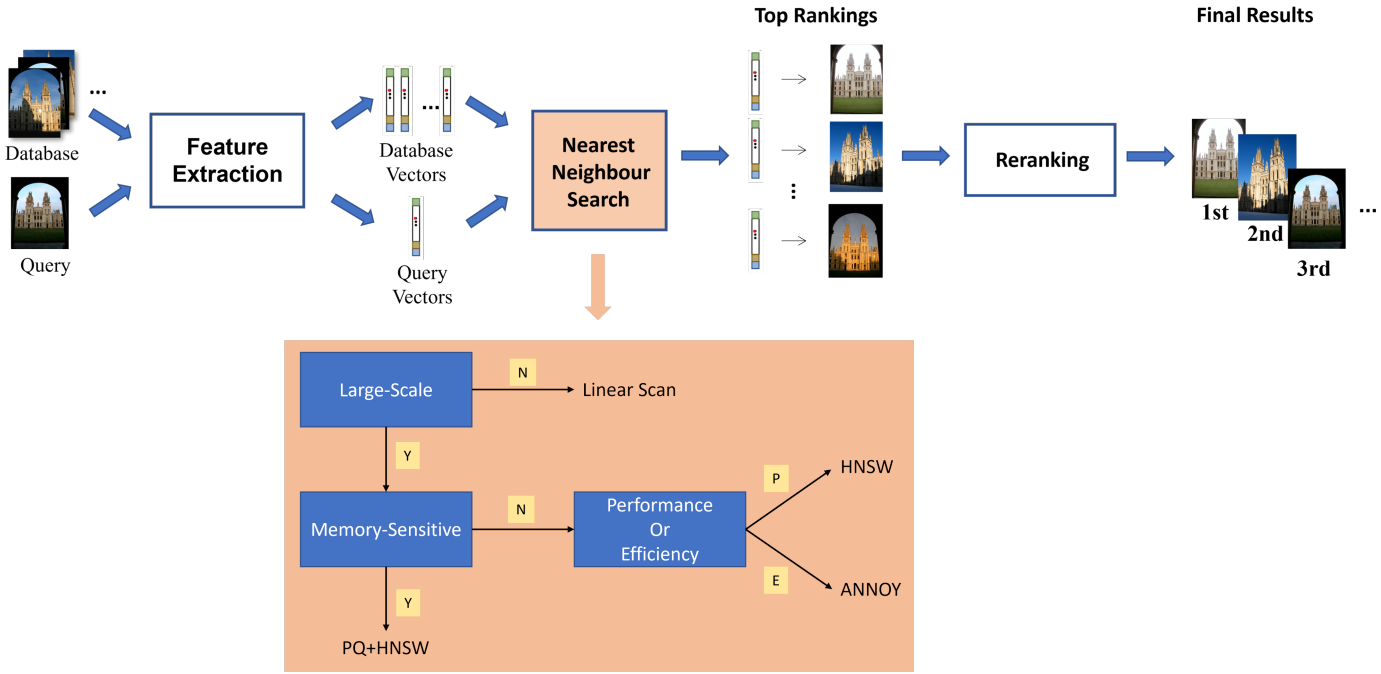
Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2022-5222893

M.Sc. Thesis

Image-Based Query Search Engine via Deep Learning

Yuanyuan Yao B.Sc.



Faculty of Electrical Engineering, Mathematics and Computer Science **Delft University of Technology**

Image-Based Query Search Engine via Deep Learning

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Electrical Engineering

by

Yuanyuan Yao B.Sc.
born in Zhejiang, China

This work was performed in:

Circuits and Systems Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Image-Based Query Search Engine via Deep Learning"** by **Yuanyuan Yao B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 22-07-2022

Chairman: _____
dr.ir. Justin Dauwels

Committee Members: _____
dr. Hadi Jamali-Rad

_____

Advisors: _____
Cristian Meo

_____
Yanbo Wang

Abstract

Typically, people search images by text: users enter keywords and a search engine returns relevant results. However, this pattern has limitations. An obvious drawback is that when searching in one language, users may miss results labelled in other languages. Moreover, sometimes people know little about the object in the image and thus would not know what keywords could be used to search for more information. Driven by this use case with many applications, content-based image retrieval (CBIR) has recently been put under the spotlight, which aims to retrieve similar images in the database solely by the content of the query image without relying on textual information.

To achieve this objective, an essential part is that the search engine should be able to interpret images at a higher level instead of treating them simply as arrays of pixel values. In practice, this is done by extracting distinguishable features. Many effective algorithms have been proposed, from traditional handcrafted features to more recent deep learning methods. Good features may lead to good retrieval performance, but the problem is still not fully solved. To make the engine useful in real-world applications, retrieval efficiency is also an important factor to consider while has not received as much attention as feature extraction.

In this work, we focus on retrieval efficiency and provide a solution for real-time CBIR in million-scale databases. The feature vectors of database images are extracted and stored offline. During the online procedure, such feature vectors of query images are also extracted and then compared with database vectors, finding the nearest neighbours and returning the corresponding images as results. Since feature extraction only performs once for each query, the main limiting factor of retrieval efficiency in large-scale database is the time of finding nearest neighbours. Exact search has been shown to be far from adequate, and thus approximate nearest neighbour (ANN) search methods have been proposed, which mainly fall into two categories: compression-based and tree/graph-based. However, these two types of approaches are usually not discussed and compared together. Also, the possibility of combining them has not been fully studied. Our study (1) applies and compares methods in both categories, (2) reveals the gap between toy examples and real applications, and (3) explores how to get the best of both worlds. Moreover, a prototype of our image search engine with GUI is available on https://github.com/YYao-42/ImgSearch.

Acknowledgments

This thesis concludes my study and research during the thesis project for the degree of Master of Science in Electrical Engineering at the Delft University of Technology. The project was performed from October 2021 to July 2022 in the Circuits and Systems group.

Academic research is never easy, full of unknowns, challenges, and confusion. I am fortunate to have Dr.ir. Justin Dauwels as my supervisor, who offered suggestions when I was flooded with too much information and overwhelmed by too many directions, meet me weekly and gave comments on my work. Thank you, Justin! Your passion and diligence inspire me a lot, and it has been an exciting and rewarding journey.

Also, many thanks to my daily co-supervisors, Cristian Meo and Yanbo Wang, who treat me as a friend, helped a lot with the details of the project and encouraged me to push forward. The project would not have progressed this smoothly without you.

My thesis project was carried out in parallel with Qi Zhang's and Yanan Hu's. We worked on the same image search engine but focused on different modules. The system would not exist without your excellent work on feature extraction and reranking.

Special thanks to Dr. Hadi Jamali-Rad for agreeing to join the thesis committee, Dr. Andrea Nanetti for revealing the application of content-based image retrieval in historical research, and the microelectronics department for the generous scholarship.

Lastly, the past two years have been struggling for all of us because of Covid-19. The chaos and isolation made me depressed, and I would not get through it without the support of my family and friends. Two years have gone by in a flash. For friends I got to know in TU Delft and are about to farewell, I believe this is not the end. Looking forward to the day we meet again.

Yuanyuan Yao B.Sc.
Delft, The Netherlands
22-07-2022

Nomenclature

Abbreviations

ANN Approximate Nearest Neighbour Search
ANNOY Approximate Nearest Neighbour Search Oh Yeah
CBIR Content-Based Image Retrieval
CNN Convolutional Neural Network
DG Delaunay Graph
DNN Deep Neural Network
EHM Engineering Historical Memory
FC Fully Connected
GH Greedy Hash
GUI Graphical User Interface
HNSW Hierarchical Navigable Small World
L2H Learning to Hash
LSH Locality Sensitive Hashing
mAP Mean Average Precision
NSW Navigable Small World
PQ Product Quantization
PQN Product Quantization Network
rmAP Relative Mean Average Precision
ROxford5k Revisited Oxford5k
RParis6k Revisited Paris6k

Contents

List of Figures

List of Tables

1 Introduction

Generally speaking, this project is about building a real-time image search engine that is purely content-based, searches with an image as a query, and returns similar images in the database. Despite there being various potential applications, we specifically care about the use case in historical research. In fact, it is open-source and will hopefully contribute to an ongoing research project Engineering Historical Memory (EHM) conducted by Dr. Andrea Nanetti, which aims to aggregate and deliver historical knowledge using emerging digital techniques such as artificial intelligence [63]. It is also worth noting that the proposed system is not limited to a specific application, but is suitable for all scenarios where people need to source or identify pictures and get more insights. For example, it can also be useful in medical diagnosis, security applications, remote sensing, etc.

More specifically, we break down the system into three main components: feature extraction, nearest neighbour search, and reranking. The motivation for such decomposition and the functions of each module are discussed in Section 1.1. The development and improvement of each module are done by three students respectively, and my part is nearest neighbour search. The importance of this module is discussed in Section 1.2, where the nearest neighbour search problem in the context of image retrieval is properly defined as well. Moreover, the structure of the rest of this thesis is explained in Section 1.3.

1.1 Pipeline

As mentioned before, in brief, the objective is to find similar images based on the content of the query image. But to what extent can two images be regarded as 'similar'? In this project, following the convention of the CBIR research, images taken from different angles, under different weather or light conditions are considered 'similar', as long as the main part of the images depicts the same figure or object [54, 86, 20]. Therefore, to retrieve similar images, the engine should be able to capture high-level information from low-level pixel values, which is usually done by feature extraction. Naturally, feature extraction is the first module in the pipeline, and the choice of specific methods will significantly affect the follow-up procedures.

1.1.1 Features: Vectors or Matrices?

Traditional feature extraction methods like SIFT [55], GIST [66] and SURF [5] mainly rely on hand-crafted features and have been proved to be insufficient for complex CBIR problems. Recently the spotlight is on deep learning methods, which can be roughly categorized into two classes: local feature extraction and global feature extraction.

Local Feature Extraction Local features are higher-level representations of specific regions and thus preserve spatial information. They can be effectively extracted, for example, by a convolutional neural network (CNN) [100]. However, a substantial part of those local features is unimportant and tends to introduce clutters and occlusions, so in practice, local feature extraction is usually followed by or performed together with keypoint selection, e.g., DELF [64], Superpoint [16], D2-Net [21], and R2D2 [72]. A (sparse) feature **matrix** will be generated, where the locations of non-zero entries indicate the locations of keypoints such as corners and blobs, and the values describe the information of image patches around the keypoints.

Global Feature Extraction Global features are more abstract and compact representations, but the spatial information is lost. Most common implementations are also CNN-based, adding fully connected layers after the last convolutional layer and regarding the outputs as global features [3], or aggregating local features extracted by CNN using aggregation methods like BoW [70], R-MAC [84], CroW [41], and GeM pooling [71]. Apart from CNN-based methods, methods using autoencoder [75], attention network [81] and generative adversarial [78] have also been explored. Different from local feature extraction, the output now is a feature **vector**.

Generally speaking, local features preserve more information and allow us to do more delicate matching, with which we tend to get better performance on matching tasks. However, image retrieval is very different from template matching: apart from being accurate, it also imposes limitations on the matching time. Normally users would expect that each query will take just a blink, which is almost unachievable for large-scale datasets if we use local feature extraction. The matching is pairwise, performing between two (usually large) matrices. Even if we manage to compress the matching time of each pair, it is still a huge cost in, e.g., a million-scale dataset.

Global feature extraction, on the other hand, generates feature vectors. Different from matching two matrices, the similarity measurement of vectors, e.g. Euclidean distance, is much more intuitive and easier to calculate. Moreover, many efficient methods have been proposed to accelerate the procedure of finding the closest vectors in the database by data compression and non-exhaustive search.

Therefore, in consideration of the matching efficiency, we choose to extract feature vectors instead of feature matrices.

1.1.2 What's Next?

Since the features are extracted as vectors, we can forget about complex algorithms like random sample consensus [24] suitable for keypoint matching and just consider the rest as a nearest neighbour search problem. The idea is to find the nearest vectors of the query's feature vector in the database and retrieve their corresponding images. The following steps will be shown in Section 1.2.

By now the goal of the image search engine can already be fulfilled. However, instead of returning the results to users directly, we can consider the whole process so far as an initial screening. Recall that in Section 1.1.1 the main constraint of using local features is the matching time in large-scale datasets. Imagine that if we confine the scope of the matching to be, e.g., 100 screened images, the matching time might be acceptable. We can then exploit the advantage of local features to perform more delicate

and hopefully, more accurate matching with a reasonable time overhead. Therefore, a reranking module is appended after the nearest neighbour search.

1.1.3 The Whole Picture

Based on the discussion above, we propose the pipeline illustrated in Fig. 1.1. The pipeline is used in two stages:

- **Offline (development)**: In this stage the global features of database images are extracted by a feature extractor and stored for subsequent queries. Depending on the nearest neighbour search methods, the features may be encoded as compact codes, or/and organized as a tree/graph (will be discussed in detail in Section 3). The codes, codebook, tree and graph will also be stored if needed. If reranking is required, local features will also be extracted and saved.

- **Online (deployment)**: This stage starts after users input query images. The global (and local) features for screening (and reranking) are also extracted, and then compared with the features of database images stored beforehand.

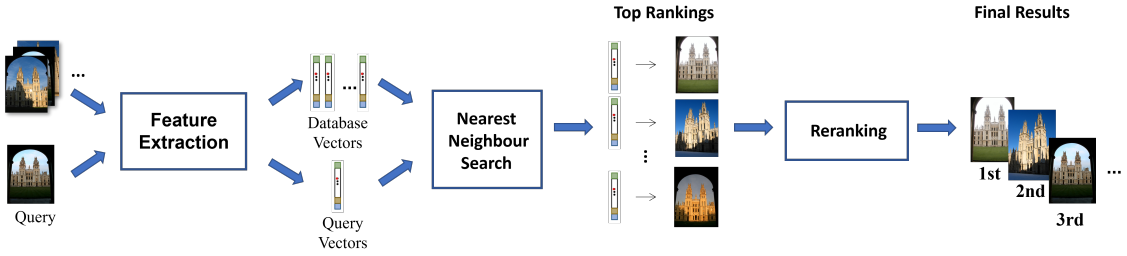


Figure 1.1: Proposed pipeline. First, features are extracted from the query image. Similarly, such features are also extracted from all images in the database. These features are extracted only once, and are then stored for processing the image-based queries. The extracted features of all database images and query image are then forwarded to an approximate nearest-neighbour search module to produce the initial ranking results, which are next refined by a re-ranking module, generating the final results.

The system is highly modular and therefore highly flexible, allowing for easier adaptation to requirements, striking a balance between speed and accuracy. Moreover, each module does not impose additional restrictions on other modules. They can be tested, improved independently and integrated into the system effortlessly.

1.2 Nearest Neighbour Search: From Exact to Approximate

The focus of this thesis is on the nearest neighbour search module, which basically determines how long each query will take in a large-scale database since the feature extraction is only performed once independently of the size of the database. Therefore, the design of this module is crucial to real-time retrieval.

To make the discussion clearer, let us introduce some notations. Denote the query vector, i.e., the feature vector of the query image as $\mathbf{q} \in \mathbb{R}^D$. Similarly, features of the images in the database can be represented by database vectors $\mathcal{X} = \{\mathbf{x}_n\}_{n=1}^N, \mathbf{x}_n \in \mathbb{R}^D$. The mathematical formulation for finding the relevant image is then finding the database vector that has the 'closest' distance to the query vector, where the closeness can be defined by any distance measure and a typical choice is ℓ_2 norm (Euclidean distance):

$$n^* = \underset{n \in \{1,\cdots,N\}}{\operatorname{argmin}} \|\mathbf{q} - \mathbf{x}_n\|_2^2, \tag{1.1}$$

which can be generalized to K-nearest neighbour search problem easily.

A trivial way to solve this problem is brute-force linear scan, i.e., calculating the distance between every database vector and the query vector to find the nearest one. For each pair, it takes D calculations and there are N pairs. Therefore, the computational complexity is $\mathcal{O}(ND)$, which might be unacceptable for real-time search when N is very large. More efficient algorithms for **exact** nearest neighbour search have been discovered in, e.g., [73, 51]. Unfortunately, they can only be efficient when the points lie in a space of constant dimension and become less and less efficient as the dimension increases [32]. In short, exact search suffers from *the curse of dimensionality* and does not scale well.

