MONASH University

DOCTORAL THESIS

"What is a Picture Really Worth?"

Understanding High Dimensional Visual Data

Author:
Ben Harwood

Supervisor:
Prof. Tom Drummond

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

at the Monash University
Australian Centre for Robotic Vision *in the*
Department of Electrical and Computer Systems Engineering

All research documented within this thesis has been conducted under the supervision of Professor Tom Drummond. The following sections of the thesis incorporate material from previously published work:

- Section 2.3 and Chapters 3 and 4 introduce and expand upon "FANNG: Fast Approximate Nearest Neighbour Graphs"[1], published in the Proceedings of the IEEE Conference on Computer Vision and Patten Recognition (CVPR), 2016.
- Section 6.2 includes relevant portions of "Smart Mining for Deep Metric Learning"[2], published in the Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017. Section 6.1 expands upon collaborative material from this publication. The contributions of Vijay Kumar B G, Gustavo Carneiro and Ian Reid on the application of global loss functions have been omitted from this thesis.
- Sections 6.3 and 6.4 include material from "Deep Metric Learning and Image Classification with Nearest Neighbour Gaussian Kernels"[3], published in the Proceedings of the 25th IEEE International Conference on Image Processing (ICIP), 2018. Benjamin J Meyer is the largest contributor to the original publication, particularly in terms of hands-on experimental work. My contributions total to approximately 30% of the research completed by Monash University students.

Declaration of Authorship

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: 

Print Name: Ben Harwood

Date: 27/01/2019

Abstract

Researchers are increasingly using digital images to teach machines how to see and understand our physical world. The advent of big data has provided a vast wealth of images to enable these sophisticated learning algorithms. While conventional wisdom states that a picture is worth a thousand words and seeing is believing, it is now important to ask what a picture is really worth and how much trust we can place in vision alone. Modern machine learning algorithms have shown significant potential for automating complex tasks that have previously required human workers. Unfortunately, our ability to develop and tune these algorithms for real-world usage is constrained by the cost of processing large amounts of complex visual data. My research has focused on reducing these inherent costs by understanding and exploiting the intrinsic structures of high dimensional visual features that have been extracted from raw image data. Specifically, this research has culminated in the development of algorithms that achieve new levels of efficiency for retrieving and comparing large numbers of visual features. These algorithms have subsequently enabled the development of new machine learning techniques that extend our current ability to automate complex visual tasks.

Acknowledgements

Completion of this thesis document has at times posed a comparable challenge to that of the research contained within it. However, these challenges have remained achievable due to a wide network of helpful individuals. I am very appreciative of the support and guidance that has accompanied me throughout my candidature. This includes working closely with my supervisor Tom Drummond and his ongoing feedback and refinement of my research endeavours. I am particularly appreciative of the initial opportunity to embark on blue-sky research that consequently had a higher risk of getting going. Working in Tom's lab has enabled many career and personal development opportunities. Many of these opportunities are a direct result of the funding and knowledge pool produced from the launch of the Australian Centre for Robotic Vision. While my inclusion in the Centre has allowed me to experience countless insightful presentations and conversations, I am particularly grateful for the time and knowledge that my collaborators in Adelaide have shared with me while I was visiting there and during our spate of video conferencing. Their experience and commitment provided a considerable head start for my initial look into deep learning methods. Similarly, the knowledge and expertise that has been shared between the past and present members of our research group at Monash has offered a dependable and thought provoking workplace. Additionally, the astute thoughts of many members of the broader Monash community have helped to shape and direct my research. In particular, I would like to highlight my discussions with Kate Smith-Miles, David Boland, Peter Tischer, David Squire, Bill Corcoran, Lindsay Kleeman and Zixiang Xiong. Administrative processes throughout my candidature have been considerably less onerous due to the very notable efforts of Ros Rimington, Sandra Pedersen and Emily Simic. Their efficiency and punctuality has afforded me more time and energy for stressing over research matters, rather than the burdens of bureaucracy. These ongoing stresses have been considerably reduced due to the rich and supportive social environments I have enjoyed at lunchtime quizzes in the postgraduate lounge, during board game evenings held in our laboratory space, training with the on-campus Aikido club and attending potluck dinners held at various dining tables throughout Melbourne's suburbs. I also hold a great deal of gratitude for family and friends who might not fall within these social circles, but who have offered various distractions, which have then allowed me to resume my studies with a renewed enthusiasm. Of special note is the contributions of my father, who offered his time to proofread this document. Unequivocally, my deepest appreciation is for the patience and support that has been endlessly provisioned by Daveen Ma. Beyond her critical role in one particular debugging session, her ancillary contributions have certainly influenced the entirety of this thesis. Lastly I would like to acknowledge the fruitfulness of our vast capacity for procrastination; as with high dimensionality it is both a blessing and a curse.

Contents

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
GPGPU	**G**eneral **P**urpose computing on **GPU**s
FLOPS	**F**loating **P**oint **O**perations **P**er **S**econd
PCA	**P**rinciple **C**omponent **A**nalysis
FAST	**F**eatures from **A**ccelerated **S**egment **T**est
SIFT	**S**cale **I**nvariant **F**eature **T**ransform
GIST	**Gist** of a scene
BRIEF	**B**inary **R**obust **I**ndependent **E**lementary **F**eatures
ORB	**O**riented FAST and **R**otated **B**RIEF
BRISK	**B**inary **R**obust **I**nvariant **S**calable **K**eypoints
FREAK	**F**ast **Re**tin**A** **K**eypoint
ANN	**A**pproximate **N**earest **N**eighbour
FANNG	**F**ast **ANN** **G**raph
LSH	**L**ocality-**S**ensitive **H**ashing
MST	**M**inimum **S**panning **T**ree
RNG	**R**elative **N**eighbourhood **G**raph
SNG	**S**parse **N**eighbourhood **G**raph
SAT	**S**patial **A**pproximation **T**ree
FLANN	**F**ast **L**ibrary for **ANN**s
NSWG	**N**avigable **S**mall **W**orld **G**raph
CNN	**C**onvolusional **N**eural **N**etwork
ReLU	**Re**ctified **L**inear **U**nits
NMI	**N**ormalised **M**utual **I**nformation
t-SNE	**t**-Distributed **S**tochastic **N**eighbour **E**mbedding

List of Symbols

$\mathcal{X}$	a dataset of vectors
$\mathbf{x}_i$	the ith vector in a dataset
x_j	the jth element of a vector
n	the total number of vectors in a dataset
m	the total number of elements in a vector
$\mathcal{S}$	a set of nearest neighbour correspondences
$\mathcal{S}_i$	the nearest neighbours of $\mathbf{x}_i$
$dist_E$	the Euclidean distance function
$dist_H$	the Hamming distance function
r	the radius of a sphere
θ	the angle between two vectors
$\boldsymbol{\mu}$	the centre of a Gaussian kernel
σ	the standard deviation of a Gaussian kernel
k	the number of elements in a set
$\dim_H$	the Hausdorff dimensionality function
ρ	the local density function
N	the distance error function
$\mathcal{G}$	an indexing graph
V	a set of vertices
v_i	the ith vertex in a graph
E	a set of directed edges
e_i	the ith directed edge in an ordered edge list
$E[i]$	the set of all edges from v_i
E_{avg}	the average out-degree of a graph
E_{max}	the edges of a fully connected graph
d_w	the diameter of a weighted graph
$\mathcal{Q}$	a set of query vectors
$\mathbf{q}$	a query vector
U	a set of visited vertices
U_{max}	the maximum search size
U_0	the starting vertex
V_q	the target vertex
$\mathcal{P}$	a priority queue

t	the occlusion threshold
τ	the exact nearest neighbour margin
τ_{max}	the largest nearest neighbour distance
T	the maximum size of a truncated edge list
p	the number of graph construction iteration
f	a function mapping from image space to feature space
θ_f	a set of learned network weights
$\mathcal{T}$	a set of labelled training images
$\mathbf{w}$	a vector of classifier weights
w_i	the classifier weight for the ith training sample
C	the number of image classes
C_{avg}	the average number of images per class
C_{max}	the maximum number of images in a class
Y	a set of class labellings
y_i	the class label of the ith training image
L	a loss function
ε	the number of training epochs
B	a set of training batches
b_i	the ith batch in a training epoch
B_ε	the number of batches per training epoch
$\mathbf{x}^a$	an anchor image
$\mathbf{x}^p$	a positive training image
$\mathbf{x}^n$	a negative training image
$\mathbf{x}^p_{\text{NN}}$	the nearest positive training image to an anchor
d_{NN}	the distance between $\mathbf{x}^a$ and $\mathbf{x}^p_{\text{NN}}$
η	the triplet loss margin
ν	the number of mined triplets per anchor
ε_{min}	the first epoch of triplet mining
ε_{max}	the last epoch of triplet mining
κ	the smart mining boundary scale
$\boldsymbol{\kappa}$	a set of recent boundary scales
$\boldsymbol{\delta}$	a set of recent training errors
δ_t	the target training error

Chapter 1

Introduction

The biological world has selected visual perception as a strong method for organisms to learn about and interact with their surrounding environment. In the digital world, the advent of compact, low cost, light weight and high resolution image sensors has enabled computer vision to enter many facets of our everyday lives. Supporting this, is the ongoing advancements in high performance and low energy computing that provide a way to process large amounts of image data in a reasonable amount of time. Together, these technologies are expanding the types of physical entities that can utilise the power of visual perception. The ubiquity of such devices has warranted the continued development of algorithms that implement the functionality of biological vision though digital means. Perhaps unsurprisingly, reverse engineering the results of around 40 million years of evolution remains as a substantial challenge. However, a secondary by-product of this ubiquitous hardware is the amount of image data being generated and stored from this multitude of sources. In combination with the high performance computing hardware, large sets of image data have enabled substantial breakthroughs in data driven machine learning. These techniques allow for previously hand-engineered sections of computer vision pipelines to be replaced with sufficiently large non-linear models. Given enough training data, model parameters can converge on a configuration that outperforms existing benchmarks from both the digital and biological worlds.

In our preliminary research we evaluated the performance of state-of-the-art feature based image matching techniques, with a focus on real-time performance in a constrained computing environment. The first step was to capture a series of images

and then break them down into small and informative regions. These regions can then be searched and compared in order to provide higher level information, such as the relative positions of physical objects shown within the images. If we are using this data to answer questions about a particular type of high level information, then we would like to locate and extract only the relevant information from the available image data. Furthermore, achieving real-time performance with limited computational resources requires an efficient implementation. As such, we would like to represent and access the information in a way that is both fast and scalable. These preliminary experiments highlighted the importance of finding and mitigating computational bottlenecks and also raised the somewhat philosophical question of what information is actually worth retaining.

My research has focused on reducing some of the inherent costs that are faced when processing large quantities of visual data. To achieve this, I believe it is important to understand and exploit the intrinsic structures of high dimensional visual features that have been extracted from raw image data. A deeper understanding of the image data itself can also lead to improvements in the way we extract information from this data. Specifically, we have developed tools that can feasibly process the large amounts of data that are required to reveal the detailed structure of high dimensional image spaces. This information was then applied to the nearest neighbour search problem; a well-studied computational bottleneck that is found in many computer vision pipelines. Our new search techniques were then utilised for the problem of embedding learning with deep convolutional neural networks. When properly trained, these networks can model a non-linear dimension reduction that ultimately distils an input image down into a compact vector of the key information found within that image data.

1.1 Experimental Resources

The experiments reported throughout this thesis were carried out using a variety of generated and publicly available computer vision datasets. Experiments were also executed across several different specifications of computational hardware. Here

we summarise the key properties of the datasets and hardware used throughout our research.

1.1.1 Datasets of Interest

Table 1.1 details the properties of datasets that where generated for various experiments throughout this thesis. Each dataset has been generated with a random uniform sampling of vectors from a particular high dimensional space. Firstly, the two dimensional Serpinski fractal data is utilised in Section 2.3.3 to validate the accuracy of particular intrinsic measurements. The floating point vectors from within a hypercube and the binary vectors from the corners of a hypercube are both used for demonstrating properties of high dimensional data, as well as computing reference values. These random vectors are a good representation of data that can fill a high dimensional space in an unbiased way.

TABLE 1.1: Generated datasets of random vectors.

Vector type	Vector length	Test vectors
Serpinski triangle	2	10k
Hypercube volume	20	100k
Hypercube corners	192	100k

Feature vectors are a common format for representing key information that has been extracted from visual data. In general, a feature vector is the output of function that maps from a local or global image region to a compact vector. Each feature type has a unique mapping function that defines exactly how an image region will be reduced to its vector form. This function also defines a feature space, which contains the full set of vectors that can be produced by that particular feature mapping. The information that is retained by each feature vector will typically be invariant to a number of different properties. These properties are application dependent, but commonly include spatial and chromatic changes such as the orientation and illumination of a local region, or the pose and colour variations expressed by a particular type of object. Due to the removal of this unwanted data, feature extraction methods can typically be viewed as a form of dimension reduction or lossy data compression.

Tables 1.2 and 1.3 present the feature sets used throughout many of our experiments. While the floating point features were publicly available in vector form, each set of binary features has instead been constructed by processing a dataset of image regions. The extraction of each binary feature type was performed using the parameters values recommended by their respective authors. A random sampling of image regions were withheld from each of these datasets in order to form the query sets.

TABLE 1.2: Public datasets of floating point feature vectors.

Feature type	Vector length	Source region	Source images	Test vectors	Query vectors
SIFT [4]	128	16×16	BIGANN [5]	1k, 10k	100
			BIGANN 1B [6]	100k, 1M, 5M, 20M	10k
GIST [7]	960	32×32	BIGANN [5]	1M	1k

TABLE 1.3: Generated datasets of binary feature vectors.

Feature type	Vector length	Source region	Source images	Test vectors	Query vectors
ORB [8]	256	31×31	Photo Tourism [9][10]	1.5M	50k
BRISK [11]	512	32×32	Photo Tourism [9][10]	1.5M	50k
FREAK [12]	512	60×60	Photo Tourism [9][10]	1.5M	50k
Binboost [13]	256	32×32	Photo Tourism [9][10]	1.5M	50k
			80M Tiny Images [14]	75M	10k

For each predefined feature space, we assume that the mapping is correctly retaining all relevant information. This means that the distance between two feature vectors can be directly equated to how closely they correspond to each other. Using this assumption, ground truth correspondences were computed for each query vector using an exhaustive linear search over the associated test vectors. In the case of duplicate test vectors or tied distances, all equidistant features were considered to be equivalent. While duplicate vectors were present in most datasets, tied distances between non-duplicate vectors were predominantly found in the 256 bit feature sets.

Lastly, Table 1.4 contains the details of datasets used for learning feature spaces. Rather than assuming ideal performance from a feature mapping, here we use images with known class labels to evaluate feature correspondences. Each dataset contains a large number of classes, with a varying number of images per class label. Each individual class is a fine-grained category, such as a particular species of bird. During

experiments, half of the image classes are processed in order to learn a new feature space, while the other half of the classes are used to test performance. Ground truth bounding boxes are available for the target object in each image, but this information is not used in any of our experiments. However, we do note that these datasets have been constructed from images with a single target object that is typically located within the central region of each image.

TABLE 1.4: Public datasets of transfer learning images.

Image set	Total images	Total classes	Training classes	Test classes
CUB Birds [15]	11788	200	100	100
Stanford Cars [16]	16185	196	98	98

1.1.2 Computational Hardware

General purpose computing on graphics processing units (GPGPU) is a relatively new and still expanding area of parallel computing. GPGPU provides a programming interface to take advantage of the highly parallel architecture found in graphics processing units (GPUs). While current computational processing units (CPUs) can run tens of threads in parallel, GPUs are capable of executing thousands of parallel threads. The GPU capabilities are possible in part due to a slower clock rate, but also due to a hierarchical structure of process execution and memory access. Due to this considerably different architecture, GPGPU programming only loosely reflects the paradigms of traditional CPU programming. As such, fully utilising the parallel capabilities that are offered by GPUs still remains a considerable challenge for programmers.

