e^{-kWx}} \text{ with } k \in \mathbb{R}_+$$

As a remainder, the jth element of the ith input vector is: $c_{k;W}(x_i)_j$. To simplify we note: $c_i = c_{k;W}(x_i)$, $c_{ij} = c_{k;W}(x_i)_j$, $c'_i = c_{k;W}(x'_i)$, $c'_{ij} = c_{k;W}(x'_i)_j$.

$$\frac{\partial c_{k;W}(x)_i}{\partial W_{ij}} = k \times c_{k;W}(x)_i(1 - c_{k;W}(x)_i)x_j$$

We then compute the gradient of the distance with the absolute value function.

$$\|c - c'\|_C = \sum_{i=1}^{q} |c_i - c'_i| \text{ with } c, c' \in [0, 1]^q$$

$$\frac{\partial \left| c - c' \right|}{\partial W_{ij}} = sgn(c - c')(\frac{\partial c}{\partial W_{ij}} - \frac{\partial c'}{\partial W_{ij}})$$

$$\frac{\partial \|c - c'\|_C}{\partial W_{ij}} = \sum_{i=1}^{q} sgn(c_i - c'_i)(\frac{\partial c_i}{\partial W_{ij}} - \frac{\partial c'_i}{\partial W_{ij}})$$

We then compute the gradient of the hinge functions.

$$\frac{\partial max(0, f(W) - \rho)}{\partial W_{ij}} = \begin{cases} 0 & \text{if } f(W) \leq \rho \\ \frac{\partial f(W)}{\partial W_{ij}} & \text{if } f(W) > \rho \end{cases}$$

$$= \mathbb{1}_{f(W)>\rho} \frac{\partial f(W)}{\partial W_{ij}}$$

$$\frac{\partial max(0, \rho - f(W) + 1)}{\partial W_{ij}} = \begin{cases} 0 & \text{if } f(W) \geq \rho + 1 \\ -\frac{\partial f(W)}{\partial W_{ij}} & \text{if } f(W) < \rho + 1 \end{cases}$$

$$= \mathbb{1}_{f(W)<\rho+1} \frac{\partial f(W)}{\partial W_{ij}}$$

Finally the gradient of the cost function is:

$$L_{continuous}(W) = \sum_{t=1}^{N} s_t \max(0, \|c_t - c'_t\|_C - \rho) + (1 - s_t)\lambda \max(0, \rho - \|c_t - c'_t\|_C + 1)$$

$$\frac{\partial L_{continuous}(W)}{\partial W_{ij}} = \sum_{t=1}^{N} s_t \mathbb{1}_{\|c_t - c'_t\|_C > \rho} \times sgn(c_{tj} - c'_{tj}) k \left(c_{tj}(1 - c_{tj}) x_{ti} - c'_{tj}(1 - c'_{tj}) x'_{ti} \right)$$
$$- (1 - s_t)\lambda \mathbb{1}_{\|c_t - c'_t\|_C < \rho+1} \times sgn(c_{tj} - c'_{tj}) k \left(c_{tj}(1 - c_{tj}) x_{ti} - c'_{tj}(1 - c'_{tj}) x'_{ti} \right)$$

$$\frac{\partial L_{continuous}(W)}{\partial W_{ij}} = \sum_{t=1}^{N} \left(s_t \mathbb{1}_{\|c_t - c'_t\|_C > \rho} - (1 - s_t)\lambda \mathbb{1}_{\|c_t - c'_t\|_C < \rho+1} \right)$$
$$\times sgn(c_{tj} - c'_{tj}) k \left(c_{tj}(1 - c_{tj}) x_{ti} - c'_{tj}(1 - c'_{tj}) x'_{ti} \right)$$

The octave code to compute the cost function is in appendix C.

Gradient of the regularization term

Gradient of the cosine regularization

$$R_{cos}(W) = \sum_{i=1}^{N}\sum_{j=1}^{q} \frac{r}{2}(1-\cos(2\pi c_{ij})) + \frac{r}{2}(1-\cos(2\pi c'_{ij}))$$

$$\frac{\partial R_{cos}(W)}{\partial W_{ij}} = \sum_{t=1}^{N} \pi k r \sin(2\pi c_{tj}) c_{tj}(1-c_{tj}) x_{ti} + \pi k r \sin(2\pi c'_{tj}) c'_{tj}(1-c'_{tj}) x'_{ti}$$

Gradient of the polynomial regularization

$$R_{poly}(W) = \sum_{i=1}^{N}\sum_{j=1}^{q} 4rc_{ij}(1-c_{ij}) + 4rc'_{ij}(1-c'_{ij})$$

$$\frac{\partial R_{poly}(W)}{\partial W_{ij}} = \sum_{t=1}^{N} 4rk(1-2c_{tj})c_{tj}(1-c_{tj})x_{ti} + 4rk(1-2c'_{tj})c'_{tj}(1-c'_{tj})x'_{ti}$$

Optimization

The parameters W are initialized using LSH. The elements of W are sampled from a normal density $\mathcal{N}(0,1)$ and each row is normalized. Our implementation [1] use the fminunc function of Octave. It is possible to run the optimization multiple times with different initializations in order to take the best solution. For reproducibility, it is possible to fix the seed before the optimization. Thus, one can run the optimization multiple times with the same initialization.

To monitor the training, the continuous and real costs are displayed before and after gradient descent. If the continuous cost doesn't decrease too much, it is probably because the cost function is too hard to optimize. In this case a decreasing in the value of k could help because it makes the surface of the cost function smoother. After training, a histogram of values of elements in continuous codes is computed. Ideally this histogram would contain an equal amount of values near to 0 or 1. If elements take values between 0 and 1, one could increase the value of k in order to sharpen the sigmoid function, another solution would be to increase the value

[1] https://github.com/mgaillard/PerceptualHashingLearning

of r in order to regularize the cost function. However, a too high value for the regularization increases the number of local minima and the cost function becomes harder to minimize. Finally, precision, recall and F-measure are computed on all pairs of images with similarity labels. If the distance between the binary codes of a pair of similar images is less (resp. greater) than ρ, it is a true (resp. false) positive. If the distance between the binary codes of a pair of dissimilar images is greater (resp. less) than ρ, it is a true (resp. false) negative. There is an example of report generated to monitor the training in appendix D.

9.4. Experiments

Experiments are conducted on the Octave implementation.

9.4.1. 2D points dataset

As a first experiment, we use a dataset of random 2D points. Indeed, with only 2 dimensions, the points are very easy to visualize. Moreover, in this case hyperplanes are lines parametrized by only two values. It means that if we hash 2D points with only one bit it is possible to plot the surface of the corresponding cost function. This experiment is thus only useful to develop and visually control that our method performs well.

The input dataset is composed of n clouds of k points in 2D. Each cloud of point is centered around a point, whose coordinates are randomly chosen from a uniform distribution between -10 and 10. In other words, clouds are in a square of side 20 centered around the origin. To generate points in clouds, we choose each coordinate from a normal distribution of variance s and whose mean is the center of the cloud. Then we generate an exhaustive list of pairs of points with similarity labels. Similar (resp. dissimilar) points come from the same (resp. different) cloud.

With two clouds

The result of our method can be seen in figure 9.6 and the surface of the cost function is in figure 9.7. In this example, we can clearly see that the solution is perfect. No regularization were used, $k = 1$ and $\rho = 0$, which means that no hyperplane should pass through a cloud.