Therefore, researchers tend to apply **approximate** nearest neighbour (ANN) search, which does not aim to retrieve exact nearest neighbours, but approximate ones, in trade of lower time or space cost. From the search complexity of linear scan, intuitively, we can quickly come up with ideas to reduce the computational cost from two aspects:

1. By reducing the dimensionality of the data, which leads to compression-based methods. The basic idea is to compress the feature vector or encode it into a much more compact representation, and thus simplify the calculation of the distance.

2. By reducing the candidates to be compared, which leads to tree-based and graph-based methods. They usually enable us to calculate and compare distances between only a small portion of the database vectors and the query vector.

If we think further, it is not difficult to realise the possibility of combining compression-based and tree/graph-based methods. Compress the original vectors into compact codes, and organize them in a way such that we can do a non-exhaustive search.

1.3 Outline

The rest of the paper is organized as follows:

- Chapter 2 is a literature review of ANN methods. We review both compression-based and tree/graph-based methods.

- Details of methods that have been tested and explored during the thesis project are shown in Chapter 3. The motivation of why specific methods are selected among various other options is also discussed.

- Chapter 4 gives an overview of our experimental results. The definitions of the metrics used throughout this project will be introduced first. Before each method is incorporated into the pipeline, a function verification experiment is performed on a prototype. The results of different methods on the prototype are shown first, followed by the results of valid methods when integrated into the system.
- Conclusions will be drawn in Chapter 5.

2 Related Work

2.1 Compression-based Methods

When considering data compression or dimension reduction, one may immediately think of principle component analysis, multi-layer perceptron or autoencoders. However, they are not the focus of this section because in many cases they have already been included as a step in feature extraction. For example, the convolutional layers are usually followed by fully connected layers, which group into a multi-layer perceptron. To avoid overlapping, here we only review compression-based methods that are specially designed for accelerating nearest neighbour search.

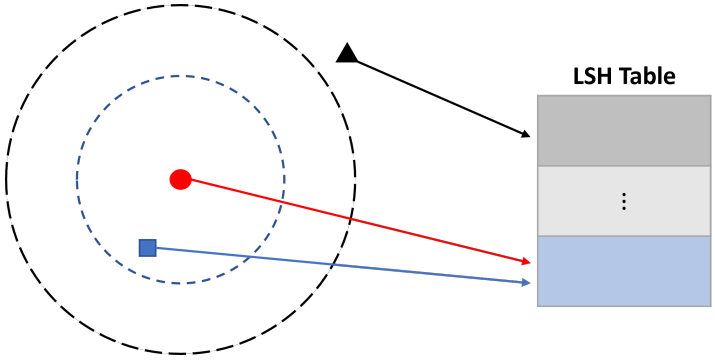
2.1.1 Data-independent

The methods in this category are locality sensitive hashing (LSH) and its variants. LSH was first introduced in [32], where a family of functions called LSH family $\mathcal{H}$ was defined. For any hash function h in the LSH family, it satisfies two conditions:

- if $\mathcal{D}(\mathbf{v}_1, \mathbf{v}_2) \leq R$, then $\mathcal{P}[h(\mathbf{v}_1) = h(\mathbf{v}_2) \geq P_1]$,
- if $\mathcal{D}(\mathbf{v}_1, \mathbf{v}_2) \geq cR$, then $\mathcal{P}[h(\mathbf{v}_1) = h(\mathbf{v}_2) \leq P_2]$,

where $c > 1$, $P_1 > P_2$, $\mathbf{v}_1, \mathbf{v}_2$ are any two items, $\mathcal{D}(\cdot)$ denotes a distance measurement, and $\mathcal{P}[\cdot]$ is the probability of a statement. h is then called (R, cR, P_1, P_2)-sensitive. Intuitively, such h tends to map 'close' items in the original space to the same representation in the new space, and map items that are far away from each other to different representations, as illustrated in Fig. 2.1.

Figure 2.1: The red dot denotes $\mathbf{v}_1$. If $\mathbf{v}_2$ is within the blue circle ($\mathcal{D}(\mathbf{v}_1, \mathbf{v}_2) \leq R$), it tends to be mapped to the same bucket as $\mathbf{v}_1$. If $\mathbf{v}_2$ is outside the black circle ($\mathcal{D}(\mathbf{v}_1, \mathbf{v}_2) \geq cR$), it tends to be mapped to a different bucket.



Various distance measurements lead to different algorithms. In the original paper, the authors apply Hamming distance. Leech lattice LSH [1], however, works in the Euclidean space. There are methods based on more complex, self-defined distance or similarity measurements as well. For example, methods like Super-bit LSH [36] and Kernel LSH [44] use angle-based distance. Algorithms derived from cosine similarity [23], Jaccard similarity [35, 76], rank similarity [93] are also discovered.

Given a hash function h, any item $\mathbf{x}$ can be mapped into a bucket $h(\mathbf{x})$. Mapping all the items will give us a hash table. For a query $\mathbf{q}$, it will also be assigned into a bucket and we just return all the items in the same bucket as nearest neighbours according to the hash table. To improve the recall, LSH forest proposes to build multiple tables and represents them as trees [4], [34] proposes a relevance value to decide the most relevant hash codes, and [56] proposes to probe multiple buckets. There are also studies about fundamentally different search methods, e.g., [67].

No matter what distance measurements or search methods are used, methods in this category is data-independent since the hash functions are already determined before any data is fed. This, however, does not fully leverage the information in the dataset and leads to inferior recall rate. In practice, data-dependent methods generally perform better and are much more preferred.

2.1.2 Data-dependent

2.1.2.1 Learning to Hash (L2H)

The keyword for L2H is similarity preserving: similar items in the original space should also be similar in the coding space. A compound hash function $\mathbf{h}$ is learned from data, mapping inputs $\mathbf{x}$ to much more compact representations (hash codes) $\mathbf{b}$, i.e.,

$$\mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_L \end{bmatrix} = \begin{bmatrix} h_1(\mathbf{x}) \\ \vdots \\ h_L(\mathbf{x}) \end{bmatrix} = \mathbf{h}(\mathbf{x}) \tag{2.1}$$

where L is the length of the code, $\{b_i\}_{i=1}^{L}$ are integers or binary values, and $h(\cdot)$ are hash functions.

Given two items $\mathbf{x}_1$ and $\mathbf{x}_2$ and their corresponding hash codes $\mathbf{b}_1$ and $\mathbf{b}_2$, we can define the similarity in the original input space $s_{ij}^o(\mathbf{x}_1, \mathbf{x}_2)$ and the similarity in the coding space $s_{ij}^h(\mathbf{b}_1, \mathbf{b}_2)$. Preserving similarity means s_{ij}^o and s_{ij}^h should be close, which can be achieved by choosing an appropriate loss function, formulating an optimization problem (with constraints), and designing an algorithm to solve it. To simplify the optimization problem or to provide an analytic result, sometimes the form of hash functions is restricted. Therefore, essentially the methods in L2H are different on:

1. The form of hash functions. Frequently used choices are linear functions [80], kernels [30], non-parametric functions [74]. Deep neural networks (DNN) become more and more popular recently, and we will save the details for Section 2.1.2.2.

2. The definitions of s_{ij}^o and s_{ij}^h. Similarities are usually defined through distances. For example, from the Euclidean distance in the original space d_{ij}^o and the Ham-

ming distance in the coding space d^h_{ij}, we can define $s^o_{ij} = \exp\{-(d^o_{ij})^2/2\sigma^2\}$ and $s^h_{ij} = L - d^h_{ij}$. Cosine similarity is also a typical choice for s^o_{ij}.

3. The definitions of the loss function, which can be divided into many groups, e.g., minimizing the similarity-distance product ($\sum s^o_{ij} d^h_{ij}$ or $\sum s^h_{ij} d^o_{ij}$) [91, 79], maximizing similarity-similarity product $\sum s^o_{ij} s^h_{ij}$ [88, 89], minimizing similarity-similarity difference $\sum (s^o_{ij} - s^h_{ij})^2$ [19, 38].

4. The techniques to solve the optimization problem. The fact that codes are discrete integers will introduce constraints, and the main difficulty lies in how to deal with them. For example, if the codes are binary, then $\text{sgn}(\cdot)$ function will occur in the objective function, or we can drop $\text{sgn}(\cdot)$ and introduce a constraint $b_i = \{0, 1\}$. Frequently used workarounds are relaxing the sgn function as a sigmoid function or a tanh function, ignoring sgn during the optimization and bringing it back when generating the final result, or applying the coordinate-descent approach [87].

The discussion above only includes pairwise similarity preserving, i.e., retaining the similarity between a pair of items. Researchers also come up with multiwise similarity preserving, which considers a tuple of items. A typical example is triplet loss hashing [65, 14]. The triplet tuple has an anchor item $\mathbf{x}$, a positive item $\mathbf{x}^+$ (that is similar to the anchor), and a negative item $\mathbf{x}^-$ (that is not similar to the anchor). The loss function is designed to maximize the Hamming distance between $(\mathbf{x}, \mathbf{x}^-)$ and minimize the Hamming distance between $(\mathbf{x}, \mathbf{x}^+)$.

Another type of method that follows a slightly different rationale is quantization-based method, which aims to minimize the reconstruction error [40, 97, 98]. The similarity between codes is not measured directly; instead, the codes will be first transformed back to the original space via the codebook, and then compare the similarity (symmetric distance computation). Alternatively, only the database vectors are encoded and then decoded to compare with the query vector (asymmetric distance computation). This may seem inefficient, but the computation between the reconstructed vectors or the reconstructed vector and the query will be simplified compared to the direct computation using original vectors. A concrete example will be given in Section 3.1.

2.1.2.2 Deep Learning Methods

Due to the great potential of DNN to approximate more complex functions, they are heavily studied to replace traditional linear hash functions in recent years. New methods in this area usually fall into this category, so a separate section is created to discuss deep learning methods specifically.

Among them, supervised deep learning methods have been widely studied. Apart from the definitions of similarity, they differ mainly on how the label information is exploited, or say how the loss function is defined. Methods like SDH [22], DSH [53], SHBDNN [18] and DDSH [37] only use labels to determine whether two images are similar and thus no extra term is introduced in the loss function. In [10, 11, 46], a classification layer is added to provide semantic supervision. If one-hot encoding is

applied, the label information can also be included in a linear regression loss as in [12, 47]. Quantization-based deep hashing methods like [96, 52, 42] are studied as well.

Supervised methods heavily rely on label information, which is not always desirable. More and more research focus on deep unsupervised hashing, which is first proposed in [43]. A simplest attempt is to drop the terms about similarity preserving and semantic information in the loss function and instead aim to minimize the quantization error only with, e.g., autoencoders [50]. Alternatively, generative models can also be used to learn hash codes [31, 13, 17].

Generally speaking, methods in this category tend to have better performance than LSH and L2H, but they have limitations as well. For example, as any other deep learning methods, they have high demand for data, otherwise are very likely to suffer from overfitting. Besides, the tests are toy examples done in relatively simple datasets like CIFAR10, so the performance in real-world image retrieval applications is unknown. Moreover, in practice we would like to train our model end-to-end, meaning that the DNN for generating good hash codes need to be incorporated into the DNN that extracts features, which is normally not a trivial task since the loss function may need to be re-designed.

2.2 Tree/Graph-based Methods

Compression-based methods can make the distance calculation between two vectors faster, and methods in this section focus on non-exhaustive search. To find the nearest neighbours, checking all the candidates in the database is not really necessary. It can be shown that, by properly organizing the database vectors, only a small portion of the candidates is needed to be visited for finding desired results.

2.2.1 Tree-based Methods

Tree-based methods are designed for efficient searching by partitioning space into multiple disjoint regions and organising the data points as a tree based on which region they belong to. If two points reside in adjacent regions, then they will locate closely in the tree as well. Therefore tree-based methods are also called partition-based methods.

Among all the methods in this category, kd-tree [26] is probably the best-known one. For a balanced kd-tree with randomly distributed points, the search complexity is $\mathcal{O}(\log N)$ on average. The worst case, however, can be $\mathcal{O}(DN^{1-1/D})$ [45]. (To be consistent with the rest of the thesis, the dimension of the vector is still denoted as D instead of k as in literature about kd-tree.) Therefore, for high dimensional vectors, the search complexity of kd-tree may be very comparable to the exact search, which is the reason why it is rarely applied in image retrieval tasks, where the feature vectors are usually high dimensional.

Many variations of kd-tree have been explored for better search performance or faster search speed, e.g., PCA-trees [61], cover trees [8], ball trees [9] and NKD-tree [77]. Empirically, kd-tree also shows good results after introducing randomization and overlapping cells, based on which a library FLANN is built and widely used in nearest

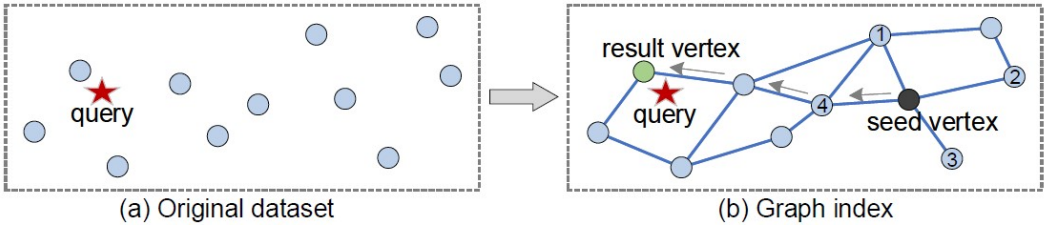
neighbour search problems [62]. Inspired by this, RPTree [15] has been proposed with theoretical guarantees.

ANNOY [6], a method proposed by a practitioner, is also worth mentioning. It partitions regions randomly without any operations like rotation or projection on data points, but works surprisingly well empirically even with high dimensional data. In fact, it is one of the best tree-based methods according to the experiments in [49].

2.2.2 Graph-based Methods

Intuitively, if point A is close to point B, then it is very likely that the neighbours of A are also close to B. Given a query point, if we can exploit the neighbour relationship between points, the searching of its closest point might be greatly simplified. How to express the neighbour relationship? A natural thought is using graphs and Fig. 2.2 shows a simple example of how it looks like. The vertices are the data points in the

Figure 2.2: An illustration for general graph-based algorithms [90]. The nodes (vertices) are the data points, neighbours of which are connected by edges. The search starts with a random seed vertex and the next goes to its neighbour that is closest to the query. Do it iteratively until the node itself is closer to the query than any of its neighbours.



original dataset, which will be connected by edges if they are close to each other. To search the nearest neighbour of the query, we apply greedy search: first randomly assign an entry point or say a seed vertex, of which the neighbours will be found by traveling through its edges. We choose the neighbour that is closest to the query as the new seed vertex and repeat the aforementioned procedure. It will terminate if the seed vertex is closer to the query than all of its neighbours, then the seed vertex will be returned as the nearest neighbour of the query.

Now a question remain is how to build the graph such that we can retrieve good results efficiently using greedy search. There is a kind of graph that ensures greedy search returns precise results called Delaunay Graph (DG) [25]. However, it also suffers from 'the curse of dimensionality': in high-dimensional space, it is almost fully connected and thus has little advantage over linear scan [29]. RNG [85] tackles this problem by dropping the redundant neighbors and making the rest distribute omni-directionally. However, a limitation is that the construction complexity is $\mathcal{O}(N^3)$, making the time of construction prohibitive even with moderate-scale datasets. Methods like NSW [58], HNSW [57], DiskANN [33] have been proposed to alleviate this problem.

There are also K-Nearest Neighbour Grpaph (KNNG)-based methods, which need extra operations to ensure global connectivity [99, 68]. Methods that based on both DG (or RNG) and KNNG [49, 28, 27] are popular as well.