As seen in Figure 1.1, the GPU architecture can be divided into three hierarchical levels. The interface between the CPU and GPU is implemented at the global level. This level has access to the largest and slowest memory, which is also the only memory that can be accessed from the CPU. In terms of processing, the global level manages the scheduling of tasks on the other levels of the hierarchy. These tasks are defined by kernels, which are modules of compiled code that have been launched from the CPU. At the shared level of the hierarchy, everything is divided into blocks

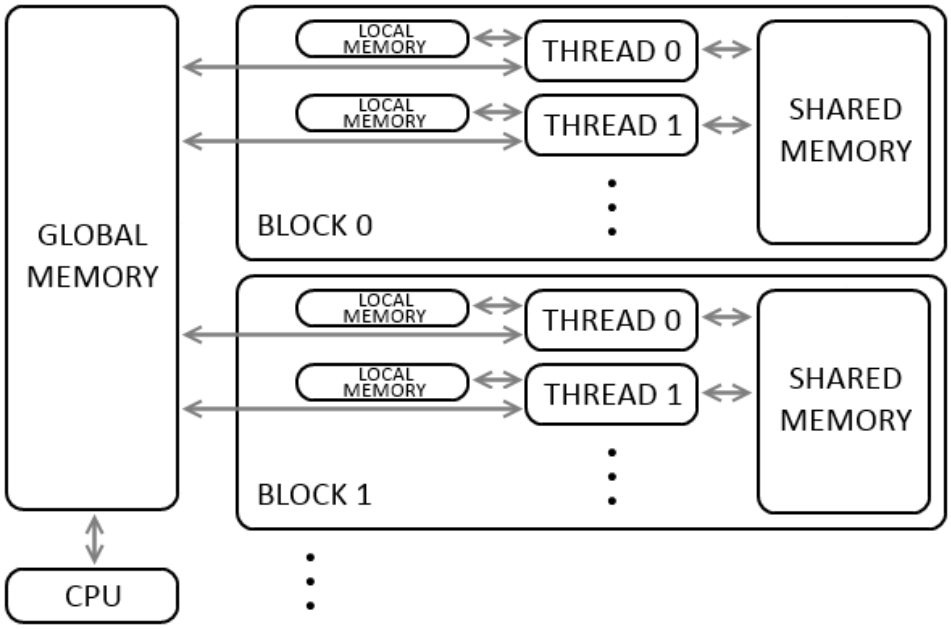


FIGURE 1.1: The internal architecture of a GPU contains three tiers of increasing computational speed and decreasing memory capacity.

that receive and compute portions of kernels. Each block is limited to tens of kilobytes of memory, however this memory can be accessed roughly ten times faster than global memory. Blocks are designed to operate independently of each other, having no direct means of communicating with each other and no guarantee of the order in which they will be executed. Lastly, the local level divides blocks into individual threads. Once again, threads have access to a considerably smaller amount of very fast memory and generally operate independently of each other. However, there is increasing support for functions that share information between a small number of consecutive threads.

TABLE 1.5: Operating parameters of GPU hardware.

Model	Total CUDA cores	Core clock (GHz)	Processing power (TFLOPS)	Global memory (GB)	Memory clock (GHz)	Memory bandwidth (GB/s)
GTX Titan X	3072	1.00	6.1	12.2	3.51	336
Tesla P40	3840	1.30	12	22.9	3.62	346

Table 1.5 provides a number of key specifications from the GPUs used within our research. In addition to the values shown, each GPU allows for a maximum of 1024 threads, $49,152$ bytes of shared memory and $65,536$ local registers per block. The

major difference between the two models, is the additional global memory in the Tesla P40. This memory allows for much larger datasets to be processed on a single GPU, although techniques do exist for running a single process across many GPUs.

1.2 Summary of Contributions

The initial goals of this candidature were to identify common computational bottlenecks and to further understand the utility of information found in high dimensional visual data. This knowledge then led to the development of algorithms that improve upon existing computer vision systems. The key outcomes of our research can be summarised as follows:

- Providing additional insight into the structure of high dimensional image spaces with the development of an efficient Hausdorff dimensionality formulation that is well-suited for parallelisation. This measure has then been utilised as a tool that is capable of processing the large amounts of data required for this type of analysis.
- The development of Fast Approximate Nearest Neighbour Graphs (FANNGs) a state-of-the-art method for indexing and searching high dimensional visual data. The FANNGs combine efficient algorithms for constructing indexing graphs as well as rapid traversal of these graphs.
- Additional analysis of the approximate nearest neighbour (ANN) search problem in the context of binary data. We show that unlike existing ANN algorithms, FANNGs are able to operate effectively across both real valued and binary feature spaces.
- Integrating FANNGs into the embedding layer of deep convolutional neural networks (CNNs) in order to accelerate and refine the learning of feature spaces. The additional information provided by FANNG queries has enabled new state-of-the-art results in the area of transfer learning.
- In utilising FANNGs to accelerate our deep learning architectures we also demonstrate that our graphs are resilient to changes in the indexed data. This

allows for a graph constructed on a particular embedding space to be utilised across many other spaces that have a similar structure.

1.3 Thesis Overview

In this chapter we have outlined the technological environment and real-world challenges that have motivated this research. We then introduced and discussed a number of datasets and hardware specifications that are used throughout our experiments. Lastly, we have summarised the key contributions of knowledge that are presented in this thesis.

Looking forwards, Chapter 2 introduces several fundamental properties of high dimensional data and contrasts them with the expectations that we have gained while living in a three dimensional world. Each of these properties is discussed in terms of the challenges they present for computer vision algorithms that are operating with high dimensional image data. For the final property, Hausdorff dimensionality, we present a novel formulation that has been designed for GPGPU acceleration. The accuracy of this method is demonstrated with a synthetic dataset before it is applied to non-synthetic data.

Chapter 3 explores the strengths and weaknesses of popular approaches to the approximate nearest neighbour search problem. These methods are broadly categorised by the type of indexing structure that each of them use to divide up a high dimensional space. As such, we only consider methods that are considered to be suitable for indexing high dimensional visual data.

Chapter 4 builds upon the ideas presented in the previous two chapters and introduces our FANNGs, a new traversable indexing graph for high dimensional visual data. Numerous variations of the search and construction algorithms are presented and evaluated. Our best performing algorithm is then contrasted with other competitive methods.

Chapter 5 continues to explore the performance capabilities of the FANNGs. In this chapter we consider the unique challenges that are faced when indexing binary feature vectors instead of real valued data. Analysis and discussion is presented around why most indexing schemes are unable to operate effectively across both of these domains.

Chapter 6 looks at applying our FANNGs to the domain of machine learning. Here we explore two related ways to utilise approximate nearest neighbour search in order to accelerate training and refine performance when learning a feature space. Firstly, we build upon the previous successes of triplet neural networks and close the triplet selection loop in a more informed way. Secondly, we define a novel loss function that can better utilise the information provided by our FANNGs in order to surpass the performance of triplet networks. Computation of this loss function is made tractable by the stability of the FANNGs even when working with outdated data.

Lastly, in Chapter 7 we once again summarising the key contributions of knowledge that this research has presented. Finally, we conclude by briefly exploring a number of promising directions in which our research could expand in the future.

Chapter 2

Properties of High Dimensional Image Spaces

Over the past few decades as storage capacity and data transfer rates have increased, there have also been increases in the number of data sources and the types of data being generated and collected. These factors have given rise to the term 'big data', which is used to describe data sets which are so large that they require new tools in order to manage and process the information they contain. When there is very little cost associated with collecting data, it is easiest to collect all of the data available from as many sources as possible just in case it might be of some later use. Additionally, increase in the quantity of data can offer benefits such as smoothing out anomalies or providing more information on low frequency events. However there is still a high cost associated with extracting information from data, and this relates to both the complexity and the quantity of data. An increase in the quantity of data will result in a linear increase in the processing time for computing independent properties of samples in a dataset, or an exponential increase for identifying correlations. Data of increasing complexity, sometimes called 'rich data', will typically compound these processing costs with an additional exponential growth in the cost of evaluating each independent property.

Visual data is inherently complex in nature. Modern digital images are likely to have at least one million pixels, with each pixel showing one of 16 million colours, and each colour can potentially be chosen independently of all other pixels. Writing down the total number of images that can be represented in this way would require

over seven million decimal digits. This presents an absurd potential for representing visual information. However, the majority of images that can be represented in this way would be filled with coloured static. In contrast, the natural images that we could capture with a typical digital camera contain a significant amounts of continuous, smooth and repeating structures. Looking for structured relationships and patterns between different combinations and thresholds of pixels gives rise to the wealth of information that can be represented within an image. But both the number of pixels and the depth of colour contribute to the cost of processing visual data.

If we consider an image of m total pixels, we can then define this image as a vector $\mathbf{x} \in \mathbb{R}^m$, where each x_i is then the colour value of the i^{th} pixel in the image. Here we say that our image $\mathbf{x}$ has m extrinsic dimensions, as it is defined in an m dimensional space and is made up of m distinct values that may or may not be independent of each other. While the rectangular grid of pixels does give images an inherent two dimensional structure, the relationships shared between adjacent pixels are not always more important than the ones between non-adjacent pixels. Hence, it can be useful to think of an image as a generalised high dimensional vector rather than a two dimensional object. This is similar to how visual features, such as those described in Section 1.1.1, represent local image regions as compact vectors with no more than a few hundred or thousand extrinsic dimensions. Any set of feature vectors or images $\mathcal{X} \subset \mathbb{R}^m$ will only ever contain a discrete subsampling of the complete high dimensional space $\mathbb{R}^m$.

High dimensional spaces, including those occupied by interesting visual data, often follow unintuitive rules when compared to the low dimensional structures we are used to viewing.[17] A key challenge of processing high dimensional data is to be aware of the relevant similarities and differences between low and high dimensional data.[18] Since we have defined our data vectors in $\mathbb{R}^m$, we can compute the Euclidean distance

$$dist_E(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{m} (\mathbf{x}_{i,k} - \mathbf{x}_{j,k})^2} = \sqrt{(\mathbf{x}_i - \mathbf{x}_j).(\mathbf{x}_i - \mathbf{x}_j)} \tag{2.1}$$

between two vectors regardless of their extrinsic dimensionality. The Euclidean distance provides an intuitive way to form quantitative comparisons and can be directly applied as a function for scoring correspondences between feature vectors. Computing correspondences in this way implies that the feature space has been constructed such that the distance between true correspondences is typically smaller than the distance between false correspondences. In other words, image regions that have certain properties in common should produce features that are closer together when compared to the features from image regions with differing properties. Identifying these correspondences can then enable practical computer vision tasks such as segmentation. At a basic level, the process of segmentation can be described as bounding regions of a space so that similar data is grouped within each bounded region. In a well structured feature space, these regions can be used to classify new data by generalising with the currently available data. In this chapter we'll continue to look at useful properties of high dimensional visual data that help us to build a clearer understanding of the structures that are formed by these spaces.

2.1 Distribution of High Dimensional Data

When considering the digital representations of images and feature vectors, each extrinsic dimension is often bounded by a minimum and maximum value. An example of this would be a pixel ranging from fully black, through all of the greys until it is fully white. These limited domains act to contain the samples from an m dimensional dataset to within an m-cube. Again assuming that each dimension is independent, a dataset will uniformly fill the area of its bounding hypercube. This behaviour is shown in Figure 2.1 for low numbers of dimensions. To achieve the same density, the two dimensional square requires n^2 samples compared with the n random samples of the one dimensional line segment. We can achieve a reasonable visual representation of three dimensional points in a cube with a sparser projection of n^2 samples onto what is ultimately a two dimensional diagram. Here, a third dimension of information is represented by shading the points according to their height from the bottom face of the cube. As the number of dimensions continues

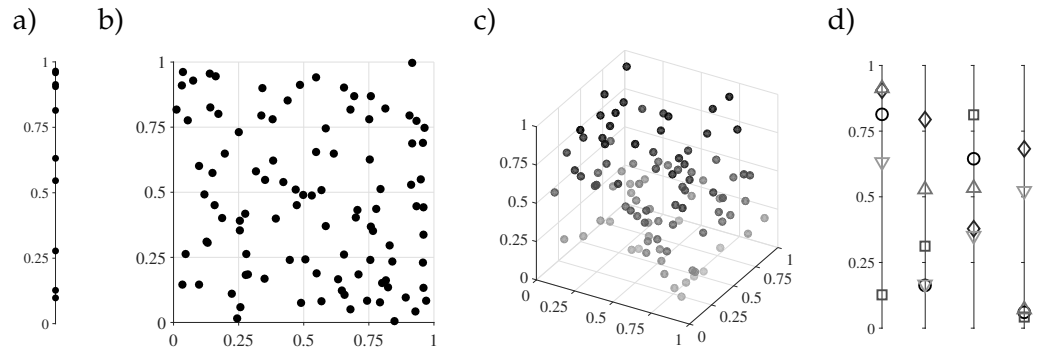


FIGURE 2.1: n-cubes of increasing extrinsic dimensionality filled uniformly with random data. a, b, c and d) 10, 100, 100 and 5 samples in 1, 2, 3 and 4 dimensions respectively.

to increase, visual representation becomes more difficult due to the increasing difference between the two dimensions of the diagram and the higher extrinsic dimensionality of the data. In four dimensions we can assign a unique shape to each sample and show each dimension independently. However, it is now much more difficult to visually assess useful properties such as the four dimensional Euclidean distance between any two samples. Ultimately, high dimensional diagrams are unable to convey an intuitive sense of the global relationships found within the data.

The lack of a natural way to view high dimensional structures results in many properties seeming unintuitive when they are first encountered. A two dimensional square has fewer corners than a three dimensional cube, and intuitively, higher dimensional hypercubes will have an increasingly large number of corners. The number of corners grows exponentially, with an m-cube having 2^m corners. This arises from each new dimension having a positive and negative side, that when combined with each corner in one fewer dimensions, will result in double the number of corners. This rapid growth in corners has a strange and unintuitive effect on the space contained within the hypercubes. If we first consider the floor of a square room in our natural three dimensional environment we can mentally partition the space into a large central region surrounded by four small corner regions. Now if we randomly select a number of locations on the two dimensional surface of the floor, then Figure 2.2a confirms that we should expect most of these locations to fall into the shaded central region of the room. However, as we begin to add additional randomly distributed dimensions to our room there is a very high chance that all of the

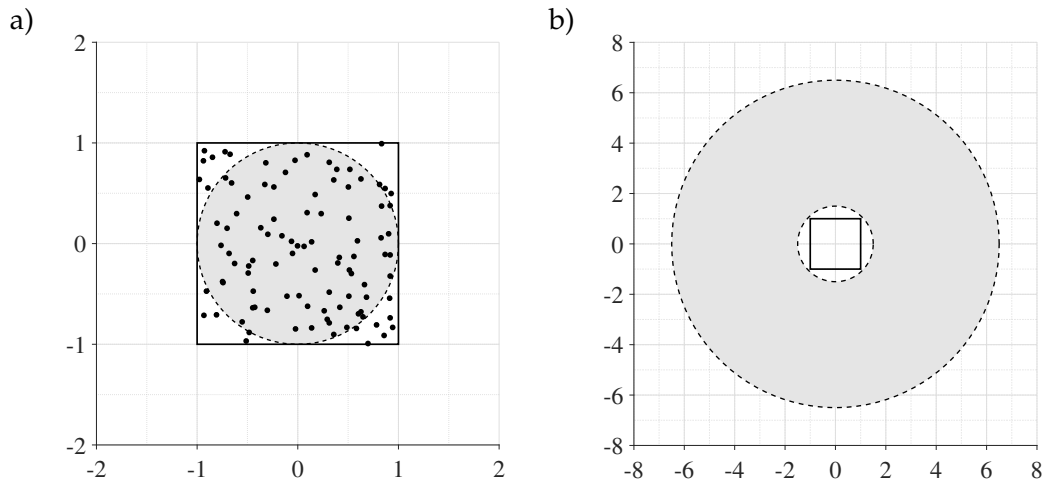


FIGURE 2.2: a) Uniform random data in a 2-cube will largely occupy the central region. b) Distributing samples throughout a 12-cube places over 99% of the samples at a distance within the shaded region when considering the Euclidean distance from the origin.

locations will end up in the corner regions. Figure 2.2b illustrates that when another ten dimensions are added to the same two dimension data, the distance from each location to the centre of the hypercube will typically fall within the bounds of the large shaded region. Comparing this range of distances with the two dimensional projection of our room shows that the locations are not only likely to be outside of a central hyperspherical region, but expected to fall entirely outside of the original room.

We begin to explain this behaviour by considering the Euclidean distance between the centre of a hypercube and each of its corners. For an m-cube with a side length of 2, each corner will be $\sqrt{m}$ distance from the centre. So as m increases, opposing corners become further apart despite all of side lengths remaining constant. Additionally, because the centre of each m-face will also remain at a distance of 1 from the centre of the hypercube, the radius of our central region will stay at a constant value of 1. Figure 2.3 further illustrates this property by considering all possible pairwise distances within m-cubes with a side length of 2. The plotted curves represent the relative likelihood that two randomly chosen locations from within an m-cube with be exactly a certain distance apart. As the number of extrinsic dimensions increases, the mean of these distributions moves away from zero. As such, we expect that higher dimensional hypercubes will largely contain locations that are further apart.