This problem is similar to the linearly separable case of SVM. There is not a maximum margin constraint but the hyperplanes are not too close to the clouds. This is probably due to the fact that the minimization in continuous space has a similar

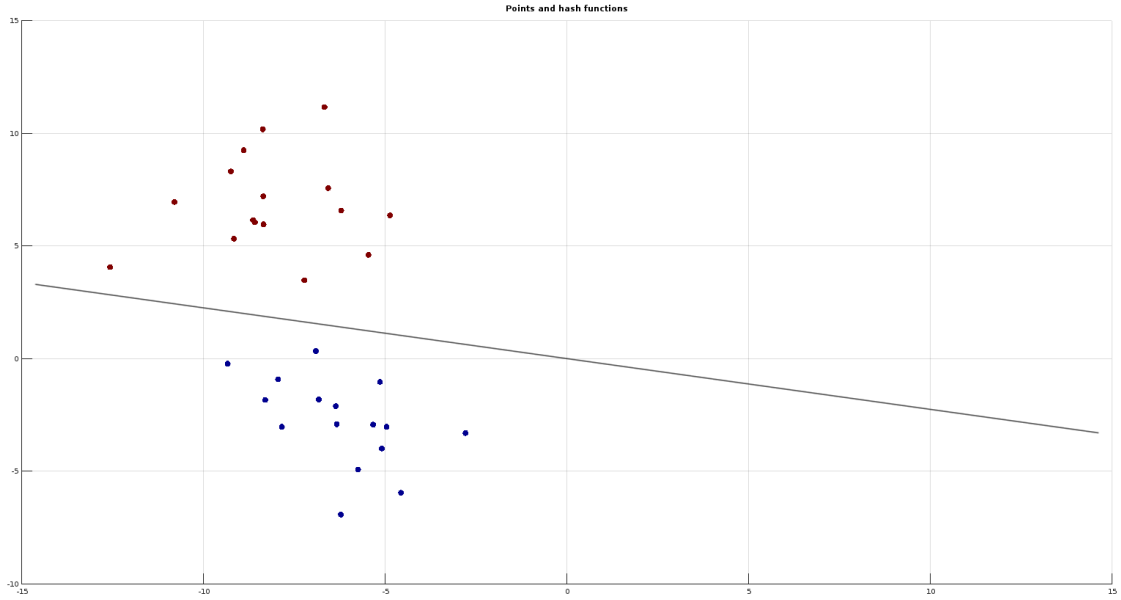

Figure 9.6.: Scatter plot of the dataset with 2 clouds of 16 points. The best hyperplane found is the black line passing through the origin.

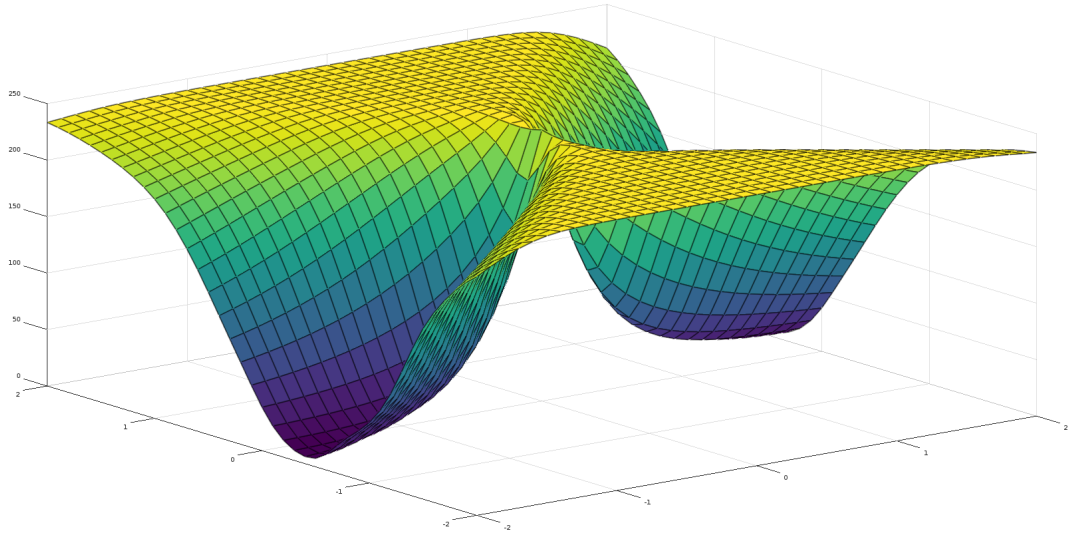

Figure 9.7.: Surface of the cost function for one hyperplane in 2 dimensions and a distribution of points shown in figure 9.6. In this case two regions seem to be optimum.

effect. Indeed, the continuous representation is never exactly equal to 0 or 1. Therefore, the optimization will force the continuous representation to take a value nearer to 0 or 1 even if the value is already near to 0 or 1. In other words, the hyperplane will stay away from the points, maximizing the margin between the hyperplane and points.

With four clouds

The result of two optimizations on 4 clouds of 16 points can be seen in figures 9.8 and 9.9. No regularization were used, $k = 1$ and $\rho = 0$. The first figure 9.8 shows a case where the solution is perfect. In this particular case, only two hyperplanes would have been enough. The second figure 9.9 shows a case where a local optimum is found. One hyperplane passes through the yellow cloud. This is possible because, from the point of view of the cost function, a value of 0.5 for two similar points is considered as satisfying because no regularization is applied.

9.4.2. CNN Features dataset

We extract features with *VGG16_block5_pool_max* because it has at the same time good retrieval performances and a low number of dimensions. It is desirable because the computation of the cost function is easier when features have less dimensions.

To evaluate the quality of the hashing, we use an adaptation of our benchmark, in chapter 7. We use the n first images of MIRFLICKR [2] and their $K = 6$ modified versions according to the modifications listed in section 7.2: blur, grayscale, resize, JPEG compression, rotation and cropping. So we have $N = (K + 1)n$ images in total. The input dataset is composed of all $N(N - 1)/2$ pairs of images along with their similarity labels. A pair of images that both come from the same base image is considered as similar. In the same way, a pair of images that don't come from the same base image is considered as dissimilar. So there are $n * (K + 1)K/2$ similar pairs and $N(N - k - 1)/2$ dissimilar pairs. Because there is too many dissimilar pairs in comparison to similar pairs, we adapt lambda so that $\lambda = K/(N - K - 1)$. CNN Features are mean-centered (all at the same time) and then normalized to unit length in L2. Once training is finished, we predict the binary codes of all N images and run the benchmark on the codes. It is important to note that there is no validation and test set. We learn our hash function from the actual data, thus overfitting is not a problem. Finally, we compare the precision, recall and F-measure of generated binary codes in Hamming distance to raw features with cosine distance.