3 Methods

As we can see from the literature review in Chapter 2, ANN is a field that has been extensively studied and many different types of approaches have been proposed. For algorithms that belong to the same category, it is not difficult to find literature that conducts comparative studies, having a rough idea of which ones are better under what circumstances. However, cross-disciplinary comparisons are very rare and do not pay special attention to the image retrieval problem and its commonly used dataset and metric [2, 49]. Therefore, we do not know which methods are the best of all. In fact, from the results in [2, 49], it can be concluded that no method is always the best. The performance depends very much on the specific dataset.

Considering the issues above, we decided to select and test methods from different categories and incorporate the most suited ones into our final pipeline. The following conditions were taken into account during selection:

- The methods must not impose any extra restrictions on other modules and the structure of the pipeline.
- The methods should be the state-of-art in their corresponding category (at least in some datasets).
- Methods that are not cutting-edge but have been widely applied in real-world applications should also be considered.
- For supervised methods, they must not require label information other than whether the two images are similar or not.

Based on the criteria above, we selected the following methods:

- Product Quantization (PQ) [40]. Its performance is one of the best in traditional L2H. Although it has been surpassed by deep hashing, PQ is still the go-to method in real-world applications.
- Product Quantization Network (PQN) [95], a deep neural network version of PQ. A PQ layer is designed to be integrated into the feature extractor to generate more compatible codes, leading to better performance.
- Greedy Hash (GH) [83]. It is also a deep hashing method, but different from PQN, it generates binary codes and preserves similarity in the coding space. Both GH and PQN do not impose restrictions on the structure of the feature extractor and the loss function.
- ANNOY [6]. One of the best in tree-based methods. Although no theoretical guarantee can be provided, it shows competitive results in [2, 49] and thus has been highly preferred by many practitioners.

- Hierarchical Navigable Small World (HNSW) [57]. State-of-the-art in graph-based methods. Has a theoretical guarantee for logarithmic search complexity.

Some of the above methods (PQ, ANNOY, HNSW) have been widely used, while PQN and GH have been proposed but, to the best of our knowledge, never applied in practice. The details will be shown concisely one by one in the rest of this chapter. Moreover, we propose two hybrid methods, PQ+HNSW and GH+ANNOY, which will also be discussed, implemented, tested and compared. Therefore, in this study we explore numerous methods for ANN.

3.1 Product Quantization (PQ)

Basic PQ was originally proposed in [40]. The idea is to split a vector into sub-vectors and quantize each sub-vector. More specifically, given a database vector $\mathbf{x} \in \mathbb{R}^D$, one can view it as a concatenation of M sub-vectors:

$$\mathbf{x} = [\underbrace{x_1, x_2, \dots, x_{D/M}}_{\mathbf{x}^{1\mathrm{T}}}, \dots, \underbrace{x_{D-D/M+1}, \dots, x_D}_{\mathbf{x}^{M\mathrm{T}}}]^{\mathrm{T}} \tag{3.1}$$

where $\mathbf{x}^m \in \mathbb{R}^{D/M}, m \in \{1, ..., M\}$. If plenty of database vectors are available, we can train a sub-codebook for each sub-vector by, for instance, K-means clustering. The sub-codebook for each $m \in \{1, ..., M\}$ is denoted as $\mathcal{C}^m = \{\mathbf{c}_k^m\}_{k=1}^K$, where K is the number of sub-codewords. The sub-vector can then be encoded as the index of the nearest sub-codeword, i.e.,

$$i^m(\mathbf{x}^m) = \underset{k \in \{1,...,K\}}{\operatorname{argmin}} \|\mathbf{x}^m - \mathbf{c}_k^m\|_2^2 \tag{3.2}$$

where $i^m : \mathbb{R}^{D/M} \to \{1, ..., K\}$ is the sub-encoder for m-th sub-vector. The vector $\mathbf{x}$ can then be encoded as the concatenation of the encoded sub-vectors:

$$\mathbf{i}(\mathbf{x}) = [i^1(\mathbf{x}^1), ..., i^M(\mathbf{x}^M)]^{\mathrm{T}} \tag{3.3}$$

where $\mathbf{i} : \mathbb{R}^D \to \{1, ..., K\}^M$. Apparently, the reconstruction will not be lossless. We can obtain the approximation of $\mathbf{x}$ by

$$\tilde{\mathbf{x}} = \mathbf{i}^{-1}(\mathbf{i}(\mathbf{x})) = \mathbf{i}^{-1}\left(\begin{bmatrix} i^1(\mathbf{x}^1) \\ \vdots \\ i^M(\mathbf{x}^M) \end{bmatrix}\right) = \begin{bmatrix} \mathbf{c}_{i^1}^1 \\ \vdots \\ \mathbf{c}_{i^M}^M \end{bmatrix}. \tag{3.4}$$

How can PQ simplify the calculation of distance? It can be illustrated by the following decomposition:

$$d(\mathbf{q}, \mathbf{x})^2 \approx d(\mathbf{q}, \tilde{\mathbf{x}})^2 = \sum_{m=1}^{M} d(\mathbf{q}^m, \mathbf{c}_{i^m}^m)^2, \tag{3.5}$$

where $\mathbf{q}^m$ are the sub-vectors of the query vector. Since each $\mathbf{c}_{i^m}^m$ only has K possible values, we can compute every possible $d(\mathbf{q}^m, \mathbf{c}_{i^m}^m)^2$ beforehand and save them into a $M \times K$ distance table $\mathbf{A}$. Then calculating the distance only takes M table look-ups:

$$d(\mathbf{q}, \mathbf{x})^2 \approx \sum_{m=1}^{M} d(\mathbf{q}^m, \mathbf{c}_{i^m}^m)^2 = \sum_{m=1}^{M} \mathbf{A}(m, i^m). \tag{3.6}$$

Therefore, with PQ, the computational complexity for each query can be reduced from $\mathcal{O}(DN)$ to $\mathcal{O}(MN)$. Practically, M is a much smaller number like 2, 4, or 8, so the reduction in computation is significant. Also, after encoding, each vector only takes $M \log_2 K$ bits, which usually leads to huge memory savings.

Extension Denote $\mathbf{C}^m = [\mathbf{c}_1^m, \cdots, \mathbf{c}_K^m]$. We can write the encoding part of PQ more compactly as:

$$\min_{\{\mathbf{b}^m\}_{m=1}^M} \left\| \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^M \end{bmatrix} - \begin{bmatrix} \mathbf{C}^1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{C}^2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{C}^M \end{bmatrix} \begin{bmatrix} \mathbf{b}^1 \\ \mathbf{b}^2 \\ \vdots \\ \mathbf{b}^M \end{bmatrix} \right\|_2^2 \triangleq \min_{\{\mathbf{b}^m\}_{m=1}^M} \|\mathbf{x} - \mathbf{C}\mathbf{b}\|_2^2, \tag{3.7}$$

where $\mathbf{b}_m$ are indicator vectors with length equals to L/M. Following the notation in (3.2), the i^m-th element of $\mathbf{b}^m$ will be 1 and others will be 0. In PQ, $\mathbf{C}$ is not optimized but settled beforehand by K-means clustering. What if we regard $\mathbf{C}$ as a variable that also needs to be optimized? If we consider all the database vectors, then (3.7) should be rewritten as:

$$\min_{\mathbf{C}, \{\mathbf{b}^n\}_{n=1}^N} \sum_{n=1}^{N} \|\mathbf{x}_n - \mathbf{C}\mathbf{b}_n\|_2^2, \tag{3.8}$$

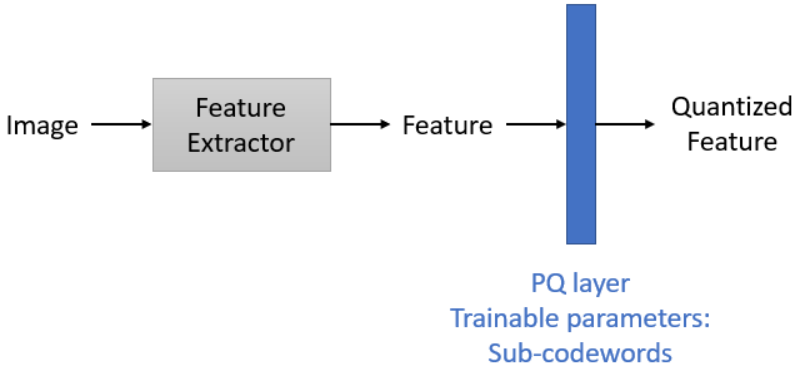
where $\mathbf{x}_n$ are database vectors and $\mathbf{b}_n$ are the corresponding (binary) codes. Ideally solving (3.8) will give us a better result, at least in the context of minimizing quantization error, but it is a more difficult problem. Different constraints on $\mathbf{C}$ and optimization tools lead to different variants of classic PQ.

3.2 Product Quantization Network (PQN)

PQN [95] is an extension of PQ, whose sub-codewords are not determined beforehand. What makes it unique is that it is a layer that can be easily integrated into standard DNNs. By doing so, we may train the features and sub-codewords together to generate more compatible codes and, hopefully, achieve better performance than classic PQ.

An illustration of how does a typical PQN look like is shown in Fig. 3.1. The only difference compared to the standard structure is that a PQ layer is inserted after the feature extractor. If we would like to make minimal changes to the original training procedure, then we can just replace the feature in the loss function with the quantized version instead and keep all the rest parts the same. A more sophisticated alternative is to output both original features and quantized features and design a new loss function.

Figure 3.1: A typical PQN. The original features are forwarded into a PQ layer, which generates quantized features. Features and sub-codewords can be trained together end-to-end.



Designing a quantization layer, unfortunately, is not so straightforward. For a layer in a neural network, a requirement that must be satisfied is that the operations involved in the layer must be differentiable so that parameter updates can be performed using back propagation. However, when we encode the sub-vectors (see (3.2)), the argmin function is non-differentiable. Therefore, researchers proposed 'soft' quantization in [95]. Let us first represent the quantized sub-vectors $\tilde{\mathbf{x}}^m$ in a slightly different way:

$$\tilde{\mathbf{x}}^m = \sum_{k=1}^{K} \mathbb{I}(k = k^*)\mathbf{c}_k^m, \tag{3.9}$$
$$\text{where } k^* = \underset{k \in \{1,\dots,K\}}{\operatorname{argmin}} \|\mathbf{x}^m - \mathbf{c}_k^m\|_2^2.$$

$\mathbb{I}(\cdot)$ is the indicator function, which is non-differentiable. An easy fix is to find its differentiable approximation, e.g.,

$$\mathbb{I}(k = k^*) = \lim_{\alpha \to +\infty} \frac{e^{-\alpha \|\mathbf{x}^m - \mathbf{c}_k^m\|_2^2}}{\sum_{k'} e^{-\alpha \|\mathbf{x}^m - \mathbf{c}_{k'}^m\|_2^2}}. \tag{3.10}$$

By plugging (3.10) into (3.9) with an appropriate α, the modified version of the quantized sub-vectors is

$$\tilde{\mathbf{x}}^m = \sum_{k=1}^{K} \frac{e^{-\alpha \|\mathbf{x}^m - \mathbf{c}_k^m\|_2^2}}{\sum_{k'} e^{-\alpha \|\mathbf{x}^m - \mathbf{c}_{k'}^m\|_2^2}} \mathbf{c}_k^m, \tag{3.11}$$

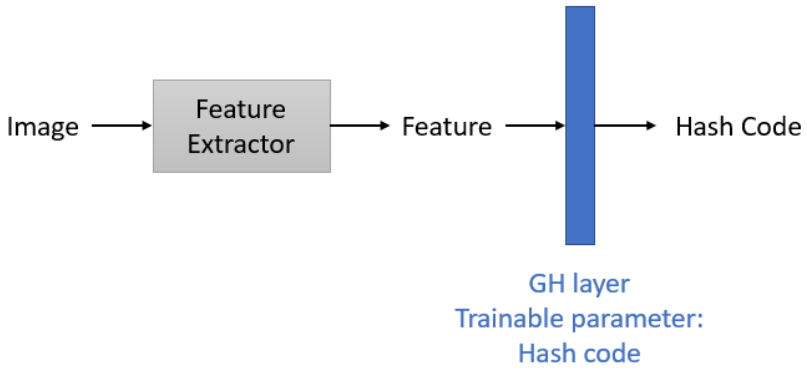
which defines our PQ layer. The inputs are the original feature vectors, and the outputs are quantized feature vectors. The layer is ready to be plugged into any neural network and the sub-codewords $\mathbf{c}_k^m$ will be trained with features. Once the neural network has been trained, we will get optimized sub-codewords, and other procedures are exactly the same as in classic PQ.

3.3 Greedy Hash (GH)

As mentioned in Section 2.1.2.1, quantization-based methods are not typical L2H methods: they do not aim to preserve similarity in the coding space; instead, the key point is more about nicely reconstructing the original vectors based on the codes. The ℓ_2 distance (if we work with $\mathbf{i}$ in (3.3)) or the Hamming distance (if we work with $\mathbf{b}$ in (3.7)) between two codes indicates nothing if the codebook $\mathbf{C}$ is not given. It is a big limitation if we want to combine PQ with tree-based methods (which will be discussed later). Therefore, we would like to also keep a more L2H-like approach in the list and GH is selected due to its flexibility and relatively good performance.

Essentially, L2H is about finding good hash functions that map the original feature vector $\mathbf{x}$ to a compact code $\mathbf{b}$ given a loss function. Different from traditional methods, GH regards a DNN as a hash function [83]. The powerful representation capabilities of DNNs make it possible to approximate any complex hash functions. Also, as in the case of PQN, hash codes can be learned together with features if we incorporate GH into the neural network (Fig. 3.2).

Figure 3.2: Incorporating GH into the system. The pattern is almost the same as the case in PQN (Fig. 3.1), except that the output now is hash code.



GH is designed to solve the following problem:

$$\begin{aligned} &\min_{\mathbf{B}} \mathcal{L}(\mathbf{B}), \\ &\text{s.t. } \mathbf{B} \in \{-1, +1\}^{N \times L}, \end{aligned} \tag{3.12}$$

where $\mathcal{L}(\cdot)$ can be any loss function and $\mathbf{B}$ are hash codes of N feature vectors. There are no restrictions on the form of loss functions. If we simply ignore the discrete constraint, (3.12) can be solved by gradient descent:

$$\mathbf{B}^{t+1} = \mathbf{B}^t - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{B}^t}, \tag{3.13}$$

where η is the learning rate. Then we apply the greedy principle, taking the closest discrete point to $\mathbf{B}^{t+1}$ in (3.13) as the optimal solution:

$$\mathbf{B}^{t+1} = \text{sgn}(\mathbf{B}^t - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{B}^t}). \tag{3.14}$$

However, $\text{sgn}(\cdot)$ is a non-differentiable function. How to incorporate it into the neural network?

Let us rewrite (3.14) into two parts:

$$\mathbf{B}^{t+1} = \text{sgn}(\mathbf{H}^{t+1}), \tag{3.15}$$

$$\mathbf{H}^{t+1} = \mathbf{B}^t - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{B}^t}, \tag{3.16}$$

where $\mathbf{H}$ is an intermediate variable that can be interpreted as the output of the neural network. Therefore, (3.15) shows the forward propagation and the question is what is the backward propagation from $\mathbf{B}^{t+1}$ to $\mathbf{H}^{t+1}$ (or from $\mathbf{B}^t$ to $\mathbf{H}^t$). We want to bypass the non-differentiable problem and define the back propagation in such a way that the update equation has the same form as in (3.16). This can be done by introducing a penalty term $\|\mathbf{H} - \text{sgn}(\mathbf{H})\|_p^p$ (entry wise matrix norm) into the loss function, which will be made as close to zero as possible. By doing so, the update equation of $\mathbf{H}$ using gradient descent can be written as

$$\begin{aligned}
\mathbf{H}^{t+1} &= \mathbf{H}^t - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{H}^t} \\
&= (\mathbf{H}^t - \text{sgn}(\mathbf{H}^t)) + \text{sgn}(\mathbf{H}^t) - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{H}^t} \\
&\approx \mathbf{B}^t - \eta * \frac{\partial \mathcal{L}}{\partial \mathbf{H}^t}.
\end{aligned} \tag{3.17}$$

Comparing (3.17) to (3.16), our object can be achieved by setting

$$\frac{\partial \mathcal{L}}{\partial \mathbf{H}^t} = \frac{\partial \mathcal{L}}{\partial \mathbf{B}^t}, \tag{3.18}$$

that is, back transmitting the gradient of $\mathbf{B}^t$ to $\mathbf{H}^t$ intactly.