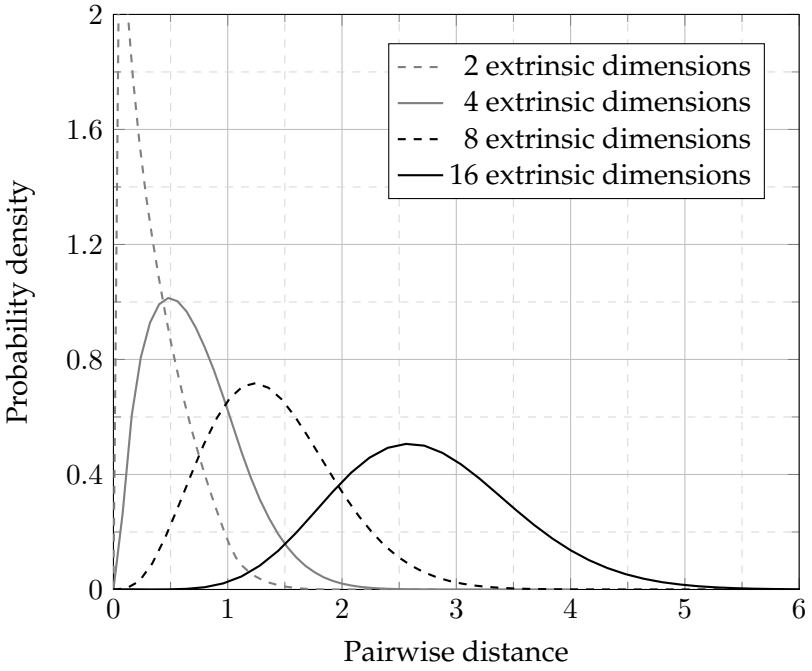


FIGURE 2.3: Probability density functions for the Euclidean distance between pairs of uniform random samples within a hypercube of increasing dimensionality.

If we continue to examine the curves in Figure 2.3, we can see that as the number of dimensions increases, the distributions increasingly spread out to cover the larger range of possible pairwise distances. However, this spread does not keep up with the rate at which the mean is moving away from zero. With the 8 dimensional data, it is already more likely than not, that two random locations will have a distance between them that is larger than the radius of our central region from Figure 2.2a. We can see this trend continue, with pairwise distances in the sixteen dimensional cube being, on average, larger than the diameter of the central region. In general, we expect that even for locations that only deviate from the centre of the hypercube by a small amount in each dimension, the sum of these deviations will continue to increase along with the number of dimensions. It is inevitable that these total distances will become large relative to the side length of our hypercubes.

When applying this understanding to computer vision, we can expect that all high dimensional visual data will comprise mostly of samples that are spread throughout the corner regions of a hypercube. For pairwise distances, such as those used for calculating feature correspondences, we can always expect to see a larger average

distance between higher dimensional feature vectors regardless of changes to the total number of samples in our dataset. With these properties in mind, we will now look at two common ways to segment high dimensional data. The primary aim of these segmentations is to utilise the relative distances within a dataset to then classify homogeneous regions of the data space. When looking for fewer segmented regions than there are samples in the dataset, dividing up a space based on relative distances can allow for efficient discovery of other relationships within the data. This is especially true in situations where a hierarchical segmentation is used.

2.1.1 Hyperplane Boundaries

Hyperplanes are a computationally efficient way to linearly separate regions of a hyperspace. We first consider that a one dimensional line can be used to partition two regions of a plane and that a two dimensional plane can then partition two regions of a three dimensional volume. In general we can define an m dimensional hyperplane as an $m-1$ dimensional Euclidean subspace that covers all samples of the form

$$\mathbf{a}.\mathbf{x} = b \tag{2.2}$$

where $\mathbf{a}$ is an m dimensional vector of constants with $|\mathbf{a}| \neq 0$ and b is a scalar constant. All samples from the hyperplane then form a connected region that separates the other m dimension samples into the two sets defined by

$$\mathbf{a}.\mathbf{x} < b \tag{2.3}$$

and

$$\mathbf{a}.\mathbf{x} \geq b. \tag{2.4}$$

Computing these sets is very efficient when $\mathbf{a}$ contains only one non-zero value. This removes the need for the vector dot product in Equations 2.3 and 2.4, and only requires accessing a single dimension x_i when classifying a sample $\mathbf{x}$. These efficiency boosts are only possible due to the fact that hyperplanes of this form will always divide a single extrinsic axis, while being parallel to all other axes. As such, these

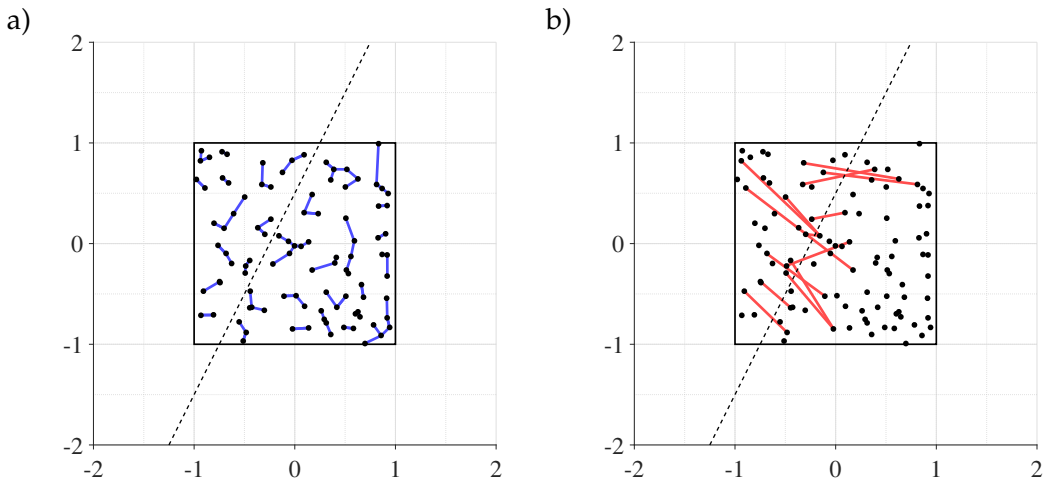


FIGURE 2.4: a) Uniform random data in a 2-cube is partitioned by a straight line that preserves groups neighbouring samples. b) Distributing the same data throughout a 12-cube, the linear partition extends through additional dimensions and now separates many samples from their closest neighbour.

axis-aligned planes are commonly used in low dimensional data structures such as quad-trees and oct-trees, as well as k-d trees in higher dimensions.

We have previously mentioned how correspondences can be defined by applying the Euclidean distance to pairs of vectors within a dataset. When using a hyperplane to divide our data, we would like to achieve a partitioning that does not separate any of the stronger correspondences. Figure 2.4a illustrates a set of closely corresponding samples in a 2-cube. For this low dimensional data, we can successfully define a hyperplane (in this case a line) that divides the data into two groups that do not break any of the correspondence lines. Figure 2.4b projects this division through an additional ten extrinsic dimensions and then considers the closest correspondences of the now twelve dimensional data. The hyperplane now separates many samples from their closest correspondence, each of which is illustrated by a broken correspondence line. As previously seen, the dimensions that are not shown in this figure are likely to account for a considerable portion of the total distances between the samples.

Practical issues with these partitionings are explored using a small dataset of one thousand SIFT feature vectors, each with 128 extrinsic dimensions. Figures 2.5a and

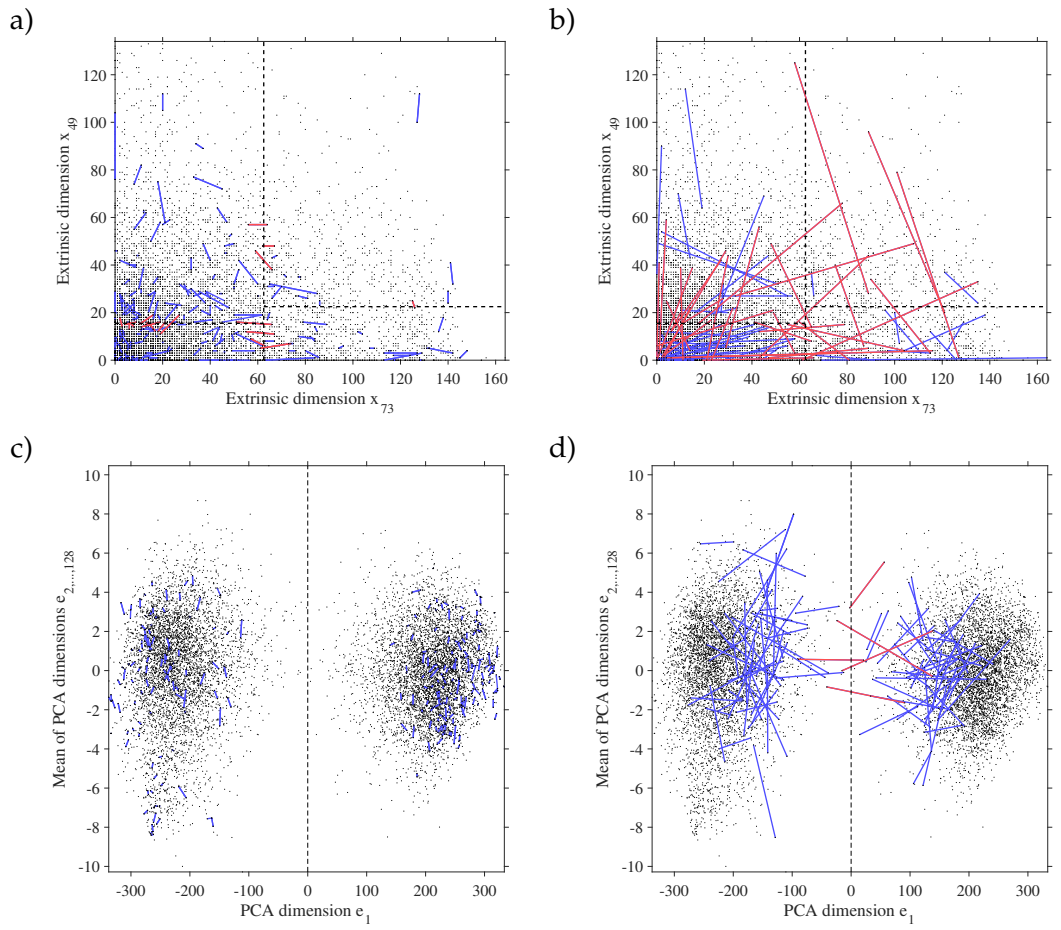


FIGURE 2.5: The impact of hyperplane partitions on SIFT correspondences. a) Best case correspondences with a partitioning of high variance axes. b) Worst case correspondences with the same partitions. c) Best case correspondences with a partition along the first eigenvector. d) Worst case correspondences with the eigenvector partition.

2.5b each partition the data using a computationally efficient hyperplane that divides a single extrinsic axis. The real-world data has less uniformity than a synthetic dataset and this results in some dimensions being easier to partition than others. With this method of partitioning, it would be ideal for the data to be spread with high variance along each extrinsic axis. In these plots we have computed the two highest variance dimensions and then projected the feature vectors onto these two extrinsic axes. In Figure 2.5a we plot the five hundred strongest correspondences, which equates to the five hundred feature vectors with the shortest distance to the next closest feature in the full 128 dimensional space. Figure 2.5b shows the same projection of the feature vectors but uses the weakest one hundred correspondences, which in this case is the one hundred feature vectors with the largest distance to the

next closest feature. For both plots, the data is firstly partitioned in half along the highest variance axis and then each half is repartitioned along the second axis. Partitioning the data in this way creates a natural hierarchical structure to the partitions, which is utilised by k-d trees to separate the data with a roughly equal number of samples on each side. In both the best and worst case scenarios, when using either the strongest or weakest correspondences, we find that each of the hyperplanes separate at least some of the corresponding feature vectors.

Figures 2.5c and 2.5d use a less restrictive partitioning that allows for more optimal selection of the first hyperplane boundary. Applying principal component analysis (PCA) to the dataset allows us to compute the direction of highest variance found across all possible orientations of the data. To achieve this, we partition the data using the mean value when projecting the data onto the first eigenvector of the PCA analysis. We then plot these projected values against the average projected value across the other 127 reorientate dimensions. This analysis reveals two easily separable clumps of data that are inherent to the structure of the SIFT space (each representing an extrema in the Difference-of-Gaussians neighbourhoods). In Figure 2.5c we see that a single partition can be performed without separating any of the five hundred strongest correspondences. However, this would no longer be the case if we attempted a second partition along the next eigenvector. Figure 2.5d further highlights the difficulty of partitioning high dimensional data, with a number of the one hundred worst case correspondence lines being broken by this well selected hyperplane.

Continuing to subdivide a dataset with hyperplanes that bisect high variance axes will produce the branching structure used by k-d trees. Figure 2.6 provides further analysis of the effects that these hyperplane divisions have on randomly sampled data with varying dimensionality. In Figure 2.6a we show the fraction of samples that have become separated from their closest corresponding sample when iteratively increasing the depth of the trees. The low dimensional data is able to be divided many times with minimal separation of correspondences. Then, at the deepest levels of this tree the number of samples in each partition is very small and so separations

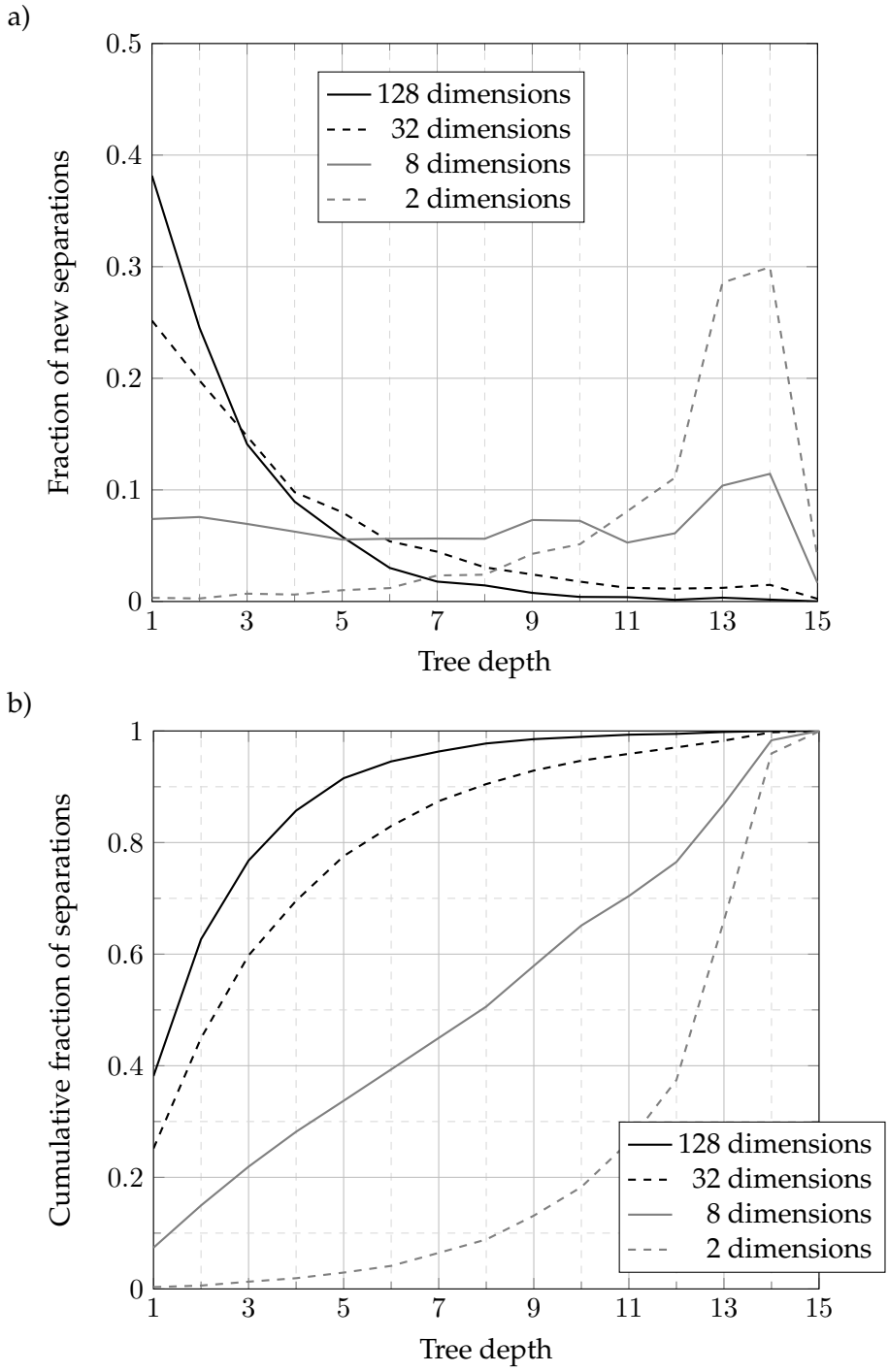


FIGURE 2.6: The distribution of separated correspondences in k-d trees built with uniform random data. a) The fraction of correspondences that have become separated at each depth in the trees. b) The cumulative sum of separated correspondences at each depth.

become inevitable. Increasing the dimensionality to a moderate value of eight dimensions results in a levelling of the rate at which the correspondences become divided. Continuing this trend, using higher dimensional data is seen to increase the number of separations that occur in the shallowest branches of these trees. Figure 2.6b plots the same data as a cumulative distribution in order to show the expected fraction of separated correspondences in at tree of a given dimensionality and depth. Here we can clearly see that the separation of correspondences will largely occur in the deepest layers of a low dimensional k-d tree and in the shallowest layers of a high dimensional tree.