[2] http://press.liacs.nl/mirflickr/

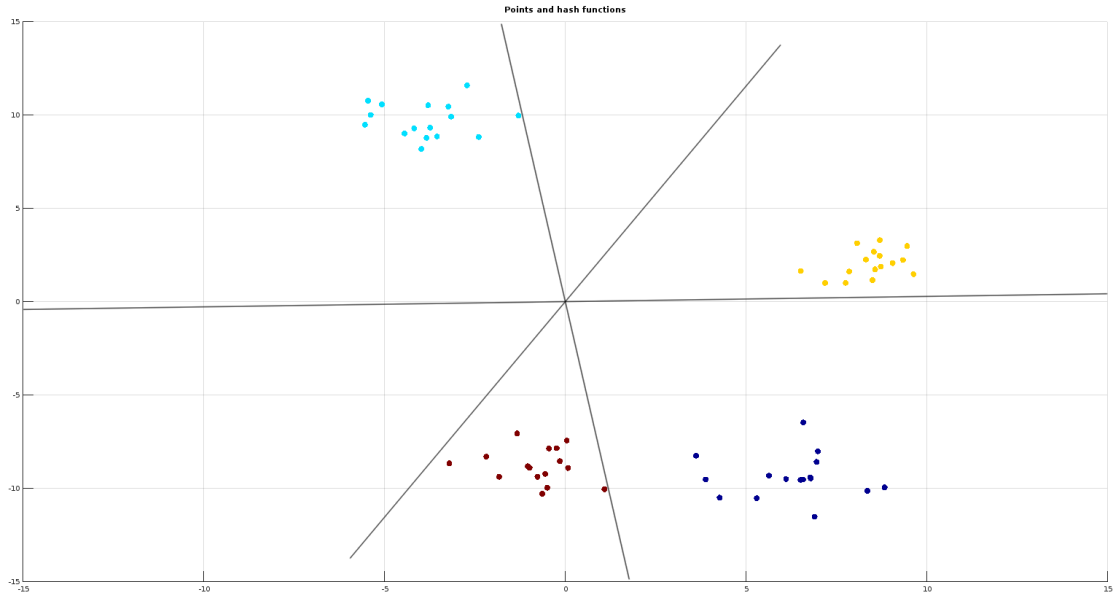

Figure 9.8.: Scatter plot of the dataset with 4 clouds of 16 points. The best hyperplanes found are the black lines passing through the origin.

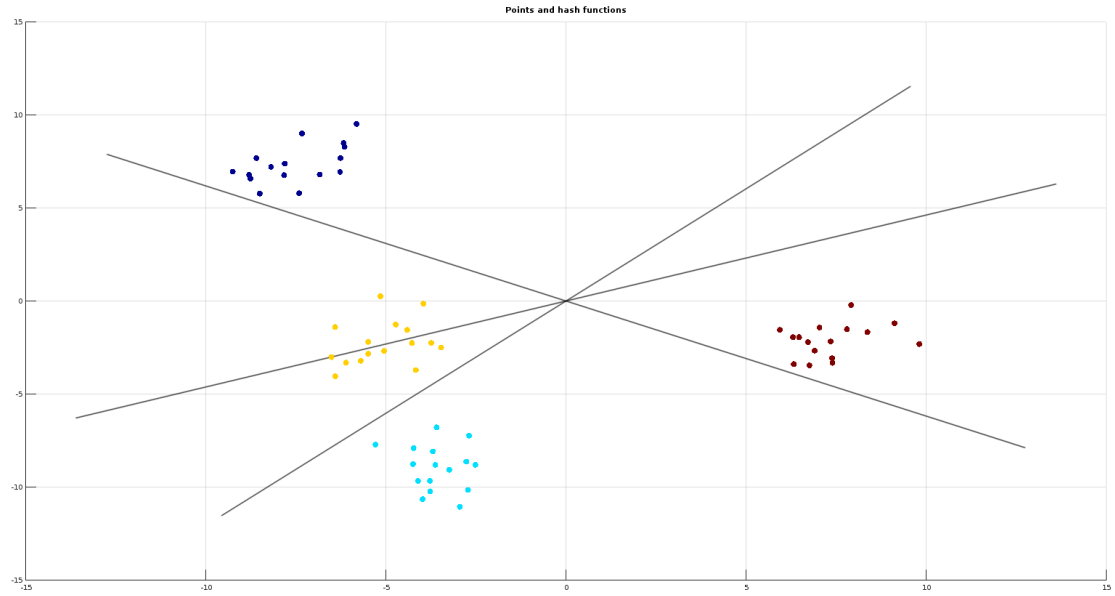

Figure 9.9.: Scatter plot of the dataset with 4 clouds of 16 points. The best hyperplanes found are the black lines passing through the origin. Here one hyperplane passes through a cloud.

Table 9.1.: Comparison of retrieval performances of our method against LSH with random projection

Method	Real cost	Radius	Precision	Recall	Fmeasure
LSH	797	0,18	0,75	0,51	0,56
Our method	21.7	0,18	0,99	0,92	0,95

Ideally, binary codes would have the same retrieval performance as raw features. It is even possible that binary codes perform better than raw features. Indeed, similarity labels stem from classes of images and not from the cosine distance between their representations. Therefore, even if angular distance is not perfect to compare image representations, it is possible to find a partitioning of the space that leads to perfect binary codes. In this particular case, the hash function learns to correct the mistakes of the image representations. This approach would be also applicable to learn the semantic similarity in a classification task. In this case, a nearest neighbor classifier would be used on the learnt binary codes.

Comparison with LSH

In this section, we compare the performances of our method to random projection. In theory, our method should perform at least as well as LSH with random projection because it is initialized exactly like LSH. We take the best model over 20 iterations for our method, and the best model over 1000 iterations for LSH. In both cases the seed is initialized with the same value. We hash the $n = 50$ first images of MIRFLICKR and their modified versions (so $N = 350$) on 16-bits binary codes with $\rho = 3$, $k = 1$ and no regularization. We plot the histogram of real cost for both methods.

Results are in table 9.1. The histogram of cost function for LSH is shown in figure 9.10, the mean cost is 1184 and the standard deviation is 106. With 1000 iterations, we can say that LSH is not likely to produce solutions that are really better than the one that we found with the minimal cost of 797. So the F-measure of 0,56 obtained with LSH is a upper bound of the performance of LSH. Our method achieves every time a better solution than LSH with a maximum real cost of 81. The mean real cost is 37.8, the standard deviation is 14.8 and the minimum real cost is 21.7. We can see that our method achieves every time a higher F-measure than LSH with random projection. By the way, it is interesting to note that the best F-measure is reached with a radius of 0.18, which is approximately equal to $\rho/16$.