To summarize, the input of the GH layer is the output of the neural network $\mathbf{H}$, and the output of the GH layer is the hash code $\mathbf{B}$. The forward propagation and the backward propagation are defined as

$$\begin{aligned}
\text{Forward: } & \mathbf{B} = \text{sgn}(\mathbf{H}), \\
\text{Backward: } & \frac{\partial \mathcal{L}}{\partial \mathbf{H}^t} = \frac{\partial \mathcal{L}}{\partial \mathbf{B}^t}.
\end{aligned} \tag{3.19}$$

3.4 ANNOY

Tree-based methods mainly differ on how the space is partitioned. For example, in kd-tree, the partition plane is always perpendicular to an axis, passing through a pivot element, which is usually the median of the sorted coordinates in the selected axis. Each partition splits the original space into two subspaces, which will be partitioned recursively until a certain condition, e.g., the maximum number of data points in a region, is satisfied. Different from kd-tree, partitioning in ANNOY does not follow a rigorous or mathematical way, but rather random [6]. Building the tree and searching with the tree are almost the same among different methods.

More specifically, in ANNOY, a tree can be built in parallel with space partitioning following the steps below:

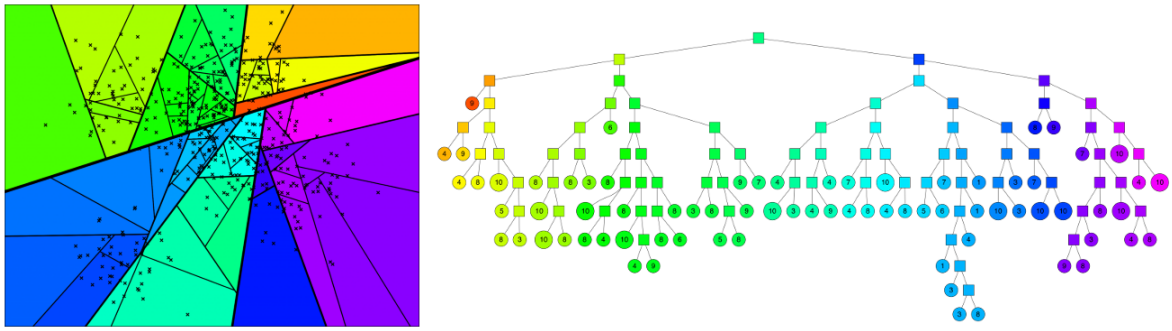
1. Select two data points arbitrarily. The hyperplane equidistant from the selected two points splits the space into two subspaces.

2. Save the random split as a node and create two branches. The points on the left sub-space go to the left branch, and the points in the right sub-space go to the right branch. depth $\leftarrow$ depth $+ 1$.

3. For each sub-space, repeat steps 1 and 2 recursively until there are at most K items left in each area created by the splitting hyperplanes.

An example can be found in Fig. 3.3. The squares are the intermediate nodes reserving the splitting hyperplanes and the circles denote the data points in subspaces with a number indicating the number of points.

Figure 3.3: Building a binary tree in ANNOY [7]. The left part gives an example of the space partition and the right part shows the corresponding tree structure.

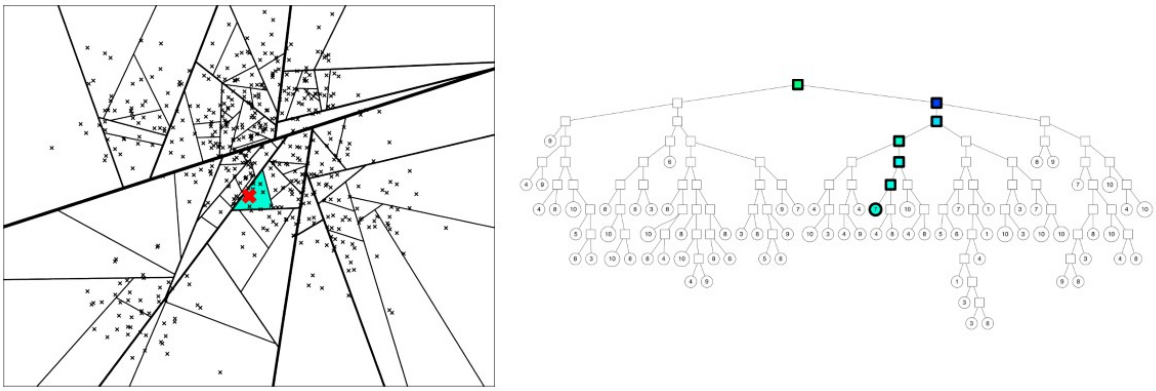


To search the nearest neighbour of a given query, we start with the root and traverse the tree following the same rule when we insert nodes: going to the left/right branch if the query is on the left/right of the splitting plane. We stop until we reach a leaf node, all the data points inside of which are regarded as nearest neighbours. An example is given in Fig. 3.4. The red cross is the query and the colored route shows the procedure of finding the shaded area where the query lies.

However, there are two more issues that need to be addressed. Firstly, as we can see in Fig. 3.4, some neighbours of the query that are even closer are not selected because they are divided into adjacent subspaces. Moreover, the number of data points in the shaded region is very likely to be different from what is required. The trick for these issues is called priority queue. Build a forest of trees, put the results of all trees into a list, and sort them based on the distance to the query. Also, we can pretentiously keep the wrong side of the tree if the points in that branch are not farther than a predetermined threshold.

In practice, bumping up the number of trees is generally a good idea if memory is not a concern. ANNOY is initially designed for vectors that are less than 100 dimensions, but empirically it works well even when the dimensionality is around 1000. The search complexity is close to $\mathcal{O}(\log N)$. However, no theoretical guarantee is given.

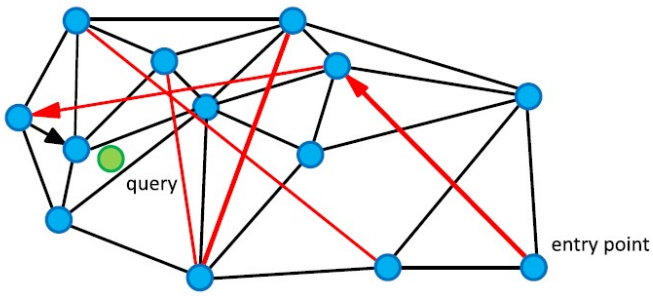
Figure 3.4: Searching with the binary tree in ANNOY [7]. The red cross denotes the query and the points in the green area are the nearest neighbours found by ANNOY. The route is colored.



3.5 Hierarchical Navigable Small World Graph (HNSW)

Before discussing the details of HNSW, it is necessary to first introduce its predecessor that is proposed by the same author: Navigable Small World (NSW) [58]. Recall that in Section 2.2.2 we mentioned that Delaunay Graph (DG) ensures greedy search always returns precise results. The biggest drawback is that the graph will be almost fully connected in high-dimensional space, offering little advantage over linear scan. NSW is an approximate version of GD with a relatively simple construction. The elements in the database are inserted sequentially: Given an element, find its nearest neighbours using greedy search and associate them with edges, by which the connectivity is automatically ensured. Long edges constructed in the early stage serve as 'highways', enabling us to move from one side of the graph to the other efficiently in the searching phase. As more and more data points are inserted, the edges will be shorter and shorter, which ensures search accuracy. An example of the NSW graph is shown in Fig. 3.5. The red

Figure 3.5: Searching with NSW graphs [58]. The nodes denote the data points. Red lines are constructed in the early stage, serving as 'highways', enabling fast moving from one side to the other. Black lines are constructed in the later stage, making the search more accurate. The path from the entry point to the nearest neighbour using greedy search is marked by arrows.

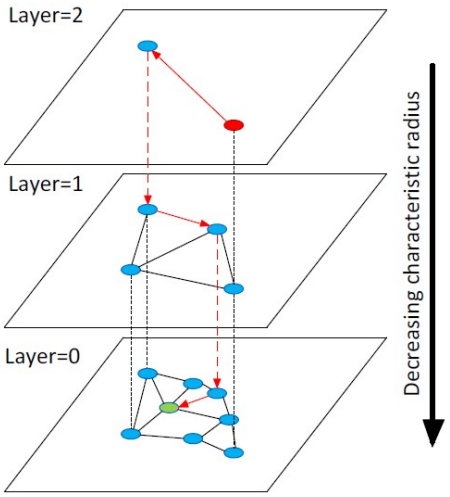


edges are the 'highways'. By following the directions indicated by arrows, we obtain the

route from the entry point to the nearest neighbour of the query using greedy search. Compared to GD, the average degree of the nodes scales logarithmically. i.e., $\mathcal{O}(\log N)$, which is far better than fully connected. The average number of visited nodes scales also logarithmically, and thus the search complexity is polylogarithmic, i.e., $\mathcal{O}((\log N)^2)$ [57]. Based on the construction procedure, we know that the construction complexity is $\mathcal{O}(N \cdot (\log N)^2)$.

Researchers did not stop here and found a variant of NSW, which is so-called HNSW [57], aiming to further reduce the search complexity to logarithmic. The idea is to introduce layers, separating the edges based on the length scale into different layers, as is shown in Fig. 3.6. There is a non-negative integer l indicating the maximum layer

Figure 3.6: The hierarchical structure [57] of HNSW. Data points become sparser and sparser from the bottom to the top. The scaling of the expected number of layers is logarithmic and there is a maximum number of connections for all layers. Searching is from the top to the bottom. For each layer, we use greedy search to find the nearest neighbour, which is the entry point for the next layer.



each data point belongs to. For example, if $l = 2$, the element can exist in layer $0, 1$, and 2. By setting an exponentially decaying probability of l, the scaling of the expected number of layers will be logarithmic. In layer 0, all data points will be included. In higher layers, the data points will be sparser and sparser, and the edges will serve as 'highways'.

Similar to NSW, we construct HNSW by sequentially inserting elements in the database. Given a new element, we generate an integer l and start from the top layer $L_{\max}$. For layer $i, i \in \{l+1, ..., L_{\max}\}$, we find the nearest neighbour in the current layer using greedy search, which will then be the entry point of layer $i+1$. For layer $i, i \in \{0, ..., l\}$, we find the K-nearest neighbour of the element and establish connections between the neighbours and the element. Suppose that the graph has been constructed, the search phase also starts from the top layer. Same as before, the nearest neighbour of layer i will be the entry point of layer $i+1$. We will maintain a dynamic list of found nearest neighbours, and the search will stop if the list is not updating.

Different from NSW, there is a limit on the maximum number of connections per

layer, and thus the search complexity can be $\mathcal{O}(\log N)$. Following the same rationale, we can see that the construction complexity can also be reduced from $\mathcal{O}(N \cdot (\log N)^2)$ to $\mathcal{O}(N \cdot \log N)$.

3.6 Hybrid Methods

As we mentioned before, compression-based methods simplify the calculation of distance between a query-candidate pair, and tree/graph-based methods reduce the number of candidates. To get the best of both worlds, we can consider combining compression-based methods and tree/graph-based methods together.

Generally speaking, PQ can be combined with any graph-based method if the graph is built based on Euclidean distances between nodes. If the D-dimensional database vectors $\mathbf{x}_n$ are compressed into M-dimensional codes $\mathbf{i}(\mathbf{x}_n)$ using PQ, the approximation of $\mathbf{x}_n$ can be reconstructed with (3.4), from which a graph can be constructed as usual. In the search phase, the approximation of distance between the query $\mathbf{q}$ and $\mathbf{x}_n$ can be calculated by table look-ups as in (3.6). During the whole process, we work with N M-dimensional codes and a codebook instead of N D-dimensional database vectors. However, it is not trivial to combine PQ with tree-based methods. The trees are generally constructed based on the space partition. But the points that are close in the original space are not necessarily close in the code space when using PQ. Therefore, the partition in the code space does not make much sense and the tree based on which will lead to poor performance.

For GH, however, it can be naturally combined with tree-based methods because the similarity in the original space will be preserved in the code space. So the partition in the code space still makes sense. Recall that tree-based methods suffer from the 'curse of dimensionality', which can also be alleviated if we use compact hash codes. It can be combined with graph-based methods as well, but in this case the graphs will be constructed based on the Hamming distance instead of the euclidean distance. A concern in GH is that the Hamming distance is much less expressive than the euclidean distance.

3.6.1 PQ+HNSW

The first combination we have tried is PQ+HNSW. The sub-codewords can be generated by either PQ or PQN. To construct the HNSW graph of the encoded database vectors, we take the following steps:

1. Following the procedure discussed in Section 3.1 (Eq. (3.3)), encode database vectors $\{\mathbf{x}_n\}_{n=1}^N, \mathbf{x}_n \in \mathbb{R}^D$ to PQ codes $\{\mathbf{i}_n\}_{n=1}^N, \mathbf{i}_n \in \{1, \cdots, K\}^M$.

2. Select unique PQ codes in $\{\mathbf{i}_n\}_{n=1}^N$ to form $\{\mathbf{i}_u\}_{u=1}^U$. Save the mapping from the current indices to the original indices as a dictionary.

3. Construct a HNSW graph of $\{\mathbf{i}_u\}_{u=1}^U$ as described in Section 3.5. For any two PQ codes $\mathbf{i}_a$ and $\mathbf{i}_b$, denote $\mathcal{M}$ as the set of m where $i_a^m \neq i_b^m, m \in \{1, \cdots, M\}$. The

distance between $\mathbf{i}_a$ and $\mathbf{i}_b$ is calculated as

$$d(\mathbf{i}_a, \mathbf{i}_b)^2 = \sum_{m \in \mathcal{M}} d(\mathbf{c}^m_{i^m_a}, \mathbf{c}^m_{i^m_b})^2. \tag{3.20}$$

Searching with the constructed HNSW graph of PQ codes is similar to searching with normal HNSW graph of original vectors, except that the distance calculations will be performed by table look-ups as in Eq. (3.6). Note that the obtained indices should be mapped back to the original indices using the dictionary saved during the construction. The final list may be longer than desired since there could be one to multiple mapping. We can simply discard the last several elements.

3.6.2 GH+ANNOY

We can also consider combining GH with ANNOY and the procedure is simpler than PQ+HNSW. First generate compact hash codes then organize them according to ANNOY (Section 3.4). The advantage is that ANNOY can now work with much shorter vectors, which not only boosts the efficiency, but also prevents memory overhead, especially when we build a forest of trees to improve the retrieval performance.

4 Experimental Results

In this chapter, we summarize the results of our experiments, which can be divided into two stages:

- **Toy examples for function verification using a simple prototype (Section 4.3).** Although extensive tests have been done in the original papers proposing PQ, HNSW and ANNOY, they are not in the context of image retrieval but more about purely solving the nearest neighbour search problem. The datasets consist of extracted feature vectors and the metric is recall. In PQN and GH (and also many research studies on deep hashing methods), however, the authors define and solve image retrieval problems and report mean average precision (which will be introduced in Section 4.1). Therefore, it is necessary to build a prototype and test various methods in a consistent scenario. After this round of testing, some unsuitable methods will be discarded.
- **Real-world examples with a more advanced pipeline and complex datasets (Section 4.4).** The methods selected in the previous phase will be integrated into the system together with an improved feature extractor, after which the pipeline will be able to tackle complex datasets. The results obtained in this part have more realistic implications.

Apart from giving numerical results, a web-based GUI is built to provide intuitive visualizations, which will be introduced in Section 4.4.3. But first, we review important metrics used throughout this chapter.