Figure 2.7 presents an evaluation of these same properties with a dataset of ten thousand SIFT feature vectors. This analysis is performed on the extrinsic axes of the data as well as with the eigenvectors produced by PCA. In Figure 2.7a we see that there is little difference between the extrinsic axes and the rotated PCA axes. The most notable difference, is that the first segmentation with the PCA axes is able to make full use of the inherent structure that was seen in Figures 2.5c and 2.5d. Otherwise, the occurrence of separated correspondences is largely uniform across the layers of these trees. This is interesting considering that the SIFT data is represented across 128 dimension but the peaks that were seen with the random high dimensional data are much less pronounced here. Additionally there is some indication of a second peak of separations occurring at the lower levels of these trees as previously seen for the low dimensional random data. Lastly, Figure 2.7b shows the cumulative distributions for these datasets. Here we can see that the advantage gained by the first PCA segmentation makes little difference to the full distribution.

Ultimately, partitioning a high dimensional space with hyperplanes can be a computationally efficient way to categorise data into a hierarchical structure. However a number of problems arise when the dimensionality is sufficiently large. In general, we see that it is typically impossible to find a hyperplane that can linearly separate high dimensional samples into two meaningful partitions without breaking a number of the strongest correspondences in the data. Furthermore, the computational efficiency of these partitions comes from taking information along a single axis when deciding where to partition the data. This contributes to the undesirable separation

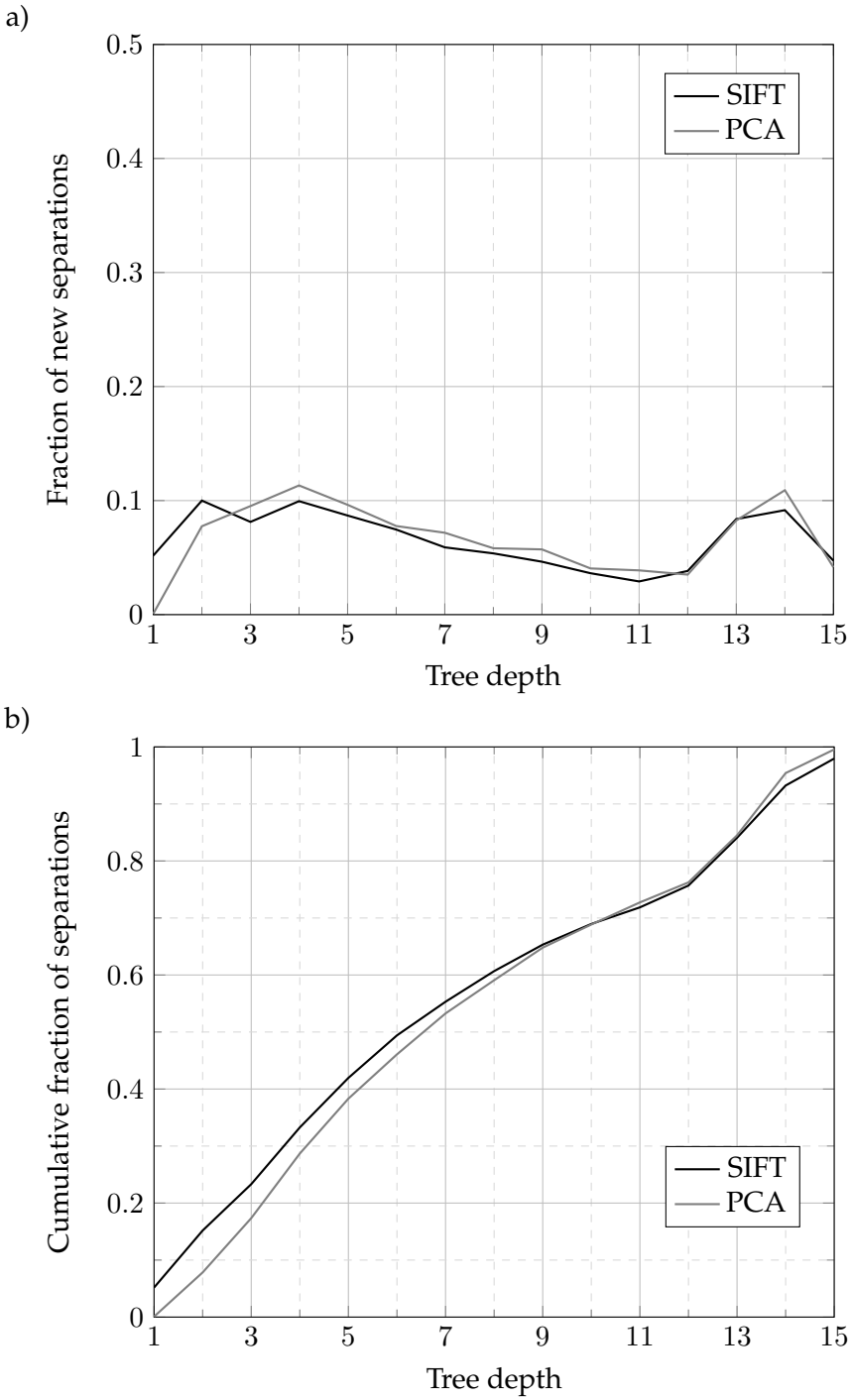


FIGURE 2.7: The distribution of separated correspondences in k-d trees built with SIFT data. a) The fraction of correspondences that have become separated at each depth in the trees. b) The cumulative sum of separated correspondences at each depth.

of samples that are in fact closely located when considering the full extrinsic space. Lastly, because a hyperplane in m dimensions is itself an $m-1$ dimensional space, the boundary created by the hyperplane is far from local. In order to retaining information about local correspondences, it follows that the segmentation method would benefit from being restricted to within local regions.

2.1.2 Clusterings

In contrast to the unbounded divisions of hyperplane boundaries, clustering is specifically designed to form segments from local neighbourhoods. These segmentations are defined by local boundaries that exist between adjacent clusters. Distance functions naturally give rise to clusters that are spherical in shape, since a pairwise distance between two samples can be equated to the radius of a hypersphere centred on one sample and with the other sample on its surface. We define an m-ball as an m dimensional Euclidean subspace that covers all samples of the form

$$(\mathbf{x}-\mathbf{c}).(\mathbf{x}-\mathbf{c}) \leq r^2 \tag{2.5}$$

where $\mathbf{c}$ is a vector to the centre of the ball and r is the desired radius. Similarly, we can define the $m-1$ dimensional hypersphere subspace that bounds the cluster as

$$(\mathbf{x}-\mathbf{c}).(\mathbf{x}-\mathbf{c}) = r^2. \tag{2.6}$$

When considering spherical regions in high dimensional space, we can firstly revisit the central region in Figure 2.2a that became devoid of samples as the extrinsic dimensionality was increased. Figure 2.8 plots the volume of a unit m-ball as well as the area of its $m-1$ dimensional surface. At two and three dimensions we can see the familiar relationships that are observable for circles and 3-spheres. For these low dimensional spaces, an increase in dimensionality results an increasing volume and surface area. However, this trend does not continue for very long, with both the volume and surface area quickly reaching a maximum value and then decaying back towards zero. While this behaviour is very unintuitive, it can be understood by carefully considering what the terms volume and area mean in this context. As the

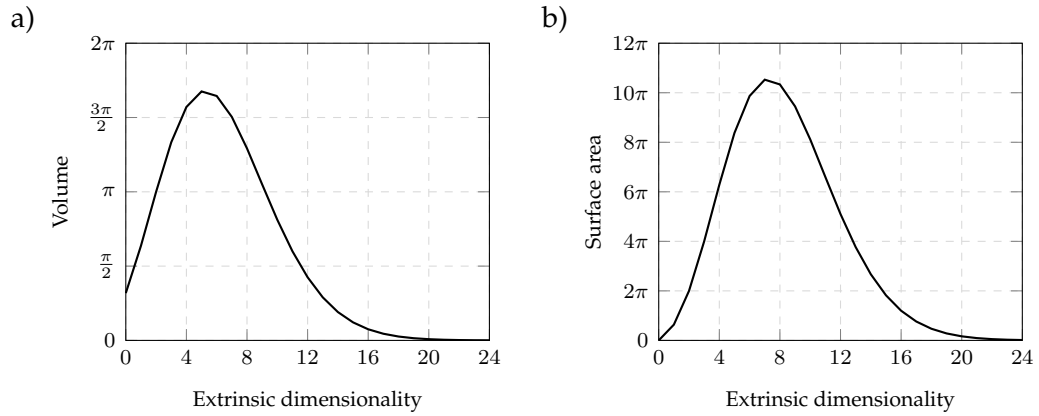


FIGURE 2.8: a) The volume of a unit m-ball reaches a maximum at around five dimensions before converging back towards zero. b) The same behaviour is seen for the unit $m-1$ dimensional hypersphere on the surface of the m-ball and with a corresponding maximum area at around seven dimensions.

number of extrinsic dimensions increases the concept of volume (and area) increases with an exponential growth rate. With each successive dimension the amount of extra volume being added to the entire space continues to grow. This cumulative growth in volume quickly dwarfs the successive amounts of volume being added to the m-ball. As such, we find that for increasing dimensionality a fixed-radius hypersphere will eventually start to shrink relative to an enclosing m-cube. This concept adds further justification for why corner regions are the most probable location for random high dimensional samples to occur.

When segmenting a high dimensional space with spherical clusters, it is useful to have an understanding of how many clusters are needed to fill the space. The kissing number is one of many useful measures of how hyperspheres can fill or cover a space. Specifically, in m dimensions the kissing number aims to maximise the number of non-overlapping m-balls of unit radius, while also positioning them such that each of these balls touches the surface of another m-ball that is positioned at the origin of the space and is also of unit radius. In terms of clustering data, the kissing number problem provides an upper bound on the number of adjacent clusters that can exist around any particular cluster in a given dimensionality. Figure 2.9 plots the current known bounds of the kissing number in up to 24 dimensions. We see that the kissing number has roughly exponential growth for a linear increase in

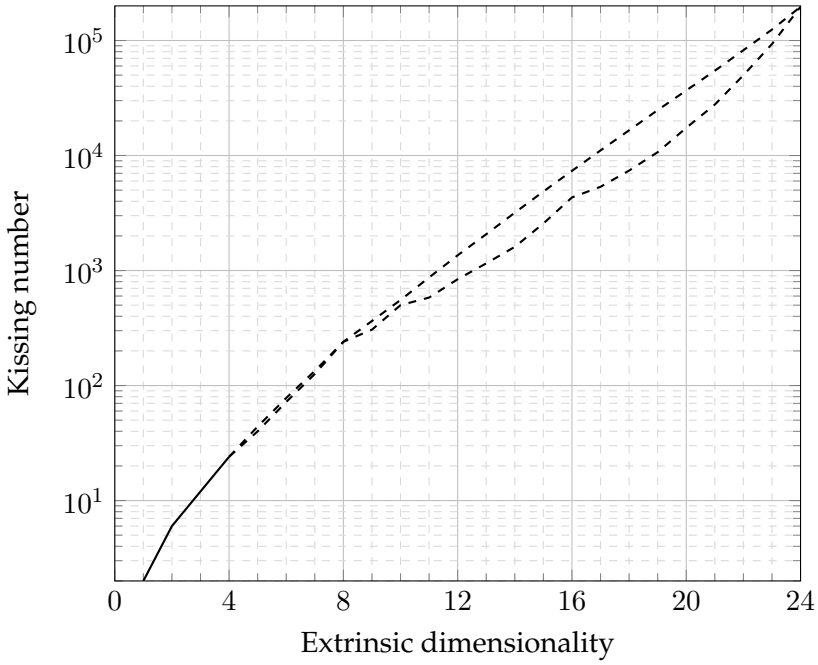


FIGURE 2.9: Current known bounds on the exponential growth of the kissing number relative to the extrinsic dimensionality of the space being filled. [19]

the number of extrinsic dimensions. This allows for a cluster to have thousands of adjacent regions in as few as twelve dimensions. Because a huge amount of data is necessary to fill a high dimensional structure, we expect that high dimensional datasets are unlikely to form structures that are more complex than a lone central cluster surrounded by a single layer of other clusters. Adding additional complexity to this structure can require exponentially more clusters, which then results in too few samples per cluster. Ultimately, high dimensional clustering will naturally result in a large number of adjacent clusters. Because each adjacency represents a local boundary, more adjacencies means a higher potential for corresponding samples to become separated into different clusters.

Following our general aim for data segmentation, meaningful clusters should aim to avoid the separation of strong correspondences within the high dimensional data that they segment. The spherical clusters we are considering are well suited for datasets that contain many small and dense regions. One such region is shown in Figure 2.10a, and is the result of combining two uniform random variables r and θ

in the form

$$x_1 = r\cos\theta \tag{2.7}$$

and

$$x_2 = r\sin\theta. \tag{2.8}$$

However, we are interested in levering the computational efficiency of working directly with the extrinsic dimensions of $\mathbf{x}$ rather than the parameters r and θ. In this case, we arrive at the distribution shown in Figure 2.10b, which is achieved over a disk by replacing r with $\sqrt{r}$ in Equations 2.7 and 2.8. Clustering this data is now more likely to separate closely corresponding samples, although this also indicates that the data is fully utilising the capacity of the space.

When considering the mapping of image data into a feature space, it can be desirable to enforce a structure that sacrifices some amount of capacity for feature representation in order to better distinguish between corresponding and non-corresponding feature vectors. In terms of data clustering, one method for achieving many dense and spherical regions is by sampling from a number of Gaussian distributions of the form

$$Pr(\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\mathbf{x}-\boldsymbol{\mu})^2}{2\sigma^2}}. \tag{2.9}$$

Here, $\boldsymbol{\mu}$ is the centre of a particular region and σ governs the size of the region. This distribution can be seen in Figure 2.10c, and was also present in the projected SIFT

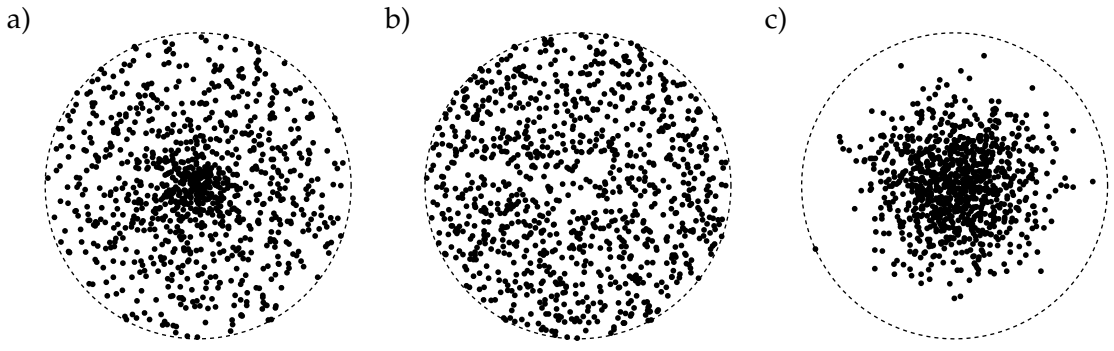


FIGURE 2.10: Random distributions within a circular region. a) A uniform sampling of the intrinsic parameters of a disc. b) A uniform sampling of the extrinsic parameters. c) Sampling from a Gaussian distribution.

data from Figure 2.5c. While the SIFT data does contain two large Gaussian regions, this small number of clusters does not allow for meaningful segmentation of the entire dataset. Local regions within each cluster will still appear as a relatively uniform distribution of samples. Again, we could consider this to be an unfavourable property as both memory and computation will be wasted on at least one extrinsic dimension that is not being fully utilised. In general, the ability to utilise local information when creating high dimensional clusterings is consistent with our aim of retaining strong correspondences. However, datasets are typically distributed so that at least some of the desirable correspondences will end up separated by the boundaries between adjacent clusters.