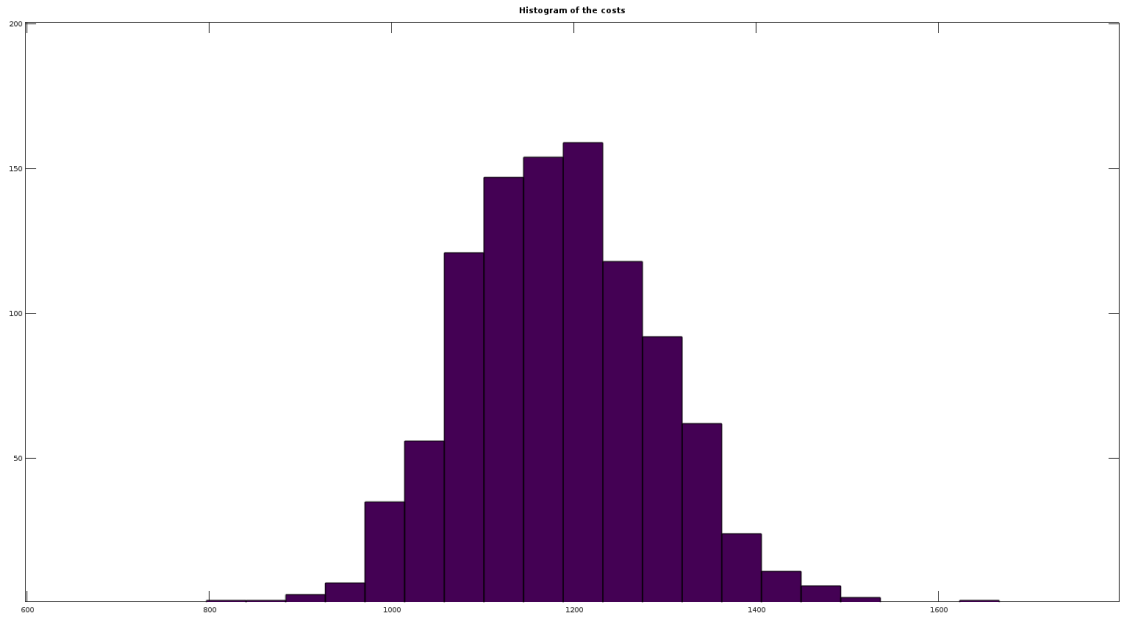

Figure 9.10.: Histogram of real cost for 1000 iterations of LSH.

9.5. Conclusion

In this chapter, we present a new approach to learn a hash function for CNN Features. Although our experiments are superficial and can't prove that our method yields state of the art performances, we have very promising results. With more time, we could have conducted more experiments. Some questions are still open: what is the influence of the regularization? How well our method performs when the number of bits increases, when the number of images increases?

The effect of regularization on the distance between two continuous elements could be better. Cosine regularization seems to introduce local minima in the cost function, thus the solutions are often worse with regularization. We didn't extensively test the polynomial regularization, but this one seems to introduce less local minima than cosine regularization. For the moment by choosing the right value for k, the regularization is not necessary, but this is a complicated process.

The k parameter is redundant because rows of W are unconstrained, they can take a big value, which is equivalent to a big k. We need to evaluate the effect of k and whether it is really required. There seems to be two options, either we keep k and we normalize the rows of W, or we remove k and we don't normalize W.

Because our implementation uses the fminunc function of Octave, we can't have an effect on the minimization of the cost function. What we could do is to plot the real cost along with the continuous cost during learning, in order to monitor the

optimization. We could also change the pairs of images in the dataset during the training to select at each step only the pairs that bring the more information.

During the training, we select the best model over multiple iterations by choosing the one with the minimum real cost. However, it is not proven that selecting the lowest real cost is equivalent to selecting the highest F-measure.

The optimization is not easy, the algorithm often falls into a local optimum and to find a satisfying optimum we need to rerun several times the gradient descent with various random inputs. Is it possible to use another optimization method that is less prone to falling in local minima? If we add hidden layers before the hashing layer, shall it be easier to optimize? In facts, the hidden layers can change the input space to ease the training of the last layer. At the end, the objective would be to program a loss layer with a neural network framework. This would allow us to use it on top of a neural network and learn non-linear projections. Moreover, we could use the GPU implementation of the framework to fasten the computation of the cost function.

Scalability is not perfect because the number of pairs of images grows quadratically. For this reason, the number of images is deliberately low because the cost function is hard to compute with a big dataset. For example, with a total of 1400 images, there is about 1 million pairs of images, which is huge. With a GPU implementation, we could go further but definitively not up to a big number of images. Another solution that we explored is to select only the pairs of images that bring the more information, because dissimilar triplets are mostly redundant. Unfortunately, we didn't have the time to evaluate extensively this option. Currently the way to apply this method to large scale reverse image search is to train the model on a small number of images with a validation set and hope that it will generalize on more images.

10. Conclusion

In this thesis, we present our study on Convolutional Neural Networks Features and Perceptual Hashing for Large Scale Reverse Image Search. A potential application is to find near-duplicate in large collections of images.

To implement an actual reverse image search system with the content of this thesis, one can make use of the CNN Features described in chapter 8 to represent images, followed by the use of the method developed in chapter 9 to learn a hash function that maps CNN Features into short binary codes, finally techniques described in chapter 6 can be used to perform efficient nearest neighbor searches on binary codes.

In the review part, we describe what is reverse image search and common methods to tackle this problem. We also advocate the use of convolutional neural networks to represent images, and hashing into binary codes to facilitate nearest neighbor search.

The potential of CNN Features for reverse image search, even with slight modifications, is proven. Image representations extracted with convolutional neural networks on unrelated classification tasks are considerably robust against modifications. We didn't expressly compared CNN Features against other state of the art descriptors, but our experiments, along with the content of other publications presented in chapter 4, tend to prove that CNN Features are state of the art descriptors for reverse image search and more generally for image retrieval.

We propose a method to learn a hash function that maps features into short binary codes while preserving similarity. This method is inspired from Minimal Loss Hashing, described in chapter 5, but uses a new approach to optimize the cost function. We keep the codes continuous during learning, this allows us to compute the gradient of the cost function it minimize it. Although our experiments are superficial and can't prove that our method yields state of the art performances, we have very promising results. With more time, we could improve our method and evaluate its performances in comparison with state of the art techniques. Interesting directions are exhibited in the conclusion of chapter 9.

We foresee the implementation of our method as a loss layer in a neural network framework to promise excellent results. Firstly, this would allow us to benefit from the computational power of GPU, secondly, we would be able to add hidden layers to learn non-linear projections, and finally, the joint learning of the image representation and the hash function would yield even better results.

A. Implementation of DCT-based perceptual hash

During our project, we implemented a version of the DCT based perceptual hash function with OpenCV. This one has the same retrieval performance than the original one in [Zau10] and is faster.

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <string>
#include <cstdint>

uint64_t DctPerceptualHash(const std::string& file_path)
{
    uint64_t hash = 0;
    cv::Mat input;
    cv::Mat grayImg;
    cv::Mat bluredImg;
    cv::Mat resizeImg;
    cv::Mat resizeFImg;
    cv::Mat dctImg;

    input = cv::imread(file_path);

    if (input.data == NULL || (input.type() != CV_8UC3 &&
    ↪ input.type() != CV_8U)) {
        // If there is a reading error, return a null hash.
        cerr << "Error reading image" << endl;
        return 0;
    }

    // Convert the image to grayscale using its luminance
    if(input.type() == CV_8UC3) {
        cv::cvtColor(input, grayImg, CV_BGR2GRAY);
    } else {
        grayImg = input;
```

```
    }

    // Mean filter with kernel 7x7.
    cv::blur(grayImg, bluredImg, cv::Size(7, 7));

    // Resize the image to 32x32 pixels.
    cv::resize(bluredImg, resizeImg, cv::Size(32,32));

    resizeImg.convertTo(resizeFImg, CV_32F);
    cv::dct(resizeFImg, dctImg);

    std::vector<float> topLeftCoeffs;
    // Take only the coeffs in rectangle (1, 1) ; (9, 9)
    topLeftCoeffs.reserve(64);
    for (int i = 1; i <= 8; i++) {
        for (int j = 1; j <= 8; j++) {
            topLeftCoeffs.push_back(dctImg.at<float>(i, j));
        }
    }