4.1 Metrics

4.1.1 Mean Average Precision (mAP)

mAP is the most frequently used criterion in image retrieval, but its definition can be difficult to understand if given directly. So let us first consider the more intuitive criteria: precision and recall:

$$\text{Precision} = \frac{|\{\text{relevant images}\} \cap \{\text{retrieved images}\}|}{|\{\text{retrieved images}\}|}, \tag{4.1}$$

$$\text{Recall} = \frac{|\{\text{relevant images}\} \cap \{\text{retrieved images}\}|}{|\{\text{relevant images}\}|}, \tag{4.2}$$

where $|\cdot|$ denotes the cardinality of a set. However, they have a drawback. For retrieval tasks, we would expect that the most relevant items should occur on the top of the results, i.e., the ranking is also important. But both precision and recall do not indicate

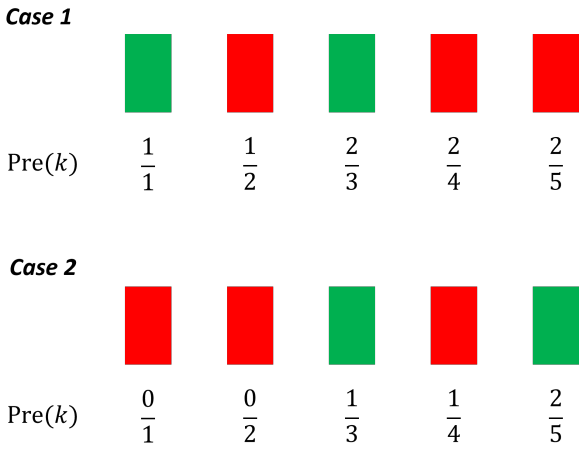
if the ranking is reasonable since various rankings lead to the same values as long as the number of retrieved relevant images is fixed according to the definitions.

A single value of precision (or recall) can not preserve ranking information. How about a list of precision values? A concept called precision-at-k Pre(k) is useful. Given a list of ranked images, we collect the first k results and calculate the precision-at-k as:

$$\text{Pre}(k) = \frac{|\{\text{relevant images}\} \cap \{\text{top-k results}\}|}{k}. \tag{4.3}$$

With a list of Pre(k), we can inverse the ranking information. An example is given in Fig. 4.1.

Figure 4.1: Two cases of five ranked images. Correct results and wrong results are marked as green and red, respectively. For each case, the corresponding Pre(1), ..., Pre(5) are listed below. Different rankings lead to different lists of Pre(k) values. On the other hand, the locations of correct results can be identified from the changes of the numerators. Therefore, we can inverse the ranking information.



Similarly, recall-at-k Rec(k) can also be defined. We can draw a precision-recall curve based on the values of Pre(k) and Rec(k). Building upon these definitions, the average precision AP is defined as the coverage area under the precision-recall curve, which can be approximated as:

$$\text{AP} = \frac{1}{R}\sum_{k=1}^{N} \text{Pre}(k)\text{rel}(k), \tag{4.4}$$

where R is the number of relevant images (ground truths), N is the size of the database, and rel(k) is an indicator that equals 1 if k-th retrieved image is relevant and 0 otherwise. Here the summation is from $k = 1$ to $k = N$, meaning that all the images in the database are ranked and took into consideration. This is not necessary for real applications, where we usually retrieve only a small part of the database images. Assume the number of images to be retrieved is K, with $K \ll N$. We also have a definition of

average-precision-at-K AP@K:

$$\text{AP@K} = \frac{1}{\min\{K, R\}} \sum_{k=1}^{\min\{K,R\}} \text{Pre}(k)\text{rel}(k). \tag{4.5}$$

The denominator now is $\min\{K, R\}$, making sure that in the best case AP@$K = 1$.

During the tests there will not be only one query but multiple query images, which is why we need to define mean average precision mAP or mAP@K:

$$\text{mAP} = \frac{1}{Q} \sum_{q=1}^{Q} \text{AP}(q), \tag{4.6}$$

$$\text{mAP@K} = \frac{1}{Q} \sum_{q=1}^{Q} \text{AP@K}(q), \tag{4.7}$$

where Q is the number of query images.

4.1.2 Relative Mean Average Precision (rmAP)

It is worth noting that, in this thesis, the absolute value of mAP is not very important since it is determined by both the quality of features and the error introduced by ANN algorithms. Instead, we care more about how much the mAP degrades after applying ANN algorithms compared to linear scan. To show this clearly, we introduce relative mAP (rmAP), which is the original mAP minus the mAP of linear scan. E.g., if the mAP of linear scan is 63.2% and the mAP of PQ is 60.4%, then the rmAP of PQ is -2.8%.

4.1.3 Matching Time and Speedup

Another important aspect is of course the matching efficiency, which can be shown most intuitively in terms of the matching time. However, due to differences in hardware, the runtime may vary significantly on different devices. So in this thesis, we also report the speedup over linear scan, a number followed by $\times$. For instance, if the matching time of linear scan and PQ is 1 ms and 0.1 ms, respectively, then the speedup is $10\times$. Throughout this thesis, the speedup is calculated based on the time of retrieving top 100 results.

4.2 Implementation

For methods like PQ and HNSW, there are existing functions in the widely-used nearest neighbour search library Faiss [39], which is written in C++ with Python bindings. But implementing the hybrid method PQ+HNSW can not be done without modifying the source codes, which is challenging considering the original programming language is C++. Therefore, we modify the codes of PQ and HNSW from open source projects on GitHub written in Python without special optimization, and we build the hybrid method PQ+HNSW upon them [59, 48].

ANNOY is proposed by a practitioner, so we stick with the official library that is also written in C++ with Python bindings [6]. It applies single instruction multiple data (SIMD) instruction, so the comparison between ANNOY and other methods in this thesis is not very fair.

We implement PQN based on the description in the paper and we modify GH from the official codes provided by the authors [82]. The combination of GH+ANNOY is relatively simple and does not require changing the source code of ANNOY.

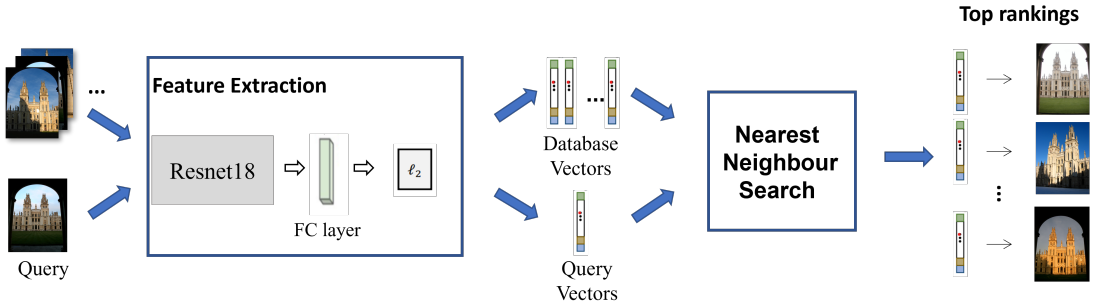
The repository for this thesis project is available on https://github.com/YYao-42/ImgSearch.

4.3 Function Verification on Prototype

4.3.1 Prototype

The prototype is a minimum implementation of the pipeline shown in Fig. 1.1. The reranking module is left out for simplicity. The feature extractor is a Resnet18 followed by a fully connected (FC) layer with 256 neurons and an ℓ_2 normalization layer. The simplified pipeline of the prototype is shown in Fig. 4.2.

Figure 4.2: The pipeline of the prototype. Compared to Fig. 1.1, the reranking model is omitted. And the feature extraction module is specified as a Resnet18 followed by a FC layer and an ℓ_2 normalization.

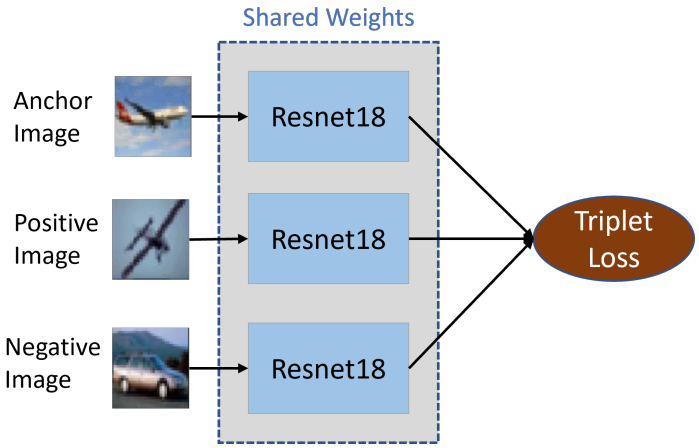


The model can be trained end-to-end with a triplet loss, which is illustrated in Fig. 4.3. To simplify the drawing, the FC and ℓ_2 normalization are absorbed into the Resnet18 block. The input is a tuple of images, containing an anchor image, a positive image that is similar to the anchor, and a negative image that is not similar to the anchor. The features of anchor, positive and negative image are denoted as $\mathbf{x}_a, \mathbf{x}_p, \mathbf{x}_n$, respectively. We choose the commonly used ℓ_2 distance as the distance measurement and the triplet loss is defined as

$$\mathcal{L}_{\text{Triplet}-\ell_2}(\mathbf{x}_a, \mathbf{x}_p, \mathbf{x}_n) = \max\{0, \text{Margin}_{\ell_2} + \|\mathbf{x}_a - \mathbf{x}_p\|_2^2 - \|\mathbf{x}_a - \mathbf{x}_n\|_2^2\}, \tag{4.8}$$

where Margin_{ℓ_2} is a hyperparameter. To minimize the triplet loss, the ideal case is that the distance between $\mathbf{x}_n$ and $\mathbf{x}_a$ is farther than the distance between $\mathbf{x}_p$ and $\mathbf{x}_a$ by at least a certain margin. Therefore, after training, we can expect that the distance between feature vectors will be a good indication of the degree of similarity of the original image.

Figure 4.3: Training the prototype end-to-end with a triplet loss. The FC and ℓ_2 normalization are absorbed into the Resnet18 block for simplicity. The features of the anchor, positive and negative images are extracted respectively with the same Resnet18, from which the triplet loss is calculated.



If the nearest neighbour search module is PQN or GH, then we need to jointly consider the training of the features and the codes. For PQN, the features of positive and negative images $\mathbf{x}_p$ and $\mathbf{x}_n$ are quantized as $\tilde{\mathbf{x}}_p$ and $\tilde{\mathbf{x}}_n$. The triplet loss is calculated based on the quantized features:

$$\mathcal{L}_{\text{Triplet}-\ell_2}(\mathbf{x}_a, \tilde{\mathbf{x}}_p, \tilde{\mathbf{x}}_n) = \max\{0, \text{Margin}_{\ell_2} + \|\mathbf{x}_a - \tilde{\mathbf{x}}_p\|_2^2 - \|\mathbf{x}_a - \tilde{\mathbf{x}}_n\|_2^2\}, \tag{4.9}$$

and an illustration can be found in Fig. 4.4. The feature of the anchor image $\mathbf{x}_a$ is not quantized during the training since we apply asymmetric distance calculation as in (3.6). All the database vectors will be quantized, but the query will not. We would like to maximize (minimize) the distance between quantized features of dissimilar (similar) images and the unquantized feature of the query image.

For GH, the training scheme becomes more complicated, as is shown in Fig. 4.5. Different from PQ, database vectors and the query vector will all be encoded as hash codes. Therefore a GH layer is inserted after feature extractor for anchor, positive and negative images to generate hash codes $\mathbf{h}_a, \mathbf{h}_p, \mathbf{h}_n$, respectively. The distance measurement in triplet loss now becomes Hamming distance. Since in GH the element of a hash code is +1 or -1, the Hamming distance given two hash codes $\mathbf{h}_i$ and $\mathbf{h}_j$ is

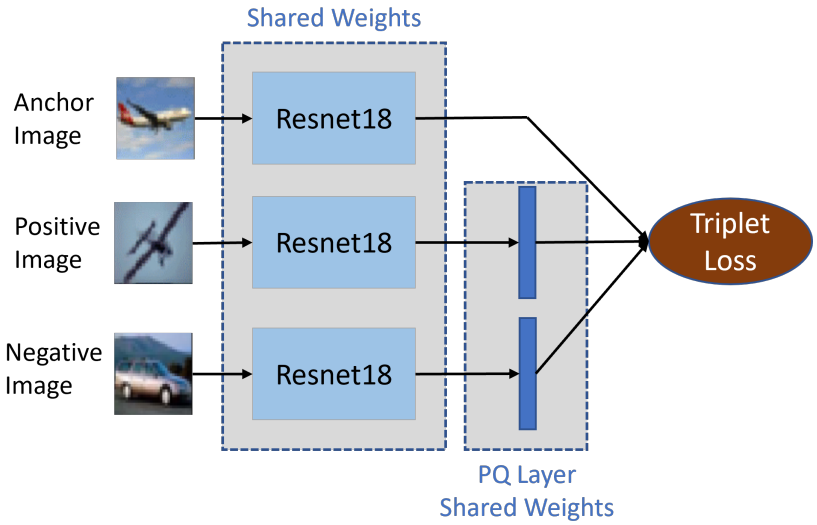
$$\mathcal{D}_h(\mathbf{h}_i, \mathbf{h}_j) = (L - \mathbf{h}_i^{\mathrm{T}}\mathbf{h}_j)/2, \tag{4.10}$$

where L is the length of the codes. Therefore the triplet loss is

$$\mathcal{L}_{\text{Triplet-H}}(\mathbf{h}_a, \mathbf{h}_p, \mathbf{h}_n) = \max\{0, \text{Margin}_{\text{H}} + (L - \mathbf{h}_a^{\mathrm{T}}\mathbf{h}_p)/2 - (L - \mathbf{h}_a^{\mathrm{T}}\mathbf{h}_n)/2\}. \tag{4.11}$$

Theoretically, we can work with $\mathcal{L}_{\text{Triplet-H}}$ only to train the model, but the performance is not so good in practice. The main reason is that hash codes are not very expressive and much information is lost after passing through the GH layer, making the training very difficult. Therefore, the original features are also output to calculate $\mathcal{L}_{\text{Triplet}-\ell_2}$

Figure 4.4: Jointly training the feature extractor and the PQ layer. Compared to Fig. 4.3, the features of positive and negative images are quantized and then forwarded to the loss function.



using (4.8). The overall loss function is the weighted sum of $\mathcal{L}_{\text{Triplet}-\ell_2}$ and $\mathcal{L}_{\text{Triplet-H}}$:

$$\mathcal{L} = \mathcal{L}_{\text{Triplet}-\ell_2}(\mathbf{x}_a, \mathbf{x}_p, \mathbf{x}_n) + \lambda \mathcal{L}_{\text{Triplet-H}}(\mathbf{h}_a, \mathbf{h}_p, \mathbf{h}_n), \tag{4.12}$$

where λ is a hyperparameter.

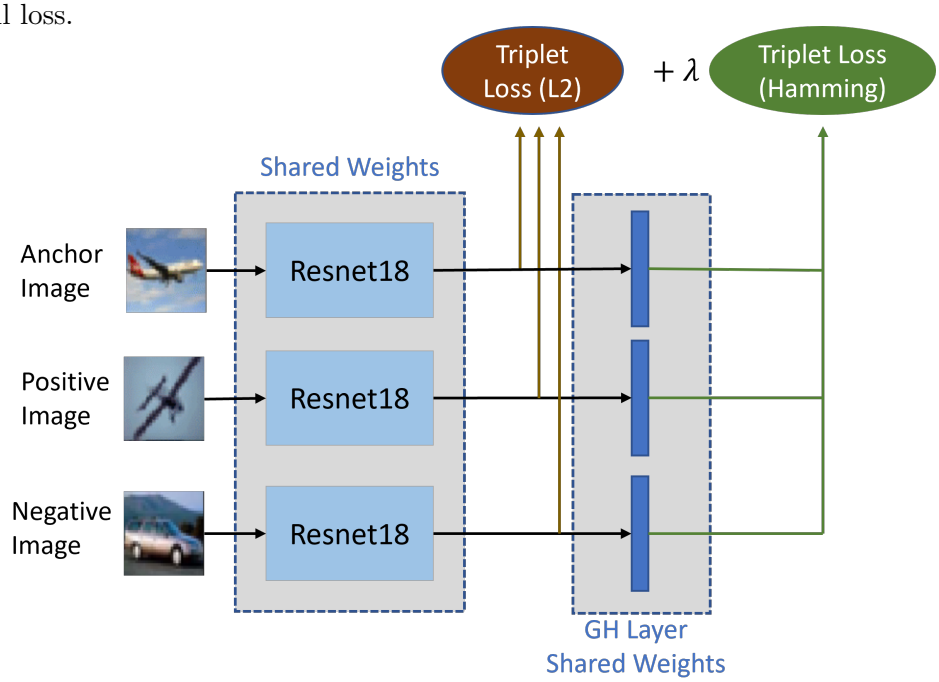
4.3.2 Datasets

The datasets used in this part are not specially designed for image retrieval problems but general classification problems. The images from the same class are considered similar and otherwise dissimilar. This is slightly in conflict with the scope of 'similar' discussed in Chapter 1, where we claimed that similar images should depict the same object. But since similarity is relative, it is reasonable to regard items of the same class as similar when there is no same item. Moreover, such datasets are frequently used for tests in deep hashing methods, so we use them as well in this simple experiment on the prototype.