2.2 The Blessing of Dimensionality

The 'curse of dimensionality' has become a common expression for describing the difficulties faced when working with high dimensional data. For instance, if we consider searching for the optimal value of a single floating point variable by dividing the domain of that variable into equal tenths, we can then test each of the ten values in order to find the best assignment. This coarse discretisation may provide a reasonable result, but a finer division could still be used to find a more accurate solution. In this case, performing twice as many evaluations will double the resolution we have for determining the optimum. Then if we start to expand our optimisation to include additional independent variables, we quickly gain an appreciation for the 'curse' label. That is, when considering multiple independent variables, we must look at all possible combinations of those values. So when we increase the dimensionality of our search, the number of evaluations grows exponentially. If we were to attempt the coarse parameter sweep over just ten or twenty independent variables, the number of evaluations quickly becomes infeasible.

Many real-world measurements are in fact taken from continuous domains such as the frequency of a photon or the changing angle of a bird's wing as it flies. Sets of visual data are typically collected as a subsampling of these high dimensional

continuous spaces, and then further mapped down as local image regions or visual features. However, the 'curse of dimensionality' states that a linear growth in the number of extrinsic dimensions will cause an exponential increase in the number of samples that can be represented in the full continuous space. By definition, starting with a subsampling will always restrict us from using the full continuous space. This property of real-world datasets has then given rise to the term 'blessing of dimensionality'. The 'blessing' indicates that when working with sufficiently high dimensional data it will typically be impossible to have enough data to fill the space. Additionally it has been observed that most real-world visual data is inherently low dimensional [20], as there are very few systems complex enough to require many independent variables for describing their behaviour within an acceptable margin of error.

If we consider that most datasets are inherently governed by only a few independent variables, then it is reasonable to expect that we can rearrange the data in a way that lets us discard any unnecessary dimensions. To explore this idea we can use the PCA data from Figure 2.5 to provide a measure of how much total variance would be lost by discarding an extrinsic dimension of the SIFT dataset. Figure 2.11 first plots the fraction of the total variance that can be retained when repeatedly removing the extrinsic dimension with the lowest variance. This process was then repeated using the eigenvectors from the PCA results. The extrinsic curve shows an almost linear reduction in the total amount of variance lost with the removal of each dimension. This indicates that each dimension does in fact account for a roughly even portion of the total variance. However, the linear transformation into the eigenvector space reveals that a more biased distribution can be achieved. With this rotation of the axes, it is possible to discard slightly more than half of the dimensions, while still retaining 90% of the total variance. Once the data has been reduced down to ten dimensions, we find that about half of the total variance is still present. While these values are better than those of the extrinsic dimensions, we might expect that data generated from only a few independent variables can be reduced without any loss of variance. To approach this kind of result, we would likely need a computationally expensive non-linear method of dimension reduction. These methods aim to detect

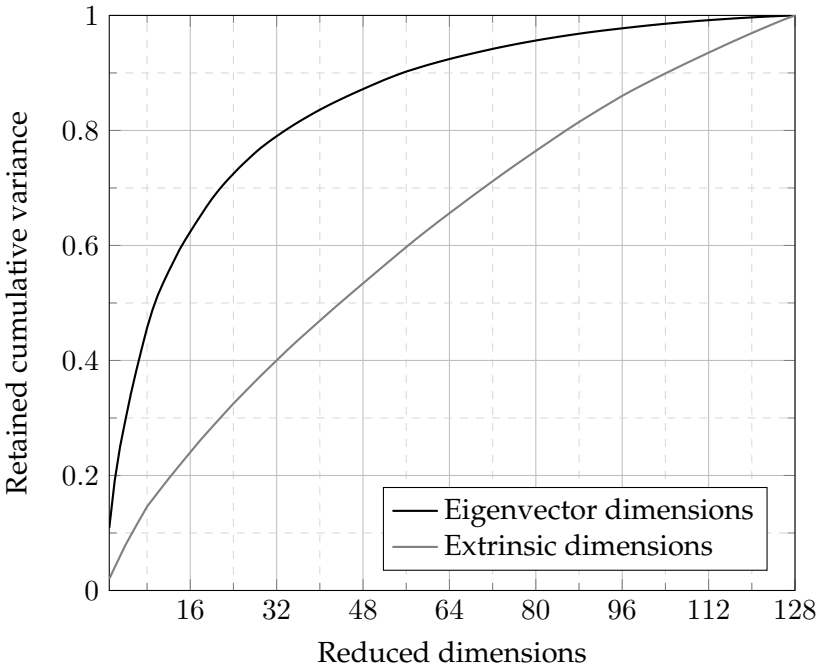


FIGURE 2.11: Retained cumulative variance when removing extrinsic dimensions from SIFT data. More variance is retained when the axes are rotated to align with PCA eigenvectors.

and parametrise lower dimensional structures that can only be fully represented in a higher dimensional space. If we consider this in a more familiar space, we can appreciate that it is impossible to take samples from a circle and then place them along a single linear axis without either overlapping two non-adjacent samples or separating two samples that were originally adjacent. This is because the surface of an $(m-1)$ dimensional hypersphere can only be defined in at least m dimensions.

2.3 Dimensionality Analysis

For any dataset that measures a fixed number of independent values (the extrinsic dimensions), we can loosely define the intrinsic dimensionality as a measure of how effectively this data fills the full extrinsic space. We would like to use such a measure in order to explain our observations from Figures 2.6 and 2.7 where we saw 128 dimensional SIFT features expressing properties of a random data with a much lower dimensionality. To illustrate the concept of a non-integer dimensionality, we will briefly discuss the well studied coastline paradox. This paradox arises

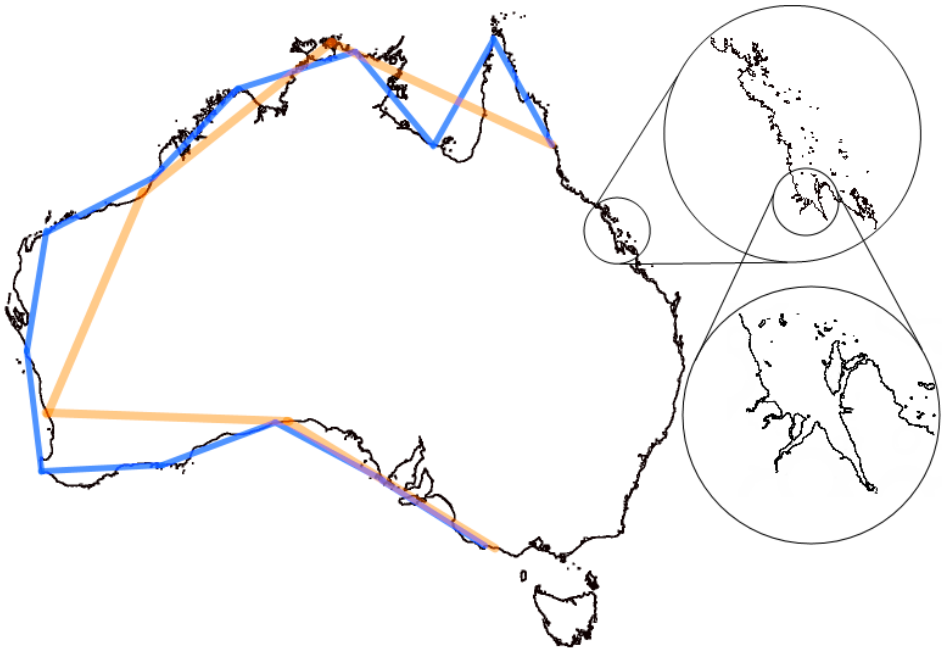


FIGURE 2.12: Natural coastlines exhibit fractal properties when their length is measured at multiple scales. Viewing coastlines at a smaller scale will unveil additional complexity that was not previously measurable.

from the unintuitive observation that there is no well defined way to measure the length of a geographical coastline. Lacking a well defined length is then closely related to all coastlines having an intrinsic dimensionality of between one dimension (a straight line) and two dimensions (the surface of our planet). These properties can be explained by the fractal structure that is observable in all coastlines [21]. On the right-hand-side of Figure 2.12 we can see that viewing the same coastline across different scales uncovers additional complexity that was not apparent at larger scales. To arrive at a resolution for the paradox, we can first represent the coastline as a series of one dimensional segments on a two dimensional surface. If we use a large one dimensional ruler that can take a single measurement across the country, then the dimensionality is 1, and our measured coast length is most likely an underestimate. To get a more accurate measure we can use lots of measurements with a smaller ruler. So the length of the coastline is then seen to be dependant on the length of our ruler, with different rulers giving us difference coast lengths. Intrinsic dimensionality can then provide a single measurable quantity by observing the relationship between the length of different rulers and the length of coast that each

one measures. For instance, if we were to halve the length of our ruler we could expect to double the number of times we can place that ruler along a straight section of coastline. However, for most sections of coastline we will find that we can now place the ruler more than double the number of times, indicating a dimensionality that is larger than 1. Two such measurements are illustrated on the left-hand-side of Figure 2.12, where the shorter ruler reveals more complexity than we can measure at the larger scale.

There are in fact many different measures of intrinsic dimensionality, and we have chosen to use the Hausdorff dimensionally within our research. This is because we can formulate the Hausdorff dimensionality in terms of the pairwise distances between all samples in a particular dataset. As such, our Hausdorff dimensionality calculations are closely related to the data correspondences we have been discussing. Additionally, computing dimensionality in this way can then leverage the same optimisations that we might apply when processing correspondences. In this section we define Hausdorff dimensionality and show how it can be used for measuring the space filling capacity of a dataset.

2.3.1 Hausdorff Dimensionality

For a given dataset $\mathcal{X}$ that contains a large number of vectors, each with m extrinsic dimensions, the Hausdorff dimensionality is given by

$$\dim_H(\mathcal{X}, r) = r\frac{\rho(X, r)}{N(\mathcal{X}, r)}. \tag{2.10}$$

Here r is the radius of many m-balls that are positioned to form a complete covering of the data, ρ is the measured density of the covering and N is a sum of relevant distances which we simply define as

$$N(\mathcal{X}, r) = \sum_{\mathbf{x}_{i,j} \in \mathcal{X}} \begin{matrix} 0 \text{ if } dist(\mathbf{x}_i, \mathbf{x}_j) \geq r \\ 1 \text{ if } dist(\mathbf{x}_i, \mathbf{x}_j) < r \end{matrix}. \tag{2.11}$$

In order to eliminate the need for a direct density measurement, we can express the Hausdorff dimensionality in terms of the change in density between two different

radii with

$$\dim_H(\mathcal{X}, r_1, r_2) = \frac{\log\left(\frac{N(\mathcal{X},r_1)}{N(\mathcal{X},r_2)}\right)}{\log\left(\frac{r_1}{r_2}\right)}. \tag{2.12}$$

A similar concept was seen in Figure 2.12 with the coastline being measured using two rulers of different lengths. We can further simplify our formulation setting $r_2 = 2 \cdot r_1$ so that

$$\dim_H(\mathcal{X}, r) = log_2 \frac{N(\mathcal{X}, 2r)}{N(\mathcal{X}, r)}. \tag{2.13}$$

2.3.2 Computing Dimensionality on a GPU

For a dataset containing n samples with m extrinsic dimensions the computational complexity of computing the Hausdorff dimensionality in Equation 2.13 is $O(n^2 m)$. This is due to the double summation around the distance functions in Equation 2.11. Since the dimensionality calculation is dominated by the cost of computing the distances, this formulation presents a considerable opportunity for optimisation with parallel computation of these distances. A naive implementation might iterate over each pair of samples in the dataset and compute the distance between them. For the SIFT dataset with $n = 10^6$ and $m = 128$, the number of floating point operations required to compute all distances will be

$$2n.\frac{n(n-1)}{2} \approx 1.3 \times 10^{15}.$$

Modern GPUs can roughly achieve teraFLOPS and modern computing clusters are reaching into the petaFLOPS, so these calculations should be achievable with current hardware. However, we must also consider that same number of memory reads are required in order to process all of the data. Due to limited amounts of fast memory, this overhead could be detrimental to the runtime. If a reduced runtime is to be achieved through parallelisation, then the implementation must also avoid the large overheads caused by frequently accessing slow memory.

Firstly, our implementation minimises the memory overhead by making full use of the intermediate memory that is shared within each GPU block. A single block is assigned to calculate all distances between two contiguous subsets of the full dataset. The size of these subsets is chosen as the maximum size that will allow for both

subsets to fit within the shared memory of a single block. As such, computing a block of distances involves two stages; the first is to have all available threads loading the two required subsets from global memory into the shared memory, and the second is to then allocate threads to compute the distances between pairs of data in the shared memory. Because Equation 2.11 does not use any strict equalities, we can utilise the monotonic property of squaring real values in order to avoid computing the square-root in Equation 2.1.

In Equation 2.11, the computation of the distances is independent of the radius parameter r. As such, once a set of distance calculations has been performed it is advantageous to immediately compute multiple N values using each desired radius. A reduce operation is then applied to collect and sum the partial N values that are being calculated within each block. As each partial value is computed in parallel, the results are combined into a single global value using atomic functions. The reduce operation for each different radius will only require a small amount of memory relative to the representation of the data vectors. It is important to only use a small number of different radius values so that the distance calculations can make full use of the shared memory. By making full use of the parallel capabilities offered by GPU computing, we are able to compute the Hausdorff dimensionality of datasets containing up to hundreds of millions of high dimensional vectors.

2.3.3 Estimating the Dimensionality of Fractals

Many fractals are infinitely repeating self-similar structures that have a precise Hausdorff dimensionality. We can utilise the exact self-similarity of these fractals in order to validate the accuracy of our formulation of the Hausdorff dimensionality with dataset containing varying numbers of samples. Specifically, we are interested in demonstrating an accurate measure of intrinsic dimensionality for datasets that are a random subsampling of a particular space. The Serpinski triangle is a well-studied fractal with exact self-similarity. Due to its triangular structure, scale changes in the Serpinski triangle produce sets of three identical copies of the fractal. Because this self-similar structure occurs each time the extrinsic measurements are doubled, it is trivial to derive the exact Hausdorff dimensionality of $\log_2 3$ from Equation 2.13.

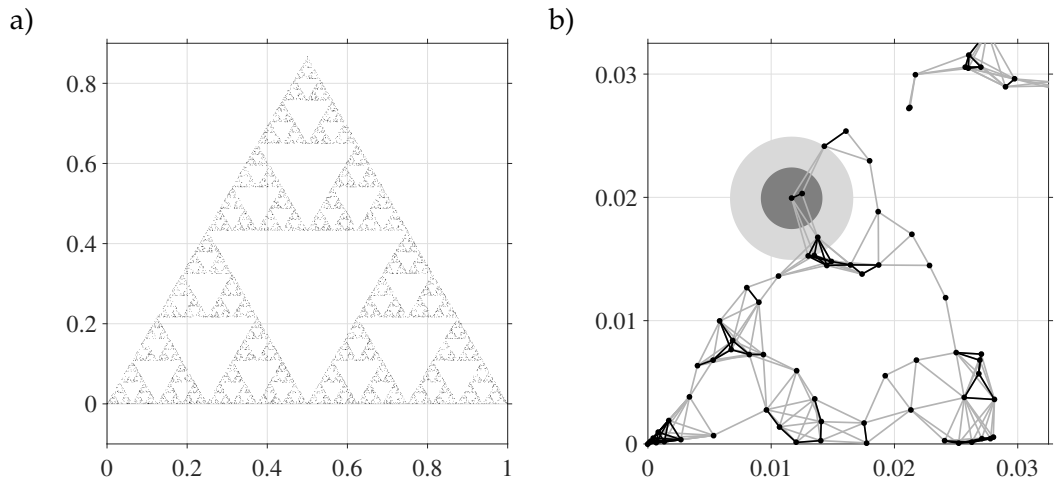


FIGURE 2.13: A discrete sampling of the Serpinski triangle. a) The complete set of random samples is shown at the largest scale. b) An example of relevant pairwise distances when computing the Hausdorff dimensionality across two fine-grained scales.

In order to apply the discrete approximation given by Equation 2.11, we generated a dataset of random samples from the surface of the fractal. This was achieved by initially defining three corner samples at the largest scale. Subsequent samples can then be drawn from increasingly smaller scales by computing the mid-point between two existing samples, chosen uniformly at random. In order for each sample to remain on the Serpinski triangle, at least one of the two existing samples must always be drawn from the initial three corners. Figure 2.13a plots these random samples on a two dimensional plane. The expected intrinsic dimensionality of around 1.585 allows the data to partially fill the planar surface. Figure 2.13b illustrated the fine scaled measurements of our intrinsic measure over a subset of the samples. Two disks are shown as an example of radii that could be used in Equation 2.13. At each radius, pairs of samples are shown connected together if they would contribute to the summation in Equation 2.11.