    // Compute the median
    std::vector<float> coeffs(topLeftCoeffs);

    // The median of a 64 elements array is the mean between
    ↪  the 31th and the 32th element.
    float median = 0;
    std::nth_element(coeffs.begin(), coeffs.begin() + coeffs
    ↪ .size()/2, coeffs.end());
    median += coeffs[coeffs.size()/2];
    std::nth_element(coeffs.begin(), coeffs.begin() + coeffs
    ↪ .size()/2 - 1, coeffs.end());
    median += coeffs[coeffs.size()/2 - 1];
    median /= 2;

    // Transform into a 64 bits integer.
    for (int i = 0; i < topLeftCoeffs.size(); i++) {
        hash <<= 1;
        if (topLeftCoeffs[i] >= median) {
            hash |= 1;
        }
    }

    return hash;
}
```

B. Detailed results of VGG16_block5_pool_max with Cosine distance

In this appendix, we present the detailed results of the benchmark for *VGG16_block5_pool_max* with Cosine distance.

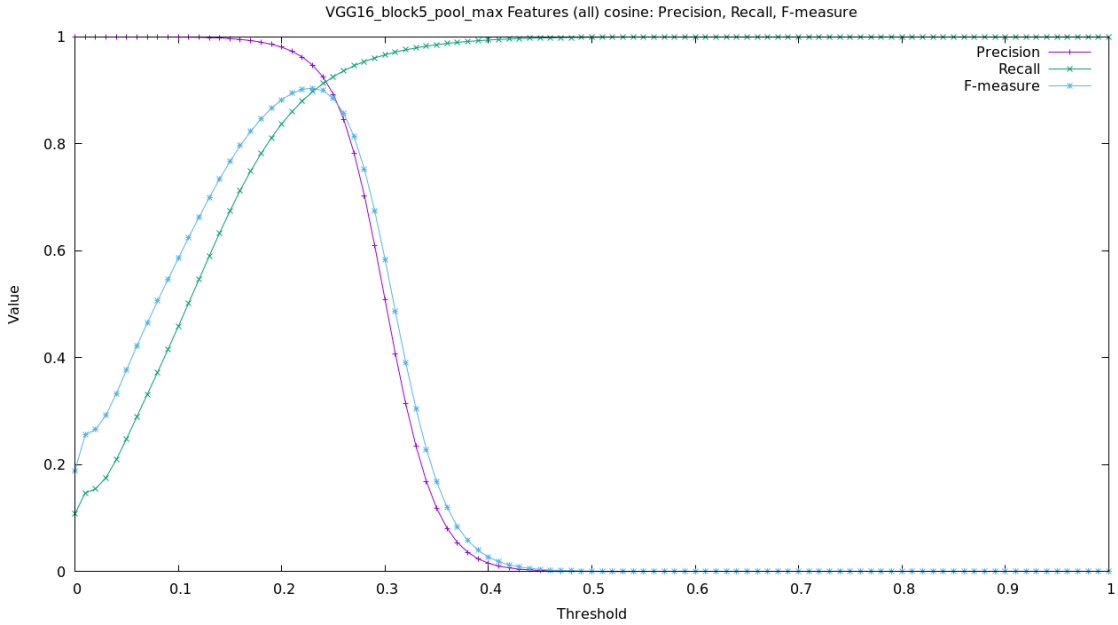

Figure B.1.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results with all modifications according to different radius (threshold).

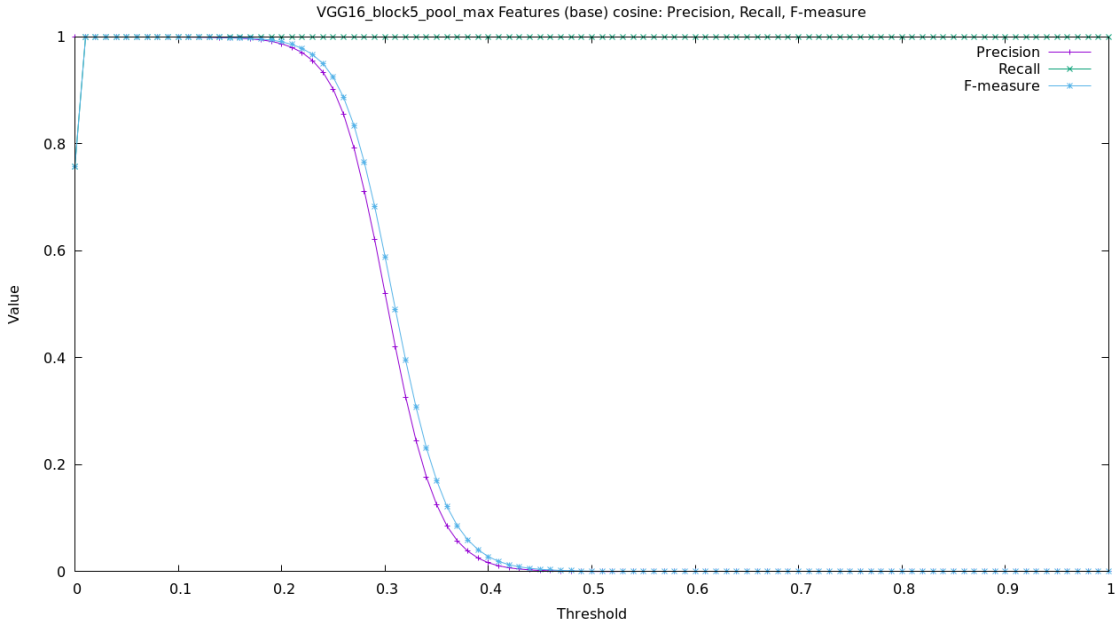

Figure B.2.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results with no modification according to different radius (threshold).