CIFAR-10 CIFAR-10 is used in almost every paper of deep hashing methods. It contains 50000 images for training and 10000 images for testing. The size of images is 32×32 and the number of channels is 3. For both the training set and test set, there are 10 classes with the same number of images. In our setting, the training set is the database and the images in the test set are query images.

CIFAR-100 CIFAR-100 is very similar to CIFAR-10, except that there are 100 classes instead of 10. The total number of images is still 60000. It is rarely used in papers but we believe doing tests with CIFAR-100 can reveal some hidden issues and provide more

Figure 4.5: Jointly training the feature extractor and the GH layer. The outputs are features and hash codes, from which triplet losses measured by ℓ_2 and Hamming distance can be calculated. A hyperparameter λ controls the relative contribution of two losses to formulate the overall loss.



insights. For instance, the feature vectors in CIFAR-10 can be nicely clustered into 10 groups, hence PQ is inherently suitable. In real applications, however, the database vectors are much more scattered. How would PQ perform then? With CIFAR-100 we can mimic this situation since there are 100 classes and the feature vectors will be more scattered.

4.3.3 Results

4.3.3.1 CIFAR-10

The rmAP and speedup of selected ANN methods are listed in Table 4.1. The benchmark is linear scan, of which the mAP@100 = 74.00%. To find top matches in a database with 50000 256-dimensional vectors using linear scan, the time cost is around 42 ms (varies from machine to machine). Indeed, accelerating matching in CIFAR-10 may seem unnecessary, but it still gives us an idea of what kind of improvement ANN methods can bring.

The results in the table basically show that ANN methods trade search efficiency with minor performance loss. But there are some interesting numbers in the table that need further explanation:

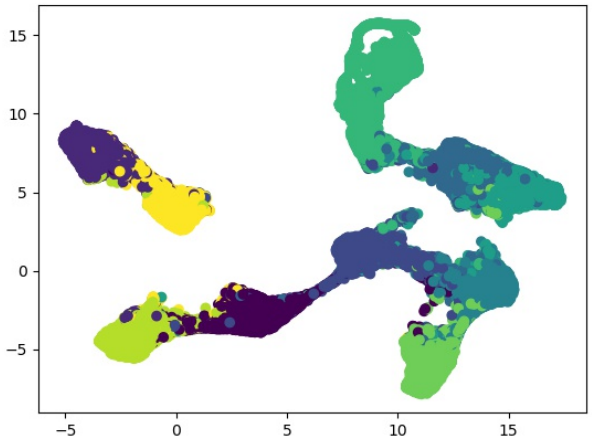
- **Why the mAP can even be higher in some cases?** The short answer is that Euclidean distance may not be the best metric to find neighbours. Using ANN methods may give us results that are not nearest in ℓ_2 space, but they can be

Metric	Compression-based			Tree/Graph-based		Hybrid		
	PQ	PQN	GH	ANNOY	HNSW	PQ+ HNSW	PQN+ HNSW	GH+ ANNOY
rmAP(%)	−1.03	+0.86	−1.15	+0.32	+0.74	+0.54	+2.80	−2.12
Speedup	15.34×	14.97×	11.43×	201.43×	12.62×	22.26×	23.54×	631.34×

Table 4.1: rmAP(%) and speedup of ANN algorithms in CIFAR-10. For compression-based methods, the feature vectors are compressed to 32-bit codes.

desired neighbours as well. In CIFAR-10, features are not randomly and uniformly scattered in space. We can apply a dimension reduction tool UMAP [60] to visualize the high-dimensional feature vectors in 2D space (Fig. 4.6). At the border of different classes, it is obvious that ℓ_2 distance is not an appropriate criterion to determine whether two items are similar. We may also need to consider topology properties. In fact, there are methods that take topology into consideration and boost the performance of image retrieval, e.g., diffusion [94], but this is beyond the scope of this thesis.

Figure 4.6: Visualizing 50000 256-dimensional feature vectors in 2D space using UMAP. Different colors indicate features of images from different classes. The distribution is not uniform but has a certain structure.



- **Why is ANNOY so good?** Empirically ANNOY can achieve logarithmic search complexity if vectors are relatively low dimensional (less than 100D). From the literature, HNSW also has logarithmic search complexity, so their performance should be close. However, in the table ANNOY is almost 10 times faster than HNSW. Recall that in Section 4.2, we mentioned that the comparison of ANNOY and other methods is not fair since ANNOY is written in C++ and uses SIMD instructions. Therefore, the performance gain does not come entirely from the algorithm itself.

4.3.3.2 CIFAR-100

CIFAR-100 is more challenging than CIFAR-10 since it has 100 classes. To be more expressive, the length of codes is increased to 128 bits to prevent significant performance loss. Naturally, we can expect that the matching efficiency will drop since the codes are less compact and the computation cost is higher. For tree/graph-based methods, the speedup should be comparable to the case in CIFAR-10 since the size of the database and the dimension of vectors are unchanged. The results can be found in Table 4.2.

	Compression-based			**Tree/Graph-based**		**Hybrid**		
Metric	PQ	PQN	GH	ANNOY	HNSW	PQ+ HNSW	PQN+ HNSW	GH+ ANNOY
rmAP(%)	−1.50	**−13.50**	**−10.17**	+0.97	+0.92	+0.41	**−13.27**	**−9.88**
Speedup	5.69×	6.05×	3.84×	212.68×	11.91×	15.75×	16.01×	334.21×

Table 4.2: rmAP(%) and Speedup of ANN algorithms in CIFAR-100. For compression-based methods, the feature vectors are compressed to 128-bit codes.

The biggest difference in CIFAR-100 compared to the case in CIFAR-10 is that we observe **significant performance loss in deep hashing methods PQN and GH**. The main challenge for PQN and GH is that, as the code length is longer, there are more parameters to train. A model with more parameters generally requires more training data, but the scale of the training set is the same in CIFAR-10 and CIFAR-100. During the training, we consistently met the overfitting problem, albeit multiple groups of parameters have been tried. People can argue that with more data available the methods may still work, but data is expensive in real-world applications.

4.3.4 Discussion

In this section, we decide the list of methods that will be integrated into the system based on the results of the experiments on the prototype.

In short, we discard PQN, GH and the hybrid methods that are built upon them. Although recent research in compression-based methods heavily focuses on deep hashing methods, they are difficult to train and may lead to significant performance loss in more complex datasets. Besides, in the final system, we would prefer that the model can generalize well across different datasets such that retraining is unnecessary. But with PQN and GH, we always need to retain the model if the length of codes is changed, which frequently happens.

ANNOY is kept due to its particularly high efficiency. As a method that is widely used by practitioners, it ensures our system has attractive matching performance. During the preliminary tests, the proposed hybrid method outperforms PQ or HNSW alone in search efficiency. We would like to investigate how it performs in real applications. For comparison purposes, PQ and HNSW will also be incorporated into the system.

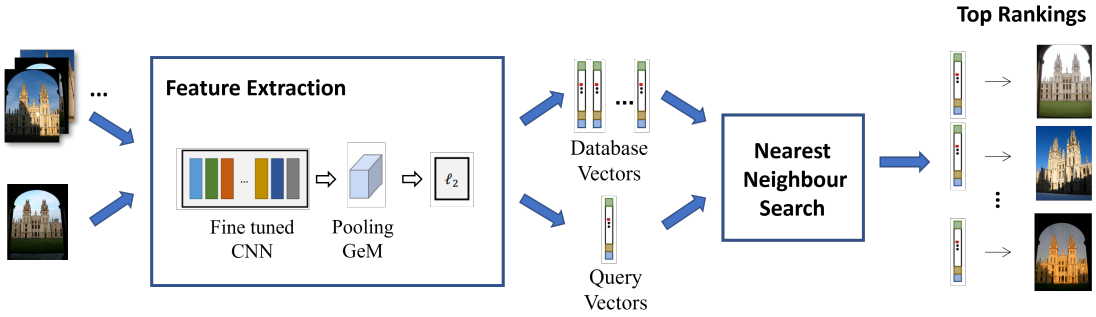
4.4 System Performance

4.4.1 System

The prototype described in Section 4.3.1 can handle relatively simple datasets, but does not perform well with more challenging datasets since the feature extractor is too simple and does not extract high-quality features. Apart from dragging mAP down, a more serious problem is that mAP may not be a good indication of the performance of ANN methods anymore. If features of different objects mix together, finding neighbours of the query vector more accurately will not necessarily lead to a higher mAP. In extreme cases, it is possible that the mAP does not change much even ANN algorithms perform badly, or drop a lot when they perform well.

Therefore, introducing a more delicate feature extractor is necessary. In this thesis, we directly apply the model proposed in [71] and its corresponding pretrained parameters provided by the authors. Instead of using the output of a FC layer as the global feature, a pooling method called Gem pooling is proposed to embed local feature matrices and generate 2048D feature vectors. We display an illustration of the pipeline in Fig. 4.7.

Figure 4.7: The pipeline of the search engine. As in the prototype, the reranking model is not considered since it is a post-processing procedure of nearest neighbour search. The CNN backbone of the feature extractor is ResNet101 and Gem Pooling is applied to embed local features [71].



4.4.2 Datasets

Using datasets that are initially designed for classification can be reasonable if we relax the condition of 'similar', as objects depicted by two images belong to the same class. But a more common and challenging use case would be the search for an object in a database of images, e.g., a specific type of car. For instance, a historian may want to learn more about a particular building that is depicted in a painting. To train and test a model that is able to achieve such needs, people collect images from different viewpoints and under different light conditions of a set of objects to create a dataset. A ground file is usually provided, indicating the true matches of each query. In this research, we test our model on various commonly used datasets in image retrieval for benchmarking. Since we attach extra attention to the application in the field of historical research,

we have also created a dataset containing images of ancient relics, manuscripts, and paintings.

Retrieval-SfM-120k The extractor we adapted from [71] is trained on retrieval-SfM-120k introduced in [70]. The dataset contains 551 reconstructed 3D models (133,659 images), 10% of which are training queries. The images are mainly about famous architecture all over the world. Since we use parameters provided by the authors directly, we do not need to download and process this dataset.

Oxford5k It contains 5062 images of 11 Oxford landmarks. For each landmark, 5 possible queries are given, which makes 55 queries in total. The label of an image can be 'Good', 'OK', 'Bad', or 'Junk', where the first two indicate that at least 25% of the object is clearly visible, and the last two mean that less than 25% of the object is visible or the image is highly distorted.

Paris6k Paris6k contains 6412 images of 12 Paris landmarks. Still, each landmark has 5 queries so there are 60 queries we can test. The labels have the same meaning as in Oxford5k. Oxford5k and Paris6k were created in 2008 and have been used for a long time to test the performance of image retrieval systems. Despite their small scale, almost all the research in image retrieval reports the results on Oxford5k and Paris6k.

Revisited Oxford5k (ROxford5k) In [69], researchers argued that there are some annotation errors in the standard Oxford5k and Paris6k. They revisited annotations as 'Easy', 'Hard', 'Unclear', and 'Negative', introduced 15 new queries, and defined three different evaluation protocols 'Easy', 'Medium' and 'Hard' based on the scope of images that should be retrieved.

Revisited Paris6k (RParis6k) Similar operations have been done in Paris6k. The label information of a dataset is indeed extremely important. If there are errors in the annotations, mAP will be meaningless and be unable to indicate the performance of algorithms. From this perspective, revisited datasets are of higher quality and are gradually being used in more and more studies.

100k Distractors Apart from annotation errors, the limited number of images in Oxford5k and Paris6k still is a problem. As discussed before, retrieval efficiency is also a critical factor for an image retrieval system, but results on (R)Oxford5k and (R)Paris6k tell little about efficiency differences of each method: even linear scan can perform well. Therefore, 100k distractors downloaded from Flickr using 145 popular tags are added into (R)Oxford5k and (R)Paris6k to generate (R)Oxford105k and (R)Paris106k. Except for expanding datasets, another function of those 100k distractors is to enrich the variety of the datasets and make feature vectors more scattered.

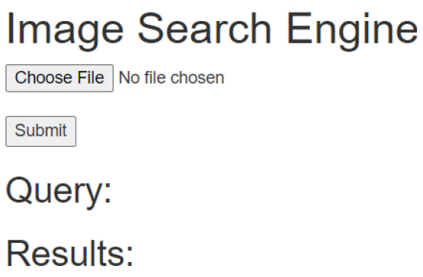
Google Landmarks (GLM) As a more challenging assessment, we also test on GLM version 2 introduced in [92], which is a million-scale dataset. It has 762k index images of 101k landmarks, which serve as our database, and 118k query images, 1% of which provide the IDs of matching images. GLM is by now one of the most realistic datasets, containing a wide range of landmarks and having large intra-class variability. From the official ground truth file, we can see that among 762k database images, there are usually less than 10 matches. Considering all the factors above, GLM is a very challenging dataset and even the most advanced algorithms can only achieve a mAP around 20%.

Custom For most datasets in image retrieval, the images are about architectures or landmarks. In historical research, however, the scope of the query can be much larger. Therefore, we create a custom dataset that contains 30 query images and 1236 database images including landmarks, manuscripts, sculptures, paintings, etc. As in (R)Oxford5k and (R)Paris6k, we add 100k distractors to expand the custom dataset.

4.4.3 Graphical User Interface (GUI)

The final objective is to build a working image search engine, which means that only reporting numbers is not enough. There should be an easy-to-use user interface that accepts query images and displays retrieved images. Considering this, a simple web application is built using Flask, a micro web framework written in Python. The initial interface before inputting anything is shown in Fig. 4.8. There are buttons to choose and submit query images. When the searching is completed, the query image and the retrieved images will be displayed under 'Query' and 'Results', respectively. An example is shown in Fig. 4.9.

Figure 4.8: The graphical user interface of the image search engine. Click 'Choose File' to select intended query images, then click 'Submit' to upload. The uploaded query image and its corresponding matching images will be displayed after the retrieval procedure is finished.