Figure 2.14 plots the results of our intrinsic measure against the exact Hausdorff dimensionality of the Serpinski triangle. Each curve is constructed by applying Equation 2.13 across a wide range of radii. This process was repeated for increasingly large sets of random samples. The results show that our intrinsic measure becomes more accurate with the inclusion of these additional samples. This is particularly

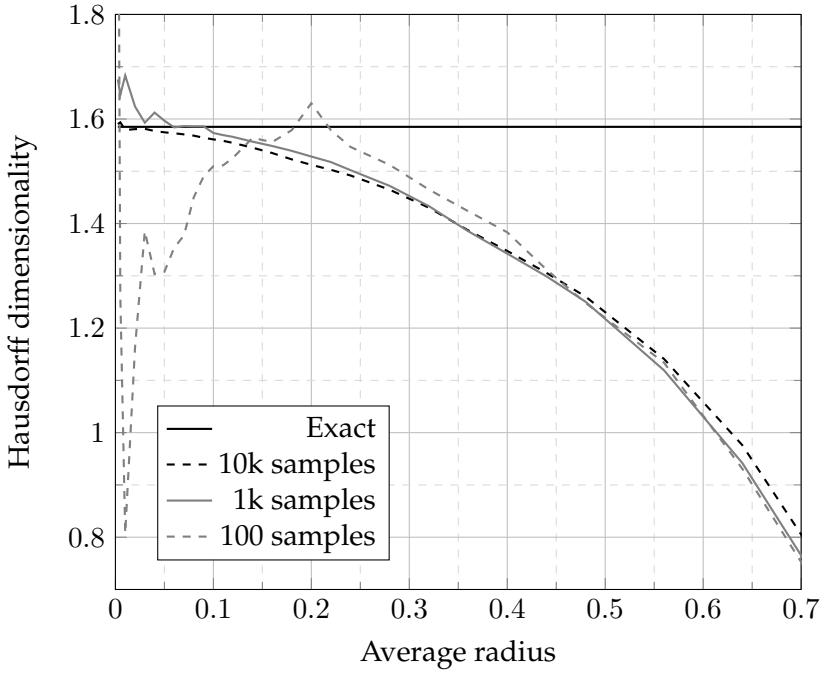


FIGURE 2.14: An approximation of Hausdorff dimensionality computed across a wide range of scales for dataset of increasing size. Having too few samples at a smaller scales is shown to impact the accuracy of the measure.

pronounced at smaller scales, which is also where our estimate of true dimensionality approaches the ground truth value. When the sampling is too coarse, there is not enough information to observe the self-similar structure that is repeating at increasingly smaller scales.

2.3.4 Estimating the Dimensionality of Visual Data

The SIFT datasets from Table 1.2 were selected for intrinsic dimensionality analysis due to the widespread use of this feature type for the past few decades. With an extrinsic dimensionality of 128, the SIFT feature space also offers a large potential for complex high dimensional structures. Analysis was performed on increasingly large subsets of the samples in order to gauge the requirements for adequately filling the feature space. Similar to the results for fractal data, we expect that a large number of samples will be needed before fine-grained variations are visible in the data. Importantly, we are unable to compute an exact Hausdorff dimensionality for datasets of real-world feature vectors. While there is an upper bound on the number

of SIFT feature vectors that can be represented, it is quite possible that not all of these feature vectors can be generated from data found in natural images. Furthermore, a practically sized dataset can only ever contain a small subset of all possible feature vectors. Even with the optimisations we have implemented, we are still artificially limiting the size of our datasets in order to keep the runtime feasible.

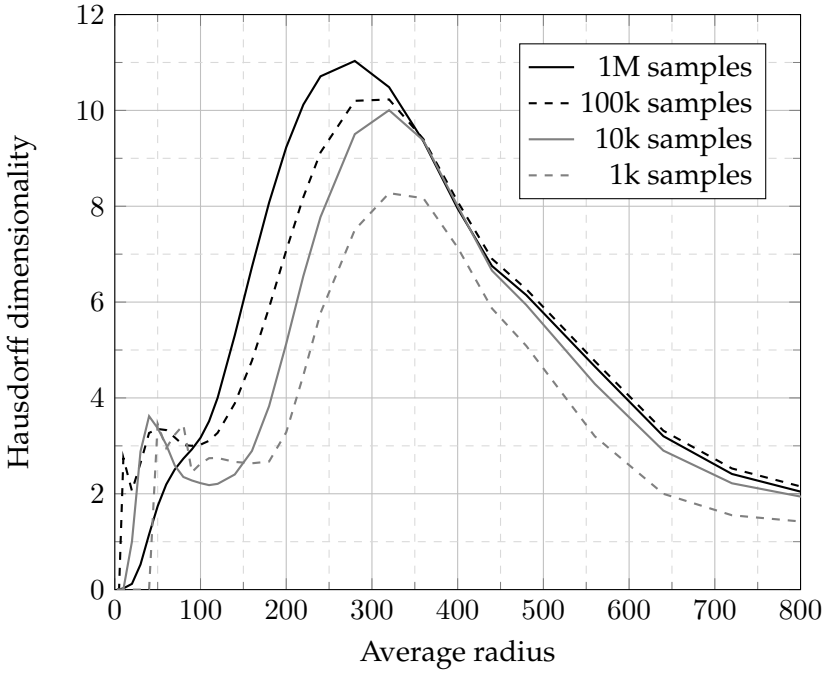


FIGURE 2.15: Intrinsic dimensionality of the SIFT feature space is estimated with an increasingly large number of samples. The results show that larger datasets result in additional fine-grained structures that give rise to higher dimensionality.

Figure 2.15 plots the results of our intrinsic measurements for increasingly large datasets of SIFT feature vectors. The key finding of these results is that the intrinsic dimensionality of the SIFT space is considerably lower than the extrinsic dimensionality that is used to represent the feature vectors. Specifically, we find that moderately sized datasets of SIFT feature vectors appear to occupy an embedded manifold of around ten dimensions within the 128 dimensional space that is available to them. While this could indicate that over 90% of the data can be removed from each feature vector without any loss of information, we are in fact observing non-linear dimensions that can only be fully represented in the complete higher dimensional space. The large amount of utilised feature space is an essential by-product of representing the complex correspondences between all pairs of samples within a dataset. Ultimately,

we expect to see large gains in the efficiency of computer vision algorithms that can operate on the small number of intrinsic dimensions of a dataset, rather than processing the full extrinsic space. However, efficient usage of these lower dimensional structures will also need to avoid the costly process of directly computing the full parameters of this intrinsic space.

Chapter 3

ANN Search Techniques

In the Chapter 2 we introduced the concept of high dimensional image spaces and presented a number of inherent properties that contribute to the challenges of efficiently processing datasets within these spaces. Despite these challenges, large datasets of high dimensional feature vectors are a central component of a broad range of computer vision tasks. This type of data is utilised in a variety of areas such as object and scene recognition [22], pose estimation [23], [24], relocalisation and loop-closing [25], 3D reconstruction [26] and machine learning [27]. Fundamental to each of these applications is a reliance on associating visually similar data in order to form correspondences between different images. Each correspondence amounts to a small piece of information that when pooled together can give rise to the high level information that is required for succeeding at these complex computer vision tasks.

Finding correspondences in large and high dimension datasets is computationally demanding and can become infeasible when there is a requirement for real-time responses, or when a large number of these matches are required. When a new feature is presented to a computer vision system, the fundamental goal is to compare this query vector $\mathbf{q} \in \mathbb{R}^m$ against the set of sampled feature vectors $\mathcal{X} \in \mathbb{R}^{n \times m}$ that are already know to the system and to identify a set $\mathcal{S} \subseteq \mathcal{X}$ containing samples that correspond strongly with the query. This problem is widely referred to as nearest neighbour search, where the elements of $\mathcal{S}$ are considered to be the neighbours of $\mathbf{q}$. Strong correspondences are usually defined as either samples that are within a fixed

distance r of $\mathbf{q}$

$$\mathcal{S} = rNN(\mathbf{q}, \mathcal{X}, r) = \{\mathbf{s} \in \mathcal{S} \subseteq \mathcal{X} \mid dist(\mathbf{q}, \mathbf{s}) < r\} \tag{3.1}$$

or as the k samples that are closest to the query

$$\mathcal{S} = kNN(\mathbf{q}, \mathcal{X}, k) = \{\mathbf{x}_i \in \mathcal{S} \subseteq \mathcal{X},\ \mathbf{x}_j \in \mathcal{X} \setminus \mathcal{S} \mid |\mathcal{S}| = k \wedge dist(\mathbf{q}, \mathbf{x}_i) \leq dist(\mathbf{q}, \mathbf{x}_j)\}. \tag{3.2}$$

In the first case, Equation 3.1 will produce nearest neighbour sets of varying size based on the density of samples around a particular query. In contrast, Equation 3.2 will always produce the set of k-nearest neighbours that are bounded by a variable radius from each query, which is again dependant on the local density of samples around that query. From a practical standpoint, the k-nearest neighbour set is typically favoured as it accommodates for local variations in density that could result in a fixed radius covering all or none of the samples in a dataset. Additionally, guaranteeing a fixed number of neighbours places a clear bound on the memory complexity for storing the neighbourhoods even if this means that some of the stored neighbours are weaker correspondences.

A naive solution for constructing nearest neighbour sets is to linearly search all samples in the known dataset and evaluate each one as a possible correspondence for the query feature. Unfortunately this solution is suitable only for trivially small datasets or for situations where response time is not important. In general, there is no known algorithm that is guaranteed to find all relevant correspondences for high dimensional data in a sublinear time. But for many real-world applications there are no explicit requirements for having access to every single low level correspondence. In fact, the contents of feature vectors and the formation of feature spaces are already heuristic in nature. Hence, it can become common practice to perform an approximate search that offers less accurate results but with a sublinear computational complexity. To achieve a sublinear complexity, the approximate nearest neighbour (ANN) search must be completed without the query being directly compared to every sample in the dataset. As such, it is common for an indexing structure

to be computed over the dataset as a means of predetermining how certain comparisons can be used to exclude the need for making other particular comparisons. For the applications we consider, the construction of the index is an offline process on a fixed dataset. Once the index is built it can be used to accelerate the search for ANN sets of an arbitrary number of previously unseen queries.

The lack of a sublinear exhaustive search algorithm means that verifying the completeness of a nearest neighbour set will sacrifice the efficiency gains of an ANN method. Instead, ANN searches are usually allocated a fixed amount of time or compute. Once these resources have been expended, the resulting nearest neighbour set is produced from the best comparisons that were made during the search. For a particular index of a particular dataset, it is possible to compute the amount of resources needed to achieve a particular average rate of recall. Here we define recall as the fraction of the nearest neighbour set that was correctly returned by an ANN search. The remainder of this chapter looks at the strengths and weaknesses of common ANN methodologies. These methods have been broadly divided into four main categories based on their respective indexing schemes. The evaluations we make are largely based on the characteristics of high dimensional data that we discussed in Chapter 2 and how this understanding can be used to achieve a more efficient ANN search.

3.1 Hashing Functions

Hash functions provide a computationally efficient many-to-one mapping that can reduce high dimensional data to a lower dimensional hash of that data. As mentioned in Section 2.2, performing a large dimension reduction with a computationally efficient mapping is likely to result in loss of information. In the context of the ANN search problem it is important that at least some information is retained about the local neighbourhoods that exist in the unreduced data. Gionis et al. [28] demonstrate that locality-sensitive hashing (LSH) is a suitable option for retaining key information for performing ANN search on high dimensional image data. Unlike cryptographic hashing methods, LSH has a high probability of mapping similar

input vectors to the same location in the hash space. Gionis et al. found that nearest neighbour recall is greatly improved when multiple independent hash tables are applied concurrently to the unreduced data. With more hash tables, it becomes more likely that a query vector will have at least one hash that is close to a hash of its nearest neighbour. Because the search is performed in a reduced space, an accurate list of neighbours can be efficiently obtained by generating a short candidate list in the hash space and then recomputing the distance to each candidate in the unreduced vector space.

Due to the importance of selecting discriminative hash functions, Shakhnarovich et al. [23] achieve further improvements to ANN performance by utilising a data-driven search for effective hash functions. Since the goal of LSH is for the hashes of similar objects to be coincident, the effectiveness of a hash function can be determined by its ability to generate these collisions. As with all data-driven ANN methods, it is assumed that the dataset being indexed represents an unbiased distribution of the query vectors that will be presented during the search phase. Shakhnarovich et al. apply a distance threshold to classify pairs of vectors from a known dataset as either neighbours or non-neighbours. Using these classifications, hash functions can then be constructed to maximise the number of collisions between neighbouring pairs of vectors while minimising the number of collisions between non-neighbouring pairs. For practical applications, the thresholds will be small enough that only a few neighbours are considered for each sample. As such, the number of neighbouring pairs is dwarfed by the number of non-neighbouring pairs. Ultimately this means that the hash functions are trying to avoid mapping non-local regions to the same location in the hash space and are unlikely to retain the local structures of these regions from the unreduced vector space.

In general, LSH methods are best suited for a fixed-radius ANN search as described by Equation 3.1. Andoni and Indyk [29] explore the relationship between pairwise distances in the unreduced vector space and the probability of generating collisions between these samples in a particular hash space. By increasing the sensitivity of a hash function with longer hash codes, a steep gradient can be achieved such that a small reduction in pairwise distance will correspond to a large increase in the chance

of a collision. The distance at which this gradient occurs can then be considered as a reliable radius for ANN search and is adjusted with the number of hash functions being used. When this radius is too small many query vectors will have very few or perhaps no neighbours within the predetermined global radius. In contrast, if the radius is too large then the computational efficiency of the search will decrease as the number of collisions increases. This loss of efficiency comes from a much larger linear search in the unreduced vector space for determining the true ranking of the proposed neighbours. In some cases, such as when global density of data is non-uniform, bounds on the search radius will necessitate the use of an indexing method that can return k-nearest neighbour results across the entire dataset to allow for a more uniform re-ranking cost.

While more discriminative searches can be performed with higher dimensional hash codes, the additional code space can lead to larger hash tables. When coupled with the requirement to use many independent hash tables to achieve decent accuracy, resulting in a larger memory footprint for storing the index. Lv et al. [30] address these memory concerns by expanding upon the idea of hashing a query vector and then additionally comparing it to nearby hash codes in each of the tables. The underlying premise is that even if the hash of a query vector is not exactly coincident with that of its nearest neighbour, the properties of LSH should produce hashes that differ by only a small amount. For instance, using one hash function and matching all hashes that differ in just a single dimension can produce a larger set of nearest neighbour candidates than matching hashes exactly across many tables. In this way, Lv et al. are able to reduce the number of hash tables while still retaining a high probability of returning multiple neighbours for each query. As with other hashing methods, each of these additional neighbours will increase the size of the linear search for re-ranking neighbours in the unreduced vector space. Adjusting the threshold on how far to explore beyond coincident hash matches gives rise to a convenient trade-off between compute and recall. A tighter bound means less linear search and more chance of missing a good neighbour, while a relaxed bound will yield a larger search and should also provide better recall. Without searching further in each hash table, LSH methods are better described as trading off memory for recall when additional

tables are added to the index.

Many more recent hashing methods have been developed, each of which builds upon the core idea of defining data-driven hash functions that perform a dimension reduction while attempting to keep local neighbourhoods separate from each other. Improved function definitions have given rise to the incremental gains seen in spectral hashing[31], kernelised LSH[32], semi-supervised hashing[33], optimised kernel hashing[34], complimentary hashing[35], uniform bin size[36], spherical hashing[37], parallel Voronoi LSH[38] and optimal data-dependent hashing[39]. A key strength across all of these methods is the underlying simplicity that allows for thorough theoretical analysis of algorithmic complexities as well as guarantees on the expected search performance. In practice these hashing methods are still outperformed by other indexing schemes in many sub-domains of ANN search. Ultimately these hash functions are aiming to divide a high dimensional space into coarse bins while often making use of computationally efficient hyperplane boundaries. As discussed in Section 2.1.1 and 2.2 either the functions will be unable to accurately represent the pairwise relationships from the unreduced space, or the functions being used are complex enough that the preprocessing of each query has a significant impact on the search performance.

3.2 Decision Trees

Tree structures are a common way to partition a dataset into discrete regions at multiple scales. As such, early work such as Friedman et al. [40] and Bentley[41], [42] applied k-d trees to the nearest neighbour search problem in a moderate number of dimensions. Bentley showed that while this structure is effective at indexing low dimensional data, the performance of trees reduces quickly as dimensionality increases. At each of its levels, a k-d tree creates a binary partition of the data assigned to each node. The partition is achieved by computing the median value along an extrinsic axis and then dividing the data along that dimension. As seen in Figure 2.5, better trees are constructed by selecting high variance axes in favour of low variance ones. Regardless of the selection order, when higher dimensional hyperplane

divisions intersect the median of an axis they will separate many nearest neighbour pairs.