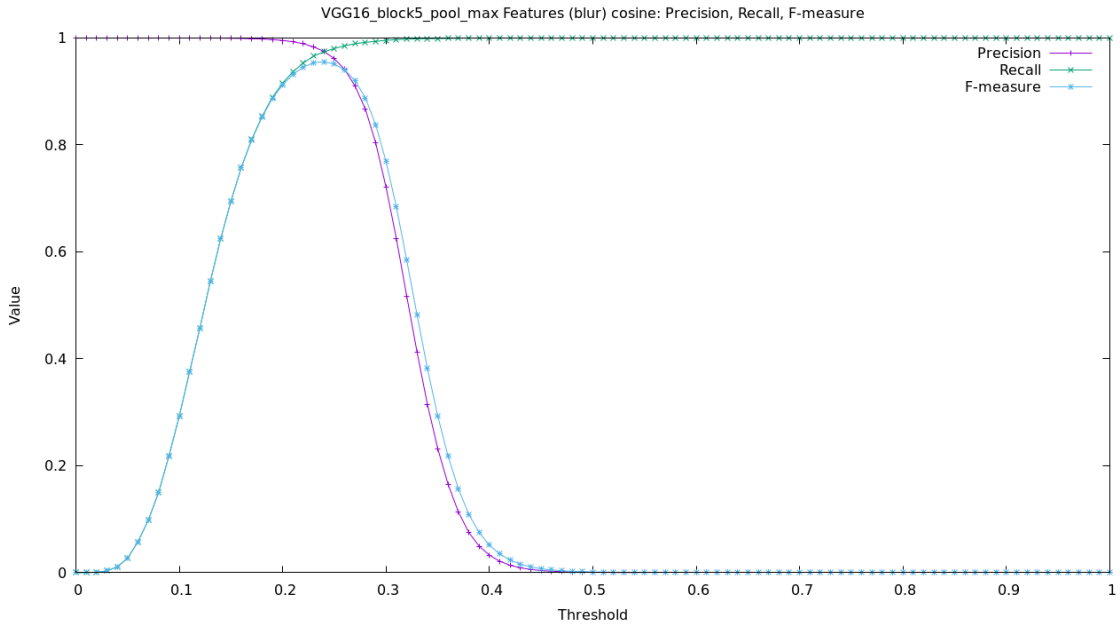

Figure B.3.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with Gaussian blur ($r = 4$, $\Sigma = 2$) according to different radius (threshold).

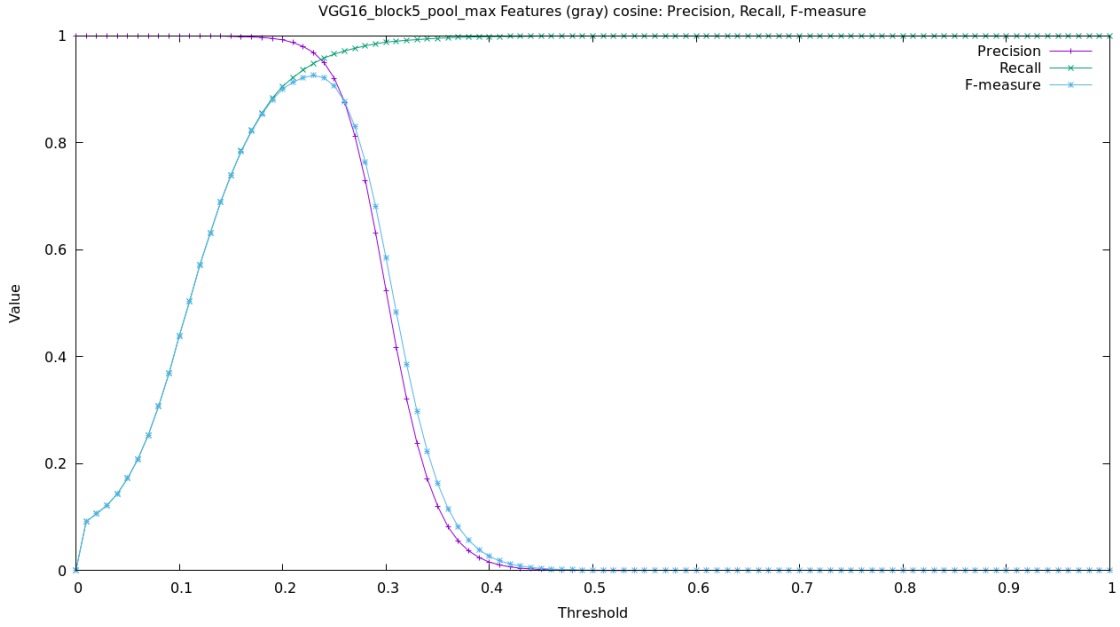

Figure B.4.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with grayscale filter according to different radius (threshold).

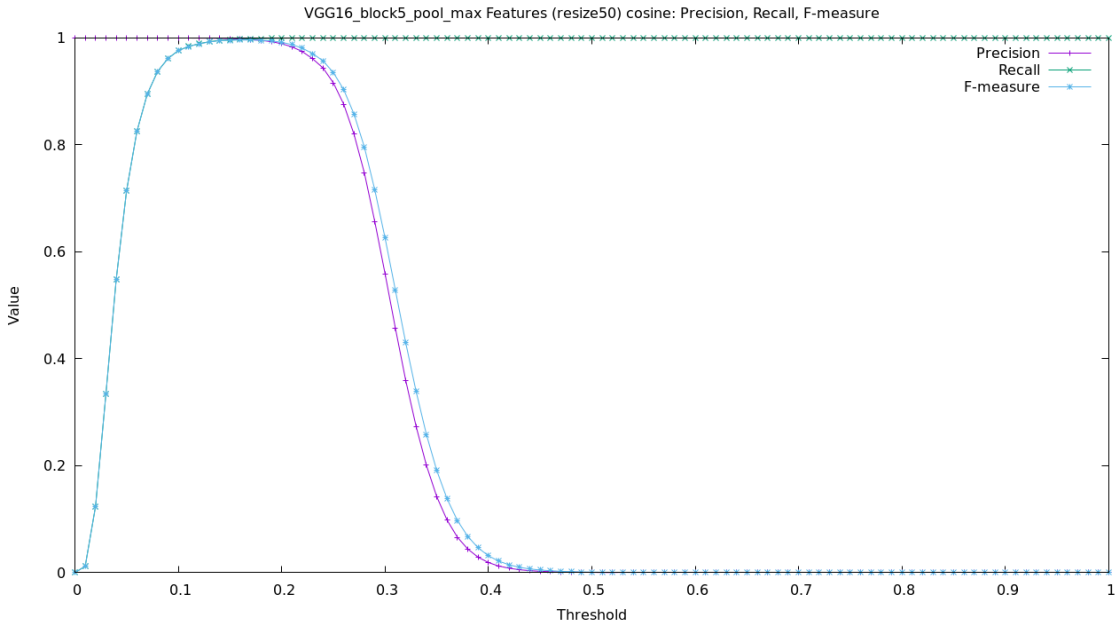

Figure B.5.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with resize to half size according to different radius (threshold).

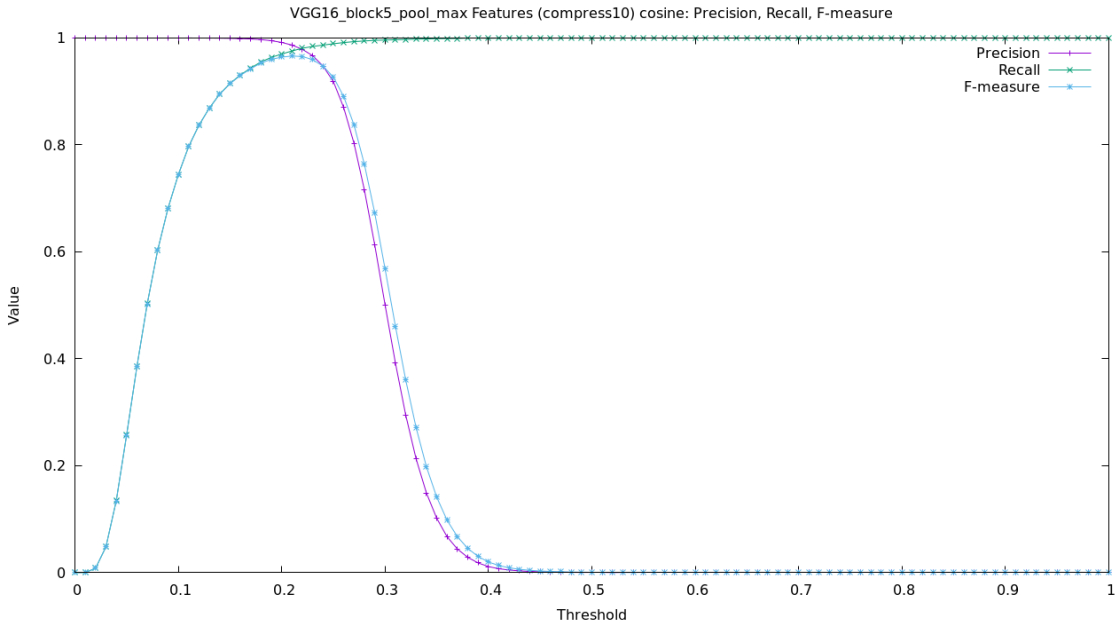

Figure B.6.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with JPEG compression (quality 10%) according to different radius (threshold).

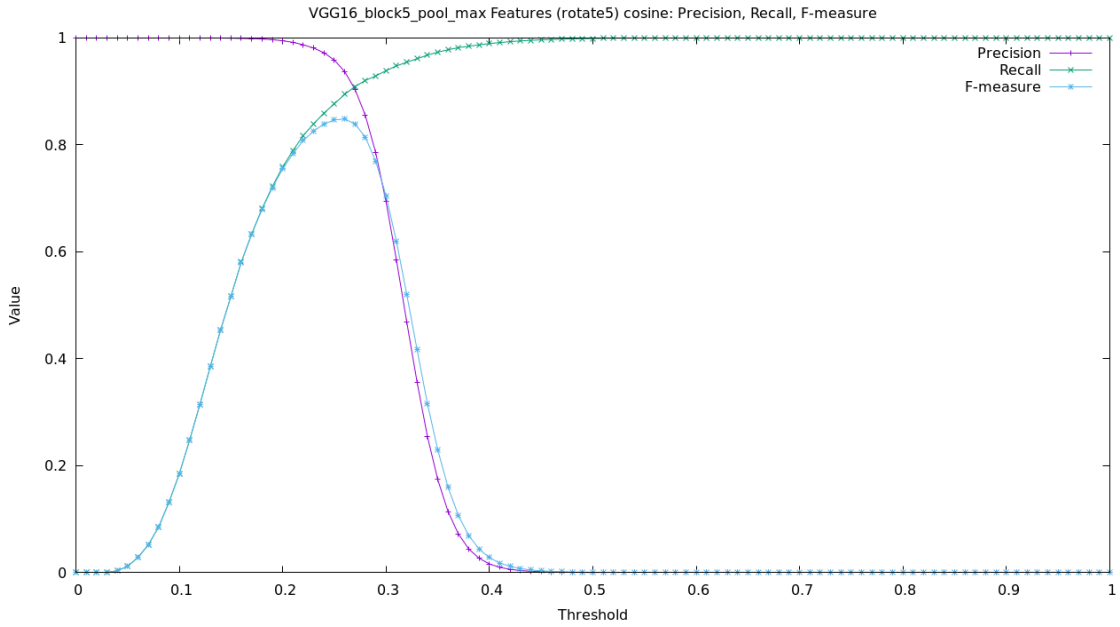

Figure B.7.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with clockwise rotation by 5 degrees according to different radius (threshold).

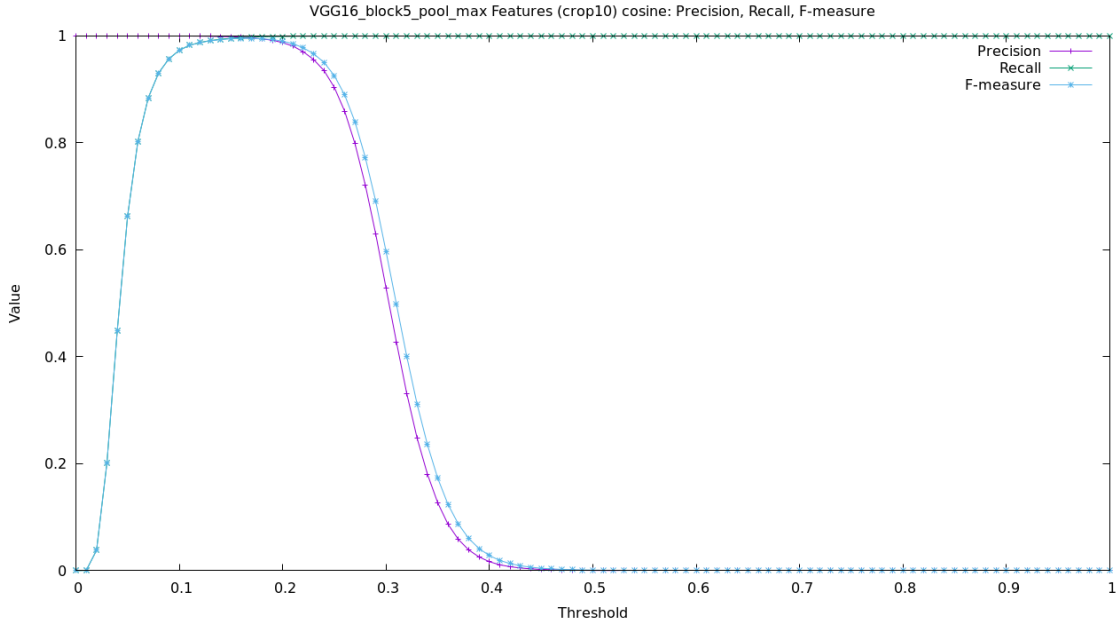

Figure B.8.: Precision, recall, Fmeasure, curves of *VGG16_block5_pool_max* search results only with cropping by 10% at the right side of the image according to different radius (threshold).

C. Octave Implementation