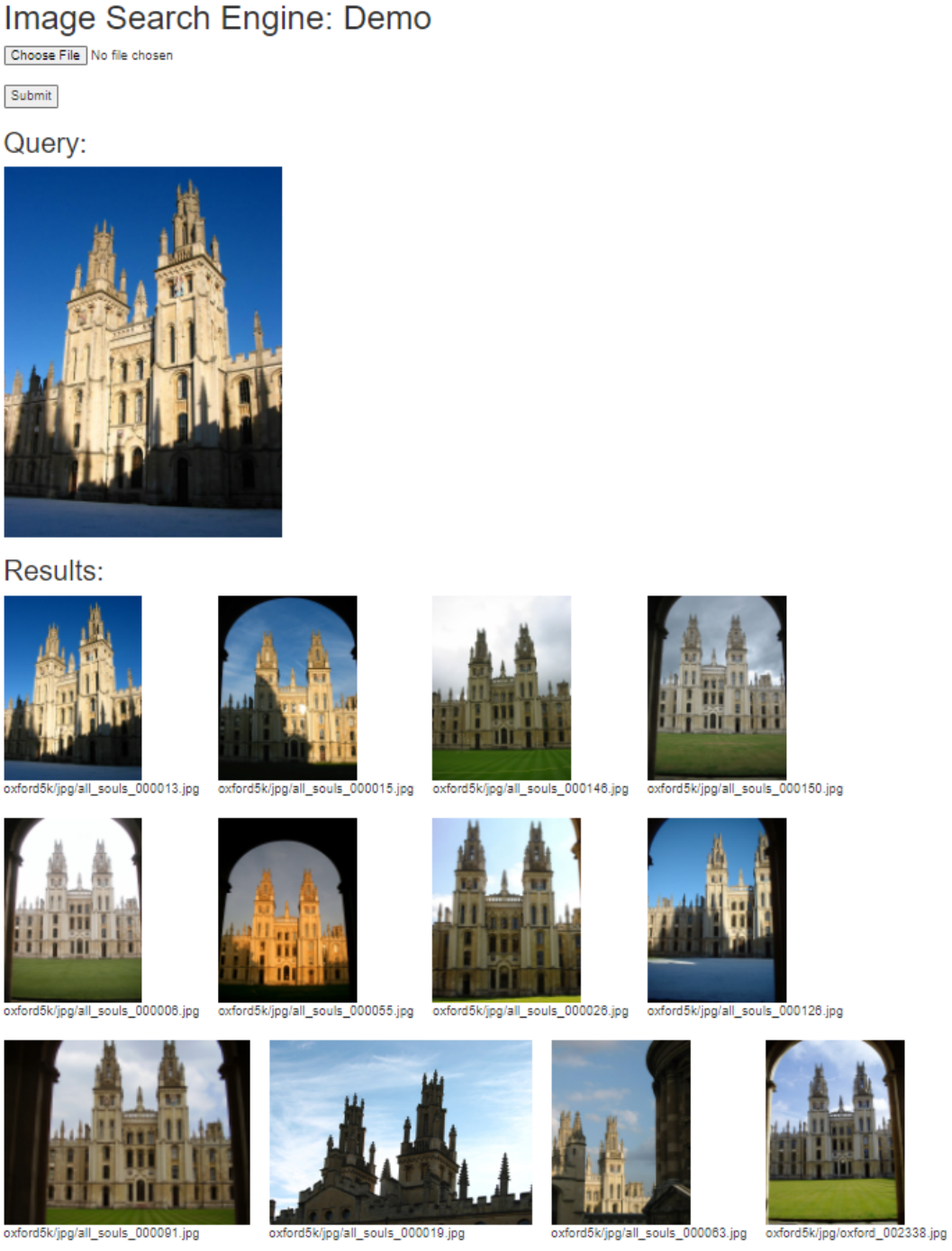


4.4.4 Results

4.4.4.1 (R)Oxford and (R)Paris

We summarize the results of rmAP(%) and speedup in (R)Oxford5k, (R)Paris6k, (R)Oxford105k and (R)Paris106k in Table 4.3. Revised datasets, as mentioned before, have three evaluation protocols: 'Easy', 'Medium' and 'Hard', denoted by 'E',

Figure 4.9: An example of the interface is when a search is finished. The relevant paths of matches are shown below the images. Since the naming of images of each dataset follows certain rules, we can roughly know the performance of retrieval.



'M', and 'H', respectively. Since in revised datasets, only the labels have been changed and the database images are the same, the matching time of original datasets and revised datasets should be similar (if not the same). Therefore the speedups of original

and revised datasets are regarded as the same and calculated using the average of their matching time.

Metric	Dataset	PQ	ANNOY	HNSW	PQ+ HNSW
rmAP(%)	Oxford5k	−2.29	0.00	−0.28	−2.28
	Paris6k	+1.32	+0.02	−0.15	+1.30
	Oxford105k	**−22.32**	0.00	−2.12	**−22.17**
	Paris106k	+0.02	+0.12	−0.14	+0.07
	ROxford5k	E:−1.01 M:−7.34 H:−7.43	E:0.00 M:0.00 H:0.00	E:−0.53 M:−0.35 H:−0.14	E:−1.00 M:−7.43 H:−7.69
	RParis6k	E:−0.36 M:+2.40 H:+5.85	E:0.00 M:0.00 H:0.00	E:−0.16 M:−0.29 H:−0.28	E:−0.57 M:+2.29 H:+5.79
	ROxford105k	E:−**25.88** M:−**18.99** H:−**13.41**	E:0.00 M:0.00 H:0.00	E:−0.04 M:+2.06 H:+0.06	E:−**25.82** M:−**18.98** H:−**13.45**
	RParis106k	E:−5.48 M:−0.45 H:+2.52	E:0.00 M:0.00 H:0.00	E:−0.36 M:−0.13 H:+0.09	E:−5.53 M:−0.69 H:+2.54
Speedup	(R)Oxford5k	9.59×	20.90×	3.92×	4.43×
	(R)Paris6k	9.80×	16.82×	4.14×	5.98×
	(R)Oxford105k	14.69×	250.04×	32.44×	64.40×
	(R)Paris106k	13.88×	222.99×	33.98×	65.69×

Table 4.3: rmAP(%) and speedup of ANN algorithms in (R)Oxford and (R)Paris. 'E', 'M', and 'H' stand for 'Easy', 'Medium', and 'Hard' mode in revisited datasets, respectively. Since revised datasets and original datasets have the same amount of database images, their speedup should be the same. For PQ, the length of codes is 128 bits. Cases with significant (more than 10%) performance loss are bolded.

From Table 4.3, several facts can be observed:

- PQ introduces significant performance loss in (R)Oxford105k, while still performing well in (R)Paris106k. This may be because features of each landmark aggregate better in Paris5k, leading to less quantization error with the same number of bits. If more bits can be assigned, we should be able to see performance improvements in Oxford datasets, which is verified in Table 4.4. However, although improved, the performance is still not satisfactory. Moreover, continuing to increase the number of bits will not make it any better.
- Tree/graph-based methods have very comparable performance to linear scan.

Metric	Method	Oxford5k	Oxford105k	ROxford5k	ROxford105k
rmAP(%)	PQ(128 bits)	−2.29	**−22.32**	E:−1.01 M:−7.34 H:−7.43	E:−**25.88** M:−**18.99** H:−**13.41**
	PQ(192 bits)	+0.46	**−15.16**	E:+0.64 M:+0.64 H:+0.90	E:−**20.06** M:−**15.25** H:−**13.40**

Table 4.4: rmAP(%) of PQ with 128-bit codes and 192-bit codes in Oxford datasets. Performance of the latter is consistently better, but still not satisfactory.

They are able to retrieve neighbours in the context of ℓ_2 distance efficiently and accurately. But the speedup of them in small-scale datasets is not so significant as in large-scale datasets.

- The performance of PQ+HNSW highly depends on the performance of PQ. Its speed advantage over PQ or HNSW alone becomes significant only in large-scale datasets.

4.4.4.2 GLM

GLM is a million-scale dataset. If no ANN algorithm is applied, i.e., if we search by linear scan, the waiting time is about 2 seconds. The ideal situation is to reduce it to several milliseconds, allowing more budget for feature extraction and reranking. From Table 4.5, we can see that HNSW and ANNOY can help us fulfil this requirement without much performance loss.

Metric	PQ	ANNOY	HNSW	PQ+ HNSW
rmAP@100(%)	−3.66	−1.07	−0.08	−3.69
Speedup	12.58×	313.50×	110.17×	35.45×

Table 4.5: rmAP@100(%) and speedup of ANN algorithms in Google Landmarks. For PQ, the 2048D feature vectors are divided into 16 sub-vectors, each of which has 2^{13} sub-codewords. The mAP@100 of linear scan is 10.70, and the matching time is around 2 seconds. With HNSW or ANNOY, the search can be finished in several milliseconds with a small mAP loss.

An unexpected fact is that PQ+HNSW is slower than HNSW, which is contrary to previous results. This is the cost of too many sub-codewords. To make the mAP of PQ acceptable, we assign 2^{13} sub-codewords to each sub-vector. Normally it is a much smaller number, e.g., 2^8. Recall that for PQ-based methods, we need to construct a distance table during each query. It turns out that the time of constructing the distance table is already longer than the total matching time of HNSW.

Despite the longer matching time and inferior retrieval performance, PQ+HNSW is still a good choice when memory is a concern. ANNOY is particularly unsuitable for

this situation since a forest of trees needs to be built to ensure high mAP. For example, in GLM, we construct 100 trees to reach this level of rmAP.

4.4.4.3 Custom Dataset

The results in Table 4.6 are obtained with 100k distractors included, otherwise it is unnecessary to use ANN methods since there are only 1236 images in the custom dataset and the values of speedup would be unrepresentative. To keep the performance loss within reasonable limits, the feature vectors are compressed to 192-bit codes in PQ-based methods, and the number of trees in ANNOY is 100.

Metric	PQ	ANNOY	HNSW	PQ + HNSW
rmAP@100(%)	−3.98	−3.66	+0.23	−3.97
Speedup	11.60×	47.24×	13.15×	19.71×

Table 4.6: rmAP@100(%) and speedup of ANN algorithms in the custom dataset. For PQ, the feature vectors are compressed into 192-bit codes. The mAP of linear scan is 77.11% and the matching time is around 260 milliseconds.

Increasing the number of trees in ANNOY leads to better performance but slower searching speed. Recall that in (R)Oxford105k and (R)Oxford106k, the speedup of ANNOY is greater than 200×. Nevertheless, ANNOY still leads to a performance loss of around 3.66%. In comparison, HNSW is also efficient and has no performance loss.

For this dataset, apart from reporting quantitative results, it is also illustrative to show some visualizations since the images in this dataset are more relevant to our application. They can be found in Appendix A.

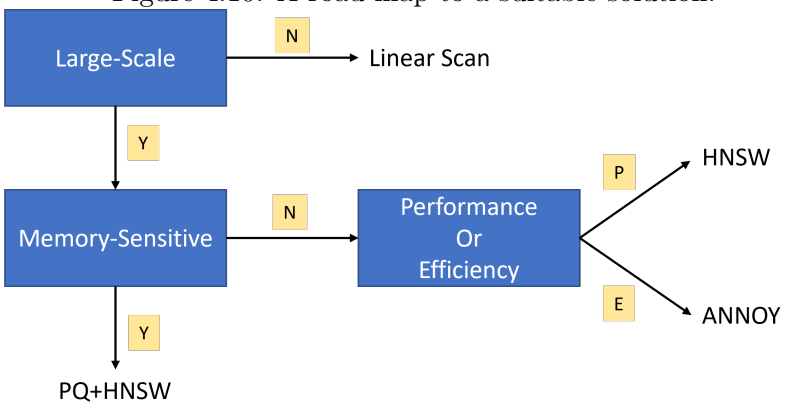
4.4.5 Discussion

The experimental results above reveal that no method is the best for all datasets. However, we can wisely select suitable solutions for specific use cases:

- For small-scale datasets (which have less than 10k images), linear scan is sufficient and super easy to apply.
- If memory is not a concern, use tree/graph-based methods like ANNOY and HNSW.
  - For higher retrieval efficiency, use ANNOY. (Theoretically their matching time should be comparable, but ANNOY is written in C++ with other optimizations.)
  - For better retrieval performance in large-scale datasets, use HNSW.
- If memory is limited, try the hybrid method PQ+HNSW.

We can make it clearer by drawing a decision tree, as is shown in Fig. 4.10.

Figure 4.10: A road map to a suitable solution.



5 Concluding Remarks

5.1 Conclusion

This work proposed a pipeline for real-time image retrieval in million-scale datasets, with a focus on the nearest-neighbour search module that is essential for efficient retrieval. Existing ANN methods from two categories, compression-based and tree/graph-based, were explored and implemented, including the widely used methods PQ, ANNOY and HNSW, and the more recent methods PQN and GH. In addition, we proposed a set of hybrid methods to get the best of both worlds, compressing high dimensional feature vectors into compact codes and organizing them as a tree or a graph to enable non-exhaustive search. The experimental setting of different methods in the original papers was inconsistent. Therefore, a prototype with a simple feature extractor was built to conduct comparisons between ANN methods under the same conditions. Despite the good results obtained in the papers, our tests on the prototype revealed the impracticality of deep hashing methods (PQN and GH) on more challenging datasets. Eventually, we narrowed down the scope of candidate methods to the following methods: PQ, ANNOY, HNSW, and PQ+HNSW.

To deal with real-world problems, a more advanced feature extractor was incorporated into the system and extensive tests on various datasets were done using ANN methods listed above. The results proved that the matching time of a million-scale dataset can be reduced from several seconds using brute force linear scan to several milliseconds using ANN methods. Meanwhile, the performance loss can be limited to an acceptable range. The results also showed that no method is the best under all circumstances. A roadmap for finding suitable solutions was created, helping practitioners to make the right decision.

This work contributes to the design of efficient and reliable image-based search engines. However, both for this module and the whole system, there is room for improvement.

5.2 Future Work

Here we list possible directions for future improvements:

- For compression-based methods, recent research focuses on deep methods, trying to train the features and codewords together to produce more compatible codes. However, as is discussed in this thesis, their performance in simple datasets like CIFAR10 is not a good indication of how they will perform in real problems. In practice, they tend to run into overfitting problems, which limits their usefulness. Therefore, there is still plenty of work to be done to make them practical.

- For graph-based methods, a challenge is that the construction of the graph generally takes very long for large-scale datasets. For example, constructing HNSW for the Google Landmark dataset takes approximately 1 day. The time cost of our newly proposed method, PQ+HNSW, is around 12 hours. It is shorter than HNSW but still a considerably long period. The graph construction can be done offline in our system so long building time is not constrained. But it brings inconvenience for parameter tuning and difficulty to scale to larger datasets. Attempts to reduce the complexity of the building might be highly beneficial.
- For hybrid methods, we have only studied the combination of PQ and HNSW carefully in this work. But in theory, PQ can be combined with any graph-based methods that construct graphs based on ℓ_2 distance. Moreover, hashing methods that preserve similarity in the original space and the coding space can be combined with any tree-based methods. Many other options are worth exploring.