In general the propagation of a query from the top to the bottom of an indexing tree is computationally efficient. However the average recall rates that are achieved with a single propagation are very low. For this reason, backtracking algorithms are used to increase the recall to a useful range. The need for large amounts of backtracking is an inherent property that is tied to the branching structure of the trees. During the propagation of a query vector, each time a branch is taken the decision is based on a local threshold which represents only a small part of the information in full space. Whenever a query is close to a threshold value there is a significant probability that its nearest neighbour lies down a different branch than the one being taken. To improve the performance of high dimensional k-d trees Beis and Lowe[43] showed that a greedy search could improve search efficiency at the expense of having lower accuracy than the backtracking branch-and-bound search that performs well for low dimensional data. The greedy ANN search backtracks by maintaining a priority queue of nodes with unexplored children and is sorted based on the distance between the parent and the query vector. Search accuracy and computational cost are traded off by setting a maximum on the number of nodes that will be visited during a search. Furthermore, Silpa-Anan and Hartley[44] construct and search multiple k-d trees in parallel using a single priority queue. Similar to the process of principal component analysis (PCA), each tree is constructed with a different rotation of the data. Building multiple independent kd-trees greatly improves the recall rate for ANN search with high dimensional data, although there are diminishing returns for each additional tree. This approach has many similarities to the use of multiple independent hash tables for improving the accuracy of LSH methods.

In contrast to k-d trees that partition data along extrinsic axis, Sivic and Zisserman[45] cluster high dimensional visual data using intrinsic properties found with k-means clustering. Then once the clusters are finalised, the data can be indexed with an inverted file. This method was improved by Nister and Stewenius[24] with the inclusion of a hierarchical tree structure. At each node of the tree, the parameter k is both the number of clusters and the number of children. While there is a

greater computational cost associated with computing intrinsic properties, Nister and Stewenius show that effective clustering can be performed with a subset of the data. The computational cost of propagating a query through a k-means tree remains much higher than for k-d trees. For a k-means tree, each comparison involves all of the extrinsic axes and since internal nodes are not sampled from the dataset these comparisons are effectively wasted. In order to control the trade-off between computational cost and search accuracy, Schindler et al. [46] showed that a greedy search exploring n paths at each level of the tree is preferable to adjusting branching factors. Although they note that the branching factor does impact the number of vectors that end up close to the boundaries formed between two adjacent clusters. In the context of ANN search a query is more likely to travel down a different branch than its nearest neighbour whenever that neighbour is close to one of these boundaries.

Many other tree structures have been defined around alternative ways to partition a vector space or to cluster a dataset of vectors. The most notable of these are VP trees[47], SR trees[48], spill trees[49] and NV trees[50]. A number of these methods explore the use of spill regions as an alternative way to optimise a tree for ANN search. Whenever a sample is found to be too close to the boundary between two nodes the vector is allocated to both sides of the boundary. While spill regions have the negative consequence of creating a larger index, they help compensate for non-intrinsic boundaries and also yield better recall for greedy searches without a mechanism for backtracking up the tree. However, Lejsek et al. [51] find that searching multiple trees in parallel is a more memory efficient way to achieve these benefits when indexing very large datasets.

Recently Muja and Lowe[52] showed that for ANN searches on high dimensional visual data, an optimised k-d tree or k-means tree will out perform an optimised hashing method. Both of these implementations are included in the Fast Library for Approximate Nearest Neighbours (FLANN)[53] as well as an efficient multi-index LSH method. The inclusion of two different indexing trees is aimed at providing good options for both fast index building or high recall searches. With both methods, the early separation of correspondences that is seen in Figures 2.6 and 2.7 limit the

efficiency of using indexing trees with high dimensional data. Each time a decision is made to travel down a particular branch of the tree there is always a significant probability that the nearest neighbour to that query does not exist down that branch. This is true even when the decisions are being made optimally based on the local information at each fork in the tree.

3.3 Vector Quantisation

As with hash functions, vector quantisation is a compact coding method that performs a many-to-one mapping of high dimensional vectors onto a reduced code space. When applied to ANN search, the code books defined in quantisation methods are similar to hash tables in that they can be built to operate effectively within a particular fixed-radius. Quantisation differs from hashing in that it aims to cluster the data with a k-means approach. Rather than using the tree structure found in k-means trees, Jégou et al. [54] continue to index the code using an inverted file. Without the hierarchical structure of a tree, clustering large amounts of data can quickly become infeasible due to the need for a large k. Too few clusters will lead to a non-discriminative mapping, but using too many clusters increases the memory cost of storing cluster centres and also reduces the robustness needed for assigning queries to the same code as their neighbours.

To avoid the excessive memory costs of storing many cluster centres Jégou et al. [5], [6] utilise product quantisation. In this method cluster centres are generated by concatenating a number of sub-quantisations that are each built on only a subset of the extrinsic dimensions. The general principle is that large codebooks can be generated from a smaller number of parameters. Additionally, since the centre locations are already known, the data vectors can be compactly stored as a quantisation of the residual between the vector and its assigned centre. The inclusion of the residual quantisation also allows for a coarser quantisation to be used for the initial clustering.

Rather than directly addressing the robustness issue, Jégou et al. present an efficient way to reduce the computational cost of re-ranking a proposed set of neighbours.

While LSH codes aim to reduce collisions between non-neighbours, quantisation is a clustering method which allows for better preservation of distance ratios from in the unreduced space. Since the distance between cluster centres is larger than the quantisation error between a data vector and its assigned centre, distances can be efficiently approximated in the code space by ignoring some or all of this error. As such, re-ranking with code space distances reduces the cost of comparing an encoded query to all similar encodings found in the index. Comparing to more samples in the code space should increase the overall recall, even when considering the additional errors introduced by the distance approximations. For applications that do not require high recall, the unreduced dataset is then no longer required so an additional reduction of the memory cost can be achieved. This smaller index size is particularly useful for indexing very large datasets on GPU hardware.

Ge et al. [55], [56] demonstrate further improvements to robustness, by finding an optimal transformation between a dataset and the cluster regions from coarse quantisation. When the dataset is rotated to place data vectors closer to their assigned centres, it follows that quantisation error will be reduced across the dataset. Kalantidis and Avrithis [57] show that this process can also be applied at the finer level residual quantisation. These transformations follow on from the use of PCA for improving recall when searching k-d trees.

A number of alternative methods have been proposed for defining codebooks and accelerating re-ranking. Notable examples include Cartesian k-means[58], iterative quantisation[59], composite quantisation[60], sparse composite quantisation[61], additive quantisation[62] and supervised quantisation[63]. However, an additional factor to consider is the applicability for each quantisation method to be optimised for GPU acceleration. When including the time cost for re-ranking neighbours in the unreduced vector space, product quantisation achieves search speeds that are roughly two times faster than querying the tree structures used in FLANN. These results can be achieved across a large range of recall values, though each value requires additional parameter tuning and the construction of a new index.

3.4 Neighbourhood Graphs

Each of the indexing methods that have been discussed so far fail to fully account for the properties of high dimensional image data that are explored in Section 2.1. When many low dimensional boundaries are used for high dimensional data, it is almost guaranteed that a query and its closest neighbours will be assigned to different sides of at least one boundary. Every time a wrong assignment occurs, these structures use computationally expensive methods to correct for the error. Hashing and quantisation methods expend time on a linear search across some of the many local boundaries. For tree structures, branches are evaluated down to the leaf nodes before backtracking or restarts are used to explore adjacent leaves. Neighbourhood graphs reduce these costs by using local information to navigate across boundaries in a local neighbourhood. Jégou et al. [64] demonstrate this type of graph by improving the efficiency of their quantisation method. A graph is constructed from the union of many trees and is then used to index the learned codebook. Ideally these graphs are constructed so that the paths from higher up in a tree can be corrected closer to the leaf nodes. Additionally, it is important that the trees share many of their edges in order to avoid the computation and memory costs associated with having a high branching factor. Wang et al. [65], [66] utilise similar hybrid methods by combining both k-d trees and quantisation with a neighbourhood graph. Using the additional structure provided by a graph, the search path of a query is used to avoid the cost of a random restart.

Neighbourhood graphs for ANN search contain a vertex at each sample in the dataset being indexed. Directed edges then define neighbourhood relationships between these vertices. For any given vertex, the out-degree equates to the number of neighbours for that vertex, while the in-degree is the number of vertices that consider that vertex to be in their own neighbourhood. Once the edges of the graph have been constructed, a search can be performed by comparing a query vector to vertices in the graph. During the search, edges in the graph are usually traversed after comparing the query to each neighbouring vertex and finding the edge that bring us closest to the query. In order to avoid local minima, Navarro [67] uses Delaunay graphs which have the interesting property that nearest neighbours can always

be reached using a greedy search. This property is not unique to Delaunay graphs and can also be seen in a fully connected graph. For the nearest neighbour search problem, Navarro suggests that lower vertex out-degree will correspond to a lower average computational cost for these searches. However, as the extrinsic dimensionality of the data increases, the Delaunay graphs quickly increase their out-degree as they converge on becoming fully connected graphs. This increase is analogous to the rapid growth of the kissing number seen in Figure 2.9. Analysing this unfortunate behaviour, Clarkson [68] concludes that neighbourhood graphs for high dimensional data should aim to retain the useful properties of Delaunay graphs but must do this using far fewer edges.

In order to bound the number of edges in a high dimensional indexing graph, Hajebi et al. [69] apply the same greedy search algorithm but instead start from a random vertex in a k-nearest neighbour graph. These graphs are able to maintain efficient exploration costs by enforcing that the out-degree of each vertex in the graph is equal to k. This restriction removes any guarantee that the greedy search will find the nearest neighbour of each query. As such, this method addresses the goals of ANN search by trading off some amount of recall for the increased computational efficiency that comes with having a lower average vertex degree. Instead of using the Delaunay neighbourhood, each vertex in a k-nearest neighbour graph defines its neighbourhood as the k shortest edges that it can form to other vertices. While a small k does lead to greater search efficiency and lower memory costs, k must be large enough to ensure that the graph is connected. This requirement is equivalent to saying that k must be large enough to include every edge from the minimum spanning tree (MST) [70] of the vertices. Even with this constraint, the degree of these graphs can remain low even when indexing high dimensional data. While clustering [71] and nearest neighbour matching [72] can be achieved with just the MST, Hajebi et al. show that their k-nearest neighbour index greatly out performs hashing and tree based methods. The main drawback of this method is the difficulty of selecting an appropriate value for k such that a target recall rate can be achieved with the resulting index. Additionally, application areas that require frequent updates to the dataset will be hampered by the larger computational complexity for the offline

construction of the graphs.

To accelerate the construction of k-nearest neighbour graphs, Dong et al. [73] propose an algorithm for approximate construction. Since the results of the queries are already being approximated, any reduction in performance due to approximating the graph can be grouped with the existing performance deficiencies. The graph construction algorithm proposed by Dong et al. is based on the concept that for each vertex the neighbour of a direct neighbour is also likely to be a direct neighbour. First, the graph can be initialised with random directed edges that are treated like bidirectional edges during the construction process. Once initialised, pairs of vertices are repeatedly sampled for sharing their local neighbourhoods with each other. As this information is shared, the destination vertices of each edge should simultaneously converge towards their source vertex. This construction process is then terminated once the frequency of changes drops below a threshold. While it is unclear whether the low dimensional concept of "the neighbour of a neighbour is also my neighbour" still applies in for high dimensional data, Dong et al. demonstrate that their approximate graphs still achieve strong results. The main draw back of this construction method is the addition of two more tunable parameters, with the sampling rate and the termination threshold both needing to be tuned for particular applications. An alternative construction method is given by Wang et al. [74] who approximate k-nearest neighbour graphs by combining many overlapping subgraphs before propagating the local neighbourhoods. These subgraphs are formed by recursively bisecting the vertex set using hyperplane divisions until each subset is small enough for quickly building a k-nearest neighbour graph. As was seen with LSH and k-d trees, this process can be repeated to generate a small number of different bisections. The union of the small subgraphs from each repeat then provides a good initialisation for refining the edge lists. Most recently, Zhang et al. [75] have defined a similar process that uses LSH to form uneven partitions of the data. These approximate construction methods demonstrate the ability of neighbourhood graphs to span divisions and boundaries that may have been imposed on a dataset. And while these graphs should always contain the MST, the use of k as a global vertex out-degree does place an artificial limit on the ability for these

graphs to represent the intrinsic structure of the data. In particular, the density of edges remains constant around each vertex, while the actual density of vertices may vary considerably in different parts of the data space.

Another alternative to Delaunay graphs for high dimensional ANN searches are the small world graphs considered by Lifshits and Zhang [76]. These graphs are more similar to Delaunay graphs than k-nearest neighbour graphs in that the vertices have both short and long range edges. However, the average vertex degree for small world graphs can be much lower than Delaunay graphs as a random sampling method is used to assign a tunable number of edges. As with k-nearest neighbour graphs, the main requirement for the edges is to ensure connectivity between all vertices. Malkov et al. [77], [78] achieve this connectivity by sequentially inserting each vertex into the graph as a ANN query using a greedy search. The new vertex is then connected with each other vertex that was visited on the path from the starting vertex to its current nearest neighbour vertex. In addition to this, the new vertex is connected to a tunable number of vertices that have been found to be current local minima for greedy nearest neighbour searches. To improve the chance that an already inserted vertex is connected to a later inserted neighbour, the entire graph is undirected. However this does not guarantee that the MST will be a subgraph due to the presence of vertices that are not part of a global nearest neighbour pair. Malkov et al. claim that edges from the nearest neighbour searches form short range edges that approximate the Delaunay graph, while the edges to local minima are long range edges that provide the small world properties. Since Delaunay graphs approach being full connected in high dimensions, it seems likely that both the short and long range edges are contributing to the approximation of these graphs and all play a critical role in defining its properties. The tunable parameter for limiting edges to local minima, as well as the hand-engineered concept of combining the two edge sets does not allow these graphs to fully depart from the artificial limits on vertex degree that are used in k-nearest neighbour graphs. Overall, the complete construction algorithm of small world graphs does offer a significant speed-up for index construction when compared to k-nearest neighbour graphs. Even with

the faster construction, the approximate small world graphs produced by this method are still competitive with k-nearest neighbour graphs when evaluated for ANN recall and search efficiency. Additionally, by using a backtracking search for each query, a single accurate index can be constructed and then used to achieve various levels of recall by simply limiting the number of steps that the backtracking search can make.

Chapter 4

Fast Approximate Nearest Neighbour Graphs

In Chapters 2 and 3 we discussed a number of challenges that limit the performance of current approximate nearest neighbour (ANN) methodologies for high dimensional visual data. Due to the inherent nature of these spaces, methods that attempt to cluster or divide the data are defining indexing structures that place clear limits on the ANN performance. These limits arise partly from the lack of hierarchical structure in the embeddings of hand engineered visual features such as SIFT. Furthermore, the non-linear distributions found in high dimensional embeddings lead to poor performance when using efficient linear segmentations of the space.

More promisingly, we have seen that traversable neighbourhood graphs can be used to index a high dimensional dataset in a way that reduces the impact of segmenting the data. When compared to the structure of low dimensional Delaunay graphs, these high dimensional graph structures do not appear to fully utilise the complete intrinsic structure of the data they are indexing. We believe that by not utilising all of this structural information, these methods achieve a lower search performance than what should be achievable with an indexing graph that fully represents the local neighbourhoods found in the data. Following on from this analysis, we have developed a graph based indexing structure to exploit the low intrinsic dimensionality of high dimensional visual data. In this chapter we introduce the core structure of our graphs and then provide a detailed exploration of a number of alternative traversal and construction strategies.

4.1 The Occlusion Rule

The key innovation of our approach to ANN search is in utilising a local comparison that gives rise to desirable global structures. Specifically, we construct a list of directed outgoing edges independently for each vertex in a graph. This involves a process of evaluating each possible edge against the list of edges that have already been included for that vertex. Our aim when evaluating edges is to only include an edge if it forms part of the intrinsic structure of the data being indexed. The insight that allows us to construct a graph with these properties is that for efficient traversal, it is only necessary for each vertex to contain at least one edge that leads towards (but not necessarily reaching) each other vertex. If this criteria holds then each local neighbourhood of the graph can be traversed to reach any other neighbourhood, even if the path arrives via intermediate neighbourhoods. In this way our graphs can enable efficient global traversals that utilise an intrinsic dimensionality that is lower than the extrinsic dimensionality of the data.