```
% Compute the cost and the gradient of a set of parameters W
% X1, X2 are input vectors and S is a similarity vector
% S(i) == 1 means that X1(i) and X2(i) are similar
% k, rho, lambda, regularization are hyper parameters of the
    ↪ cost function
function [cost grad] = HyperModel_ContinuousCostFunction(X1,
    ↪ X2, W, S, k, rho, lambda, regularization)
% Binary codes of the samples
a1 = 1.0 ./ (1.0 + exp(-k*X1*W));
a2 = 1.0 ./ (1.0 + exp(-k*X2*W));

% Hamming distance between the two binary codes.
Diff = a1 - a2;
HammingDist = sum(abs(Diff), 2);

% ------------ Cost ------------

cost = sum(S.*max(0, HammingDist - rho) + (1-S).*lambda.*max
    ↪ (0, (rho+1) - HammingDist));

% Regularization of activations
if (regularization > 0)
        % Cosine regularization
        % cost += sum(sum((regularization/2)*(1 - cos(2*pi*
            ↪ a1))));
        % cost += sum(sum((regularization/2)*(1 - cos(2*pi*
            ↪ a2))));

        % Polynomial regularization
        cost += sum(sum(4*regularization*a1.*(1 - a1)));
        cost += sum(sum(4*regularization*a2.*(1 - a2)));
endif

% ------------ Gradient ------------

DiffSign = sign(Diff);
```

```
HingeSimilar = [HammingDist > rho];
HingeDissimilar = [HammingDist < (rho+1)];

grad = X1'*(S .* HingeSimilar .* DiffSign .* k .* a1 .* (1 -
    ↪ a1)) ...
                    - X2'*(S .* HingeSimilar .* DiffSign .* k
                        ↪ .* a2 .* (1 - a2)) ...
                    - X1'*((1-S) .* lambda .* HingeDissimilar
                        ↪ .* DiffSign .* k .* a1 .* (1 - a1))
                        ↪ ...
                    + X2'*((1-S) .* lambda .* HingeDissimilar
                        ↪ .* DiffSign .* k .* a2 .* (1 - a2));