Visualization: Examples

A

Figure A.1: Map of the World by Venetian monk Fra Mauro (1450)

Figure A.2: Code of Hammurabi

Figure A.3: Great Colonnade at Apamea

Figure A.4: Church of Holy Wisdom

Image Search Engine: Demo

Choose File No file chosen

Submit

Query:

Results:

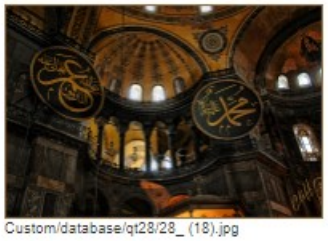

Custom/database/qt28/28_ (7).jpg


Custom/database/qt28/28_ (10).jpg


Custom/database/qt28/28_ (22).jpg


Custom/database/qt28/28_ (3).jpg


Custom/database/qt28/28_ (20).jpg


Custom/database/qt28/28_ (18).jpg


Custom/database/qt28/28_ (16).jpg


Custom/database/qt28/28_ (14).jpg


Custom/database/qt28/28_ (19).jpg


Custom/database/qt28/28_ (13).jpg


Custom/database/qt28/28_ (8).jpg


Custom/database/qt28/28_ (12).jpg

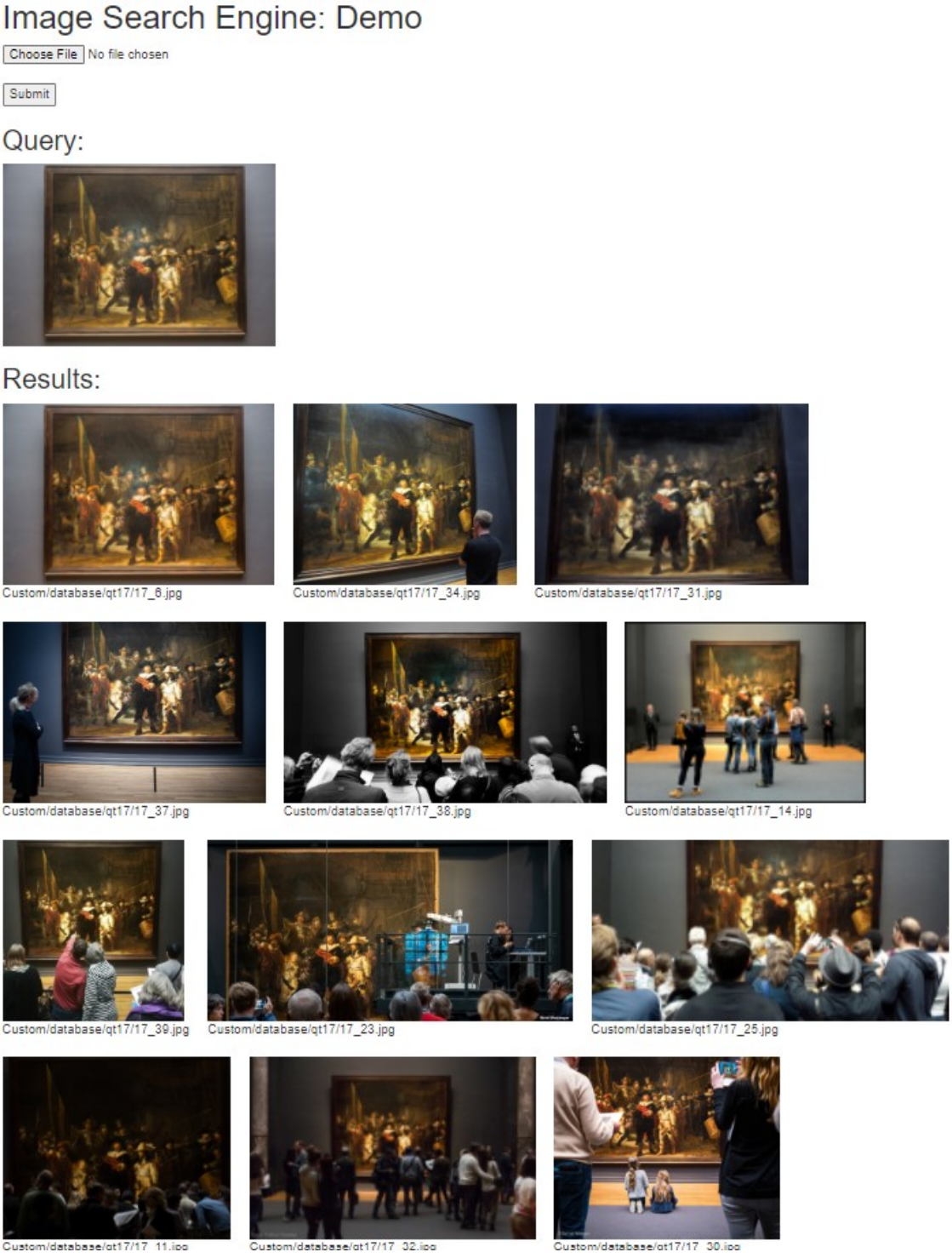

Custom/database/qt28/28_ (15).jpg


Custom/database/qt28/28_ (17).jpg


Custom/database/qt28/28_ (2).jpg


Custom/database/qt11/11_5.jpg

Figure A.5: The Night Watch by Rembrandt van Rijn (1642)



Bibliography

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018.

[3] 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.

[4] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660, 2005.

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[6] Erik Bernhardsson. Annoy: Approximate nearest neighbors in c++/python. https://pypi.org/project/annoy/. Python package version 1.17.0.

[7] Erik Bernhardsson. Nearest neighbors and vector models–part 2–algorithms and data structures. https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html. Accessed 14 Jun. 2022.

[8] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104, 2006.

[9] Lawrence Cayton. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*, pages 112–119, 2008.

[10] Shen Chen, Liujuan Cao, Mingbao Lin, Yan Wang, Xiaoshuai Sun, Chenglin Wu, Jingfei Qiu, and Rongrong Ji. Hadamard codebook based deep hashing. *arXiv preprint arXiv:1910.09182*, 2019.

[11] Yaxiong Chen and Xiaoqiang Lu. Deep discrete hashing with pairwise correlation learning. *Neurocomputing*, 385:111–121, 2020.

[12] Yudong Chen, Zhihui Lai, Yujuan Ding, Kaiyi Lin, and Wai Keung Wong. Deep supervised hashing with anchor graph. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9796–9804, 2019.

[13] Bo Dai, Ruiqi Guo, Sanjiv Kumar, Niao He, and Le Song. Stochastic generative hashing. In *International Conference on Machine Learning*, pages 913–922. PMLR, 2017.

[14] Qi Dai, Jianguo Li, Jingdong Wang, and Yu-Gang Jiang. Binary optimized hashing. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1247–1256, 2016.

[15] Sanjoy Dasgupta and Kaushik Sinha. Randomized partition trees for exact nearest neighbor search. In *Conference on learning theory*, pages 317–337. PMLR, 2013.

[16] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 224–236, 2018.

[17] Kamran Ghasedi Dizaji, Feng Zheng, Najmeh Sadoughi, Yanhua Yang, Cheng Deng, and Heng Huang. Unsupervised deep generative adversarial hashing network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3664–3673, 2018.

[18] Thanh-Toan Do, Anh-Dzung Doan, and Ngai-Man Cheung. Learning to hash with binary deep neural network. In *European Conference on Computer Vision*, pages 219–234. Springer, 2016.

[19] Thanh-Toan Do, Anh-Dzung Doan, Duc-Thanh Nguyen, and Ngai-Man Cheung. Binary hashing with semidefinite relaxation and augmented lagrangian. In *European Conference on Computer Vision*, pages 802–817. Springer, 2016.

[20] Shiv Ram Dubey. A decade survey of content based image retrieval using deep learning. *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.

[21] Mihai Dusmanu, Ignacio Rocco, Tomas Pajdla, Marc Pollefeys, Josef Sivic, Akihiko Torii, and Torsten Sattler. D2-net: A trainable cnn for joint description and detection of local features. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 8092–8101, 2019.

[22] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. Deep hashing for compact binary codes learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2475–2483, 2015.

[23] Kave Eshghi and Shyamsundar Rajaram. Locality sensitive hash functions based on concomitant rank order statistics. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 221–229, 2008.

[24] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[25] Steven Fortune. Voronoi diagrams and delaunay triangulations. *Computing in Euclidean geometry*, pages 225–265, 1995.

[26] Jerome H Friedman, Jon Louis Bentley, and Raphael A Finkel. *An algorithm for finding best matches in logarithmic time*. Department of Computer Science, Stanford University, 1975.

[27] Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[28] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[29] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[30] Junfeng He, Wei Liu, and Shih-Fu Chang. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1129–1138, 2010.

[31] Qinghao Hu, Jiaxiang Wu, Jian Cheng, Lifang Wu, and Hanqing Lu. Pseudo label based unsupervised deep discriminative hashing for image retrieval. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 1584–1590, 2017.

[32] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[33] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[34] Hervé Jégou, Laurent Amsaleg, Cordelia Schmid, and Patrick Gros. Query adaptative locality sensitive hashing. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 825–828. IEEE, 2008.

[35] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Qi Tian, and Bo Zhang. Min-max hash for jaccard similarity. In *2013 IEEE 13th International Conference on Data Mining*, pages 301–309. IEEE, 2013.

[36] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. Super-bit locality-sensitive hashing. *Advances in neural information processing systems*, 25, 2012.

[37] Qing-Yuan Jiang, Xue Cui, and Wu-Jun Li. Deep discrete supervised hashing. *IEEE Transactions on Image Processing*, 27(12):5996–6009, 2018.

[38] Qing-Yuan Jiang and Wu-Jun Li. Scalable graph hashing with feature transformation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[39] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[40] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[41] Yannis Kalantidis, Clayton Mellina, and Simon Osindero. Cross-dimensional weighting for aggregated deep convolutional features. In *European conference on computer vision*, pages 685–701. Springer, 2016.

[42] Benjamin Klein and Lior Wolf. End-to-end supervised product quantization for image search and retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5041–5050, 2019.

[43] Alex Krizhevsky and Geoffrey E Hinton. Using very deep autoencoders for content-based image retrieval. In *ESANN*, volume 1, page 2. Citeseer, 2011.

[44] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(6):1092–1104, 2011.

[45] Der-Tsai Lee and Chak-Kuen Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[46] Jiayong Li, Wing WY Ng, Xing Tian, Sam Kwong, and Hui Wang. Weighted multi-deep ranking supervised hashing for efficient image retrieval. *International Journal of Machine Learning and Cybernetics*, 11(4):883–897, 2020.

[47] Ning Li, Chao Li, Cheng Deng, Xianglong Liu, and Xinbo Gao. Deep joint semantic-embedding hashing. In *IJCAI*, pages 2397–2403, 2018.

[48] Ryan Li. Hnsw implemented by python. https://github.com/RyanLiGod/hnsw-python.

[49] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[50] Kevin Lin, Jiwen Lu, Chu-Song Chen, and Jie Zhou. Learning compact binary descriptors with unsupervised deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1183–1192, 2016.

[51] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[52] Bin Liu, Yue Cao, Mingsheng Long, Jianmin Wang, and Jingdong Wang. Deep triplet quantization. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 755–763, 2018.

[53] Haomiao Liu, Ruiping Wang, Shiguang Shan, and Xilin Chen. Deep supervised hashing for fast image retrieval. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2064–2072, 2016.

[54] Ying Liu, Dengsheng Zhang, Guojun Lu, and Wei-Ying Ma. A survey of content-based image retrieval with high-level semantics. *Pattern recognition*, 40(1):262–282, 2007.

[55] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[56] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *33rd International Conference on Very Large Data Bases, VLDB 2007*, pages 950–961. Association for Computing Machinery, Inc, 2007.

[57] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[58] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[59] Yusuke Matsui. Nano product quantization (nanopq): a vanilla implementation of product quantization (pq) and optimized product quantization (opq) written in pure python without any third party dependencies. https://github.com/matsui528/nanopq.

[60] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[61] James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on pattern analysis and machine intelligence*, 23(9):964–976, 2001.

[62] Marius Muja and David Lowe. Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, 5, 2009.

[63] Andrea Nanetti. Engineering historical memory. https://engineeringhistoricalmemory.com/. Accessed: 2022-04-30.

[64] Hyeonwoo Noh, Andre Araujo, Jack Sim, Tobias Weyand, and Bohyung Han. Large-scale image retrieval with attentive deep local features. In *Proceedings of the IEEE international conference on computer vision*, pages 3456–3465, 2017.

[65] Mohammad Norouzi, David J Fleet, and Russ R Salakhutdinov. Hamming distance metric learning. *Advances in neural information processing systems*, 25, 2012.

[66] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.

[67] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. *arXiv preprint cs/0510019*, 2005.

[68] Youngki Park, Heasoo Hwang, and Sang-goo Lee. A novel algorithm for scalable k-nearest neighbour graph construction. *Journal of Information Science*, 42(2):274–288, 2016.

[69] Filip Radenović, Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondřej Chum. Revisiting oxford and paris: Large-scale image retrieval benchmarking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5706–5715, 2018.

[70] Filip Radenović, Giorgos Tolias, and Ondřej Chum. Cnn image retrieval learns from bow: Unsupervised fine-tuning with hard examples. In *European conference on computer vision*, pages 3–20. Springer, 2016.

[71] Filip Radenović, Giorgos Tolias, and Ondřej Chum. Fine-tuning cnn image retrieval with no human annotation. *IEEE transactions on pattern analysis and machine intelligence*, 41(7):1655–1668, 2018.

[72] Jerome Revaud, Philippe Weinzaepfel, César De Souza, Noe Pion, Gabriela Csurka, Yohann Cabon, and Martin Humenberger. R2d2: repeatable and reliable detector and descriptor. *arXiv preprint arXiv:1906.06195*, 2019.

[73] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 151–162, 1975.

[74] Fumin Shen, Chunhua Shen, Qinfeng Shi, Anton van den Hengel, and Zhenmin Tang. Inductive hashing on manifolds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2013.

[75] Yuming Shen, Jie Qin, Jiaxin Chen, Mengyang Yu, Li Liu, Fan Zhu, Fumin Shen, and Ling Shao. Auto-encoding twin-bottleneck hashing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2818–2827, 2020.

[76] Anshumali Shrivastava and Ping Li. Densifying one permutation hashing via rotation for fast near neighbor search. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2014.

[77] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[78] Jingkuan Song, Tao He, Lianli Gao, Xing Xu, Alan Hanjalic, and Heng Tao Shen. Unified binary generative adversarial network for image retrieval and compression. *International Journal of Computer Vision*, 128(8):2243–2264, 2020.

[79] Jingkuan Song, Yang Yang, Yi Yang, Zi Huang, and Heng Tao Shen. Inter-media hashing for large-scale retrieval from heterogeneous data sources. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 785–796, 2013.

[80] Christoph Strecha, Alex Bronstein, Michael Bronstein, and Pascal Fua. Ldahash: Improved matching with smaller descriptors. *IEEE transactions on pattern analysis and machine intelligence*, 34(1):66–78, 2011.

[81] Haibo Su, Peng Wang, Lingqiao Liu, Hui Li, Zhen Li, and Yanning Zhang. Where to look and how to describe: Fashion image retrieval with an attentional heterogeneous bilinear network. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(8):3254–3265, 2020.

[82] Shupeng Su, Chao Zhang, Kai Han, and Yonghong Tian. Pytorch implementation for "greedy hash: Towards fast optimization for accurate hash coding in cnn" (nips2018). https://github.com/ssppp/GreedyHash.

[83] Shupeng Su, Chao Zhang, Kai Han, and Yonghong Tian. Greedy hash: Towards fast optimization for accurate hash coding in cnn. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 806–815, 2018.

[84] Giorgos Tolias, Ronan Sicre, and Hervé Jégou. Particular object retrieval with integral max-pooling of cnn activations. *arXiv preprint arXiv:1511.05879*, 2015.

[85] Godfried T Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.

[86] Ji Wan, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. Deep learning for content-based image retrieval: A comprehensive study. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 157–166, 2014.

[87] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):769–790, 2017.

[88] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Semi-supervised hashing for scalable image retrieval. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3424–3431. IEEE, 2010.

[89] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Sequential projection learning for hashing with compact codes. 2010.

[90] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

[91] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. *Advances in neural information processing systems*, 21, 2008.

[92] Tobias Weyand, Andre Araujo, Bingyi Cao, and Jack Sim. Google landmarks dataset v2-a large-scale benchmark for instance-level recognition and retrieval. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2575–2584, 2020.

[93] Jay Yagnik, Dennis Strelow, David A Ross, and Ruei-sung Lin. The power of comparative reasoning. In *2011 International Conference on Computer Vision*, pages 2431–2438. IEEE, 2011.

[94] Fan Yang, Ryota Hinami, Yusuke Matsui, Steven Ly, and Shin'ichi Satoh. Efficient image retrieval via decoupling diffusion into online and offline processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9087–9094, 2019.

[95] Tan Yu, Junsong Yuan, Chen Fang, and Hailin Jin. Product quantization network for fast image retrieval. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 186–201, 2018.

[96] Cao Yue, M Long, J Wang, Zhu Han, and Q Wen. Deep quantization network for efficient image retrieval. In *Proc. 13th AAAI Conf. Artif. Intell.*, pages 3457–3463, 2016.

[97] Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *International Conference on Machine Learning*, pages 838–846. PMLR, 2014.

[98] Ting Zhang, Guo-Jun Qi, Jinhui Tang, and Jingdong Wang. Sparse composite quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4548–4556, 2015.

[99] Wan-Lei Zhao. k-nn graph construction: a generic online approach. *arXiv preprint arXiv:1804.03032*, 2018.

[100] Liang Zheng, Yi Yang, and Qi Tian. Sift meets cnn: A decade survey of instance retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 40(5):1224–1244, 2017.