% Gradient of the regularization term
if (regularization > 0)
        % Cosine regularization
        % grad += X1'*(regularization*pi*sin(2*pi*a1) .* k
            ↪ .* a1 .* (1 - a1)) ...
        %         + X2'*(regularization*pi*sin(2*pi*a2) .* k
            ↪ .* a2 .* (1 - a2));

        % Polynomial regularization
        grad += X1'*(4*regularization*(1 - 2*a1) .* k .* a1
            ↪ .* (1 - a1)) ...
                                + X2'*(4*regularization*(1 -
                                    ↪ 2*a2) .* k .* a2 .*
                                    ↪ (1 - a2));
endif
end
```

D. Training Report

```
initial_cont_cost =  3000.3
initial_real_cost =  1006.0
optimal_real_cost =  27.985
optimal_cont_cost =  0.020816

------ Real cost statistics: ------
Min: 27.9854
Max: 64.9679
Median: 55.3994
Mean: 49.4509
Standard deviation: 19.1954

------ Analysis ------
Middle activation: 1721
Number of training samples: 61075
Number of similar triplets: 1050
Number of dissimilar triplets: 60025
Ratio of similar triplets: 0.017192
Number of true positives: 1031
Number of false positives: 278
Number of false negatives: 19
Number of false negatives: 59747
Ratio of preserved similar triplets: 0.981905
Ratio of preserved dissimilar triplets: 0.995369
Precision: 0.787624
Recall: 0.981905
F1-measure: 0.874099
```

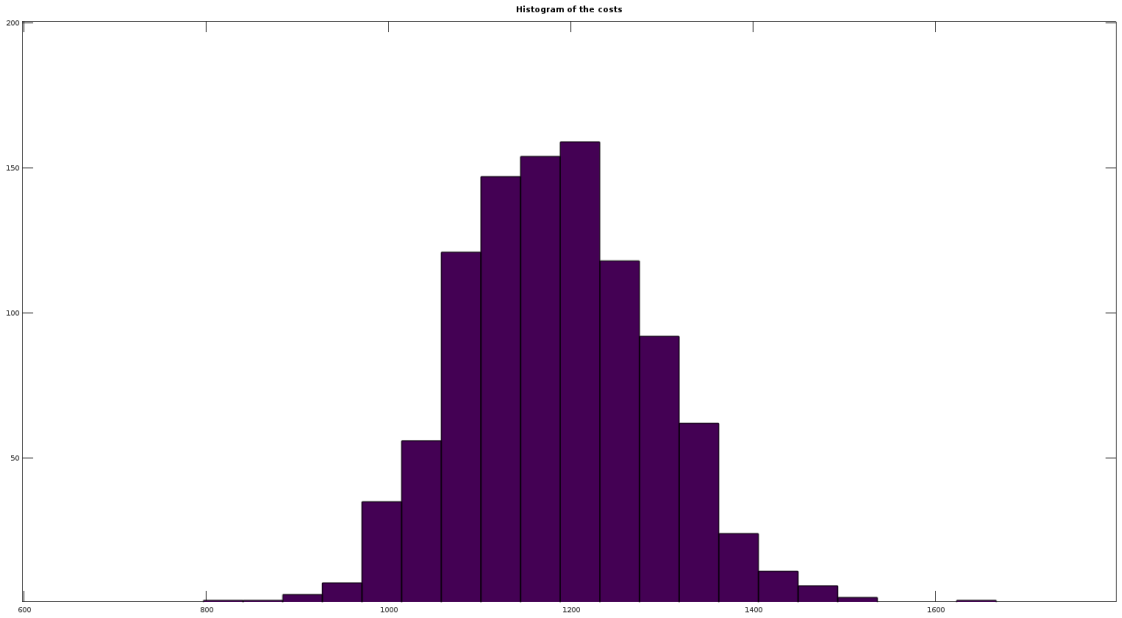

Figure D.1.: Histogram of the real cost of 1000 iterations of LSH

List of Figures

List of Tables

Bibliography

[BSCL14] Artem Babenko, Anton Slesarev, Alexandr Chigorin, and Victor Lempitsky. Neural codes for image retrieval. In *European conference on computer vision*, pages 584–599. Springer, 2014.

[C+15] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[CB08] Savvas A. Chatzichristofis and Yiannis S. Boutalis. Cedd: Color and edge directivity descriptor: A compact descriptor for image indexing and retrieval. In *Proceedings of the 6th International Conference on Computer Vision Systems*, ICVS'08, pages 312–322, Berlin, Heidelberg, 2008. Springer-Verlag.

[Cha02] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[DJH+09] Matthijs Douze, Hervé Jégou, Sandhawalia Harsimrat, Laurent Amsaleg, and Cordelia Schmid. Evaluation of GIST descriptors for web-scale image search. In *CIVR 2009 - International Conference on Image and Video Retrieval*, pages 19:1–8, Santorini, Greece, July 2009. ACM.

[DJLW08] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 40(2):5:1–5:60, May 2008.

[FCNL13] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2013.

[GD04] Christophe Garcia and Manolis Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 26(11):1408–1423, 2004.

[GE17] Mathieu Gaillard and Elöd Egyed-Zsigmond. Large scale reverse image search - A method comparison for almost identical image retrieval. In *Actes du XXXVème Congrès INFORSID, Toulouse, France, May 30 - June 2, 2017*, pages 127–142, 2017.

[GV16] Simon Gog and Rossano Venturini. Fast and compact hamming distance index. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 285–294. ACM, 2016.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[KH11] Alex Krizhevsky and Geoffrey E Hinton. Using very deep autoencoders for content-based image retrieval. In *ESANN*, 2011.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[Low04] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NF11] Mohammad Norouzi and David J Fleet. Minimal loss hashing for compact binary codes. In *Proceedings of the 28th international conference on Machine Learning (ICML-11)*, pages 353–360, 2011.

[NFS12] Mohammad Norouzi, David J Fleet, and Ruslan R Salakhutdinov. Hamming distance metric learning. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1061–1069. Curran Associates, Inc., 2012.

[Nor16] Mohammad Norouzi. *Compact Discrete Representations for Scalable Similarity Search*. PhD thesis, 2016.

[NPF12] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast search in hamming space with multi-index hashing. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3108–3115. IEEE, 2012.

[OT01] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Comput. Vision*, 42(3):145–175, May 2001.

[SH07] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *RBM*, 500(3):500, 2007.

[SH09] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

[SPKL15] Tom Sercu, Christian Puhrsch, Brian Kingsbury, and Yann LeCun. Very deep multilingual convolutional neural networks for LVCSR. *CoRR*, abs/1509.08967, 2015.

[SVI+15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[Wik17a] Wikipedia. Content-based image retrieval — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Content-based%20image%20retrieval`, 2017. [Online; accessed 19-August-2017].

[Wik17b] Wikipedia. Image retrieval — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Image%20retrieval`, 2017. [Online; accessed 19-August-2017].

[WSSJ14] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[Zau10] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. 2010.