For each pair of vertices $v_i, v_j \in V$ in an indexing graph $\mathcal{G} = (V, E)$ and with $v_i = \mathbf{x}_i$ from a dataset $\mathcal{X} \subset \mathbb{R}^m$, we require there to be a directed edge $e(v_i, v_k) \in E$ such that v_k (the vertex at the end of this edge) is closer to v_j than the starting vertex v_i is. If this condition holds for all vertices in a graph, then we consider it to be sufficiently connected for ANN searches. Every vertex in the graph will have at least one edge that will provide a step towards every other vertex. Mathematically, our initial criteria for edge selection requires that

$$\forall\{v_i, v_j \in V \mid i \neq j\},\ \exists\, v_k \in V \text{ such that}$$
$$k \neq i \ \wedge\ e(v_i, v_k) \in E \ \wedge\ dist(v_k, v_j) < dist(v_i, v_j). \tag{4.1}$$

This equation sets a lower bound on the number of edges in a graph, but also for many different graphs that will satisfy it. If we consider the situation when $k = j$, then the edge $e(v_i, v_j)$ will be included in our graph. If we assign $k = j$ for all pairs of i and j, then we will end up with a fully connected graph. As such, this lower bound is necessary but not sufficient for defining efficient traversable graphs.

We would like to define an upper bound on the number of edges in our graph that will also satisfy Equation 4.1. To achieve this, we postulate that it is undesirable for any two outgoing edges to have an angle θ of less than $\pi/3$ between them (or equivalently when $\cos\theta > 1/2$). As such we define an angular constraint where

$$\forall\{v_i, v_j, v_k \in V \mid i \neq j \neq k\}, e(v_i, v_k) \notin E \textbf{ if}$$
$$\exists\, e(v_i, v_j) \in E \text{ such that } \frac{(v_j - v_i).(v_k - v_i)}{\|v_j - v_i\|\,\|v_k - v_i\|} > \frac{1}{2}. \tag{4.2}$$

With this additional condition, an edge $e(v_i, v_j)$ can not be included in our graph if there is already an existing edge that also starts at v_i and is pointing in what we consider to be a similar direction. If there are fewer redundant edges leaving each vertex, then we can expect a reduction in the total size of the graph.

In practice, Equation 4.2 can form different graphs depending on the order in which edges are added to it. For the nearest neighbour search problem, we believe that it is important for the minimum spanning tree (MST) to be present as a subgraph. This will guarantee that each vertex is connected to its closest neighbour in the most direct way possible. To achieve this, we only allow shorter edges to remove longer ones. From the perspective of the starting vertex, we say that the longer edge is occluded by the shorter one. With this additional constraint, we define our angular occlusion rule as

$$\forall\{v_i, v_j, v_k \in V \mid i \neq j \neq k\},\ e(v_i, v_j) \textbf{ occludes } e(v_i, v_k) \textbf{ if}$$
$$dist(v_i, v_j) < dist(v_i, v_k)\ \wedge\ \frac{(v_j - v_i).(v_k - v_i)}{\|v_j - v_i\|\,\|v_k - v_i\|} > \frac{1}{2}. \tag{4.3}$$

Figure 4.1 provides a simple illustration of the angular occlusion rule. In Figure 4.1a the edge $e(v_i, v_j)$ is added to the graph because there are no shorter edges within a 60°range. This is also equivalent to saying that there are not vertices within the shaded region that is formed by a hyperspherical cone that is cut from a hypersphere centred on v_i and with a radius equal to $dist(v_i, v_j)$. Figure 4.1b demonstrates an occlusion of the longer edge $e(v_i, v_k)$ which will not be added to the graph as it falls within the 60°exclusion area around $e(v_i, v_j)$. Lastly, we can see that all vertices

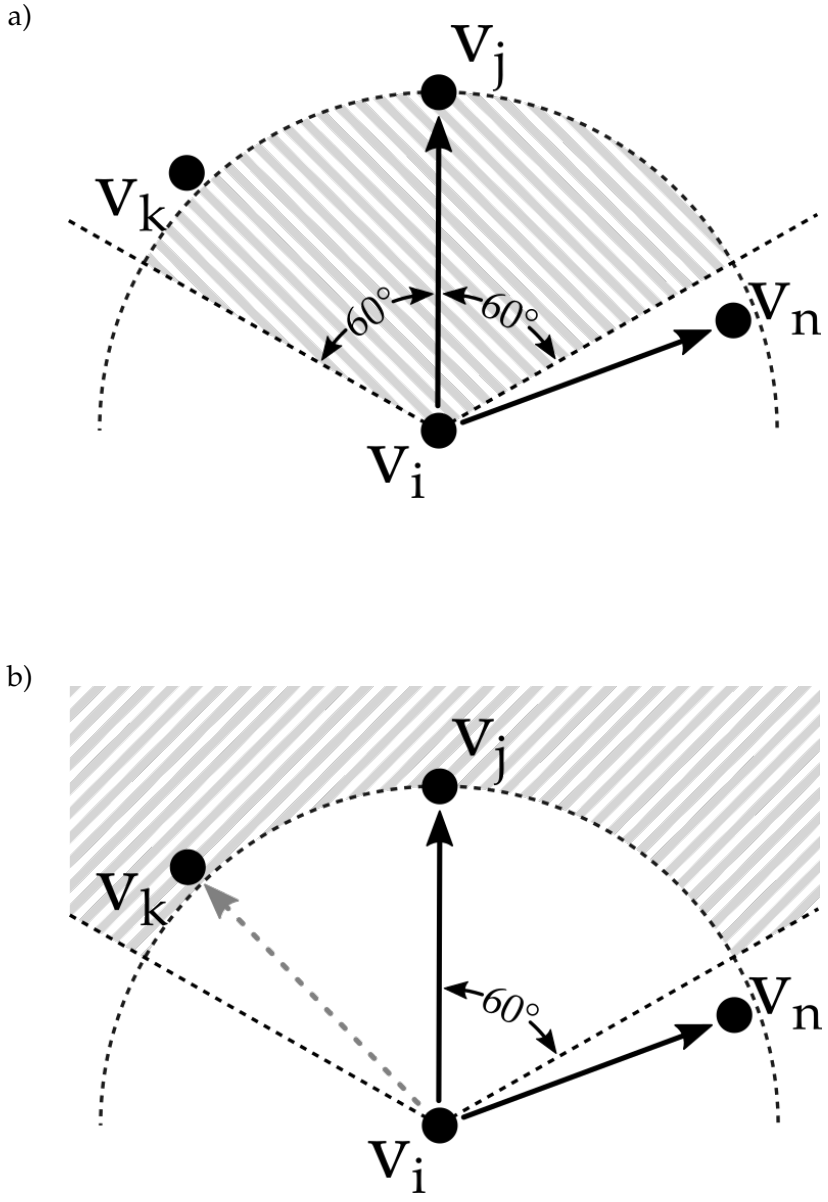


FIGURE 4.1: A simple illustrative example of the angular occlusion rule given in Equation 4.3. a) An unoccluded edge from v_i to v_j is added to the graph. b) The occluded edge from v_i to v_k is not added.

within the shaded outer region of the conical volume will also produce edges that are occluded by $e(v_i, v_j)$.

We observe that a very similar occlusion rule can be formulated to make use of distance relationships rather than angular ones. If we consider the situation where three vertices form a perfect equilateral triangle, then as with the angular occlusion rule, these vertices should form a fully connected subgraph. By adjusting the vertices to form an internal angle of less than $\pi/3$ we must correspondingly shorten the opposite edge of the triangle (relative to at least one of the other two edges). Since we are only allowing for shorter edges to occluded longer ones, we can consider an occlusion rule where edges are occluded if they are the longest edge of a triangle formed with an already included edge. This allows us to define our distance occlusion rule as

$$\forall\{v_i, v_j, v_k \in V \mid i \neq j \neq k\},\ e(v_i, v_j)\ \textbf{occludes}\ e(v_i, v_k)\ \textbf{if}$$
$$dist(v_i, v_j) < dist(v_i, v_k)\ \wedge\ dist(v_j, v_k) < dist(v_i, v_k). \tag{4.4}$$

The distance occlusion rule is illustrated in Figure 4.2 in the same way that we introduced angular occlusion. Figure 4.2a shows that for the distance rule, the shaded empty region that is needed for an edge to remain unoccluded is a lens formed by the intersection of two hyperspheres. These spheres are centred on the two ends of the edge being considered and both have a radius equal to the length of that edge. In Figure 4.2b we see that each edge added to the graph will occlude all edges with the same starting vertex that attempt to cross through the hyperplane bisecting that edge. As we have discussed in Chapter 2, it is often unfavourable to divide high dimensional spaces with hyperplane boundaries. In this case however, the minimal criteria we have specified in Equation 4.1 ensures that whenever we do cross a hyperplane, we are guaranteed that the neighbourhood we arrive in will have at least one way to get back towards where we came from.

In comparison to the angular occlusion rule, the distance rule requires a larger empty volume of space before it will form an edge in the graph. Equivalently, this rule forms a larger occlusion region and so some edges will be occluded despite the fact

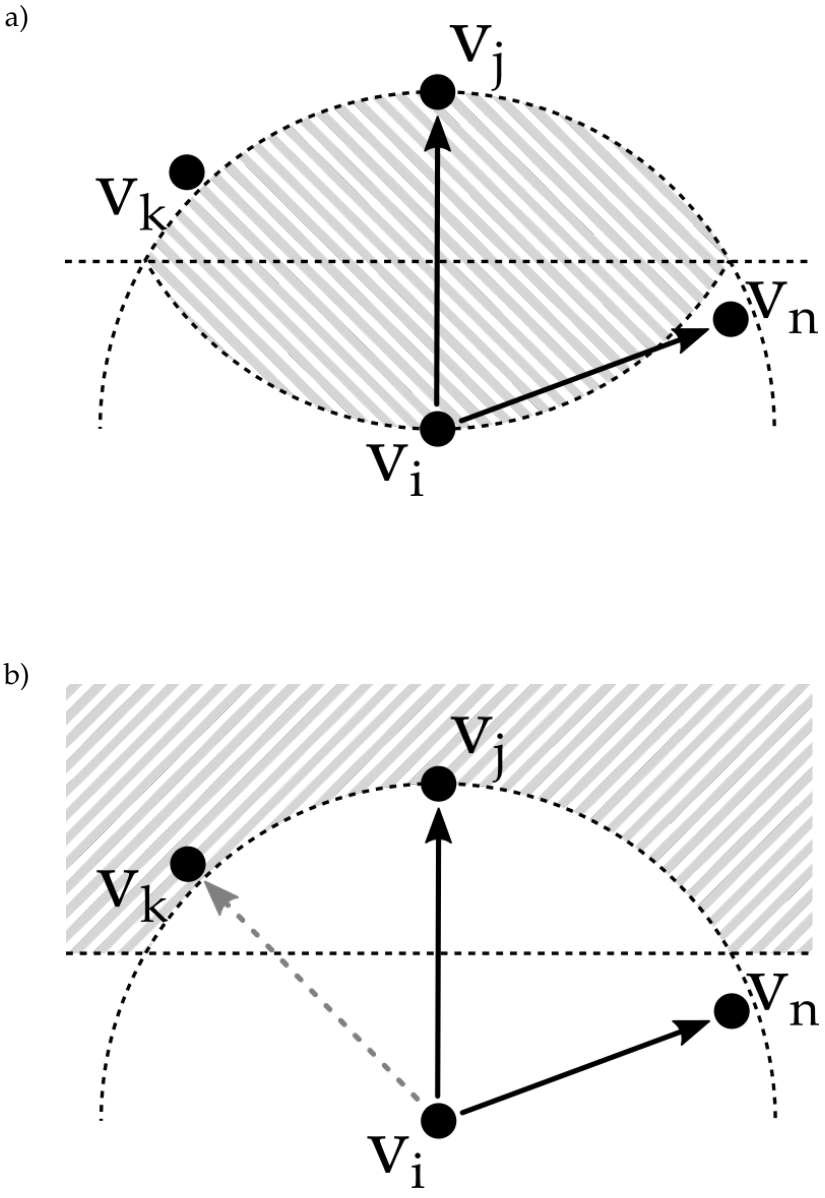


FIGURE 4.2: A simple illustrative example of the distance occlusion rule given by Equation 4.4. a) An unoccluded edge from v_i to v_j is added to the graph. b) The occluded edge from v_i to v_k is not added.

that they form an angle of more than $\pi/3$ with the occluding edge. Ultimately, while both occlusion rules will greatly reduce the number of edges in the graphs they define, the distance rule will almost always result in graphs with fewer total edges. Figure 4.3 plots this relationship with the distribution of vertex degree for both occlusion rules. There values are produced from indexing ten thousand SIFT vectors in a 128 dimensional feature space. The relative position of the centroid of each distribution demonstrates the higher degree vertices that are produced by the angular occlusion rule. Specifically, the angular occlusion graph has an average vertex degree of 49.9, while the distance occlusion graph has a much lower average degree of 15.3. Lastly, we note that there are no strong correlations between the out-degree and in-degree of the vertices in either type of graph.

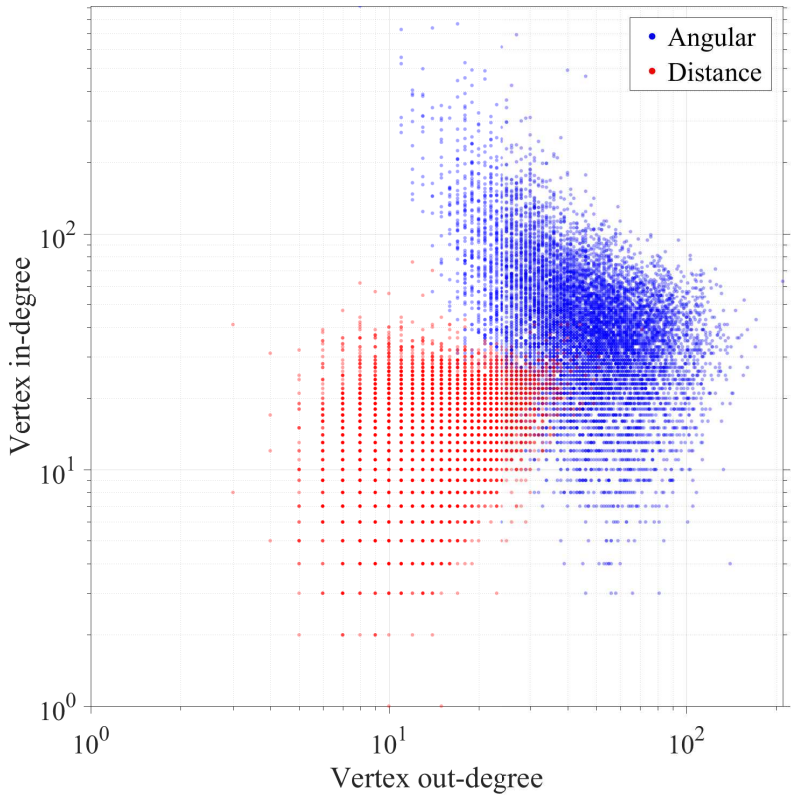


FIGURE 4.3: The out-degree and in-degree of each vertex contained in graphs constructed with either Equation 4.2 or 4.3. Each graph contains ten thousand SIFT vectors in a 128 dimensional feature space.

4.1.1 Related Graph Structures

We note a number of structures that are defined in a similar way to Equation 4.4 and have the common aim of creating sparse global edges based on simple and local rules. Firstly, we look at two graph structures that have a number of major differences from the graphs we have proposed. Most notably, these other structures are intended to be used in two dimensions, they are undirected graphs, and to our knowledge, they have not yet been applied to the ANN search problem. Here we briefly describe these graphs and in Section 4.4.2.2 we evaluate their ANN performance against our proposed structures. We then discuss a number of prior uses of Equation 4.4 that have arisen independently due to its close relationship with the triangle inequality.

As previously discussed in Section 3.4, the edges in Delaunay graphs define a useful set of local edges for low dimensional data. Due to this, they have also had some influence on the design of high dimensional indexing graphs. At a more practical number of dimensions, Delaunay graphs have inspired the formulation of a number of low dimensional graphs. Each of these low dimensional graphs are both a spanning subgraph of the Delaunay graph, as well as a supergraph of the MST. Toussaint [79] defines relative neighbourhood graphs (RNGs) by connecting all pairs of vertices only when there are no other samples that are closer to either member of a pair than the distance between that pair. This rule defines an exclusion zone like the shaded lens seen in Figure 4.2a. However, it is important to note that RNGs allow a vertex to prevent the inclusion of an edge, while our graphs only allow for edges to occlude each other. Urquhart [80] defines a structure similar to RNGs by starting with a full Delaunay graph and then removing every edge that is the longest side of a triangle. While the definition of RNGs can be trivially extended to higher dimensions, it is unclear how many edges should be removed from the simplices of higher dimensional Delaunay graphs, in order to produce an Urquhart graph. It is also likely that the rapid growth in vertex degree for higher dimensional Delaunay graphs will ultimately make Urquart graphs unsuitable to construct in these spaces.

Figure 4.4 illustrates four different graphs, each of which are built on the same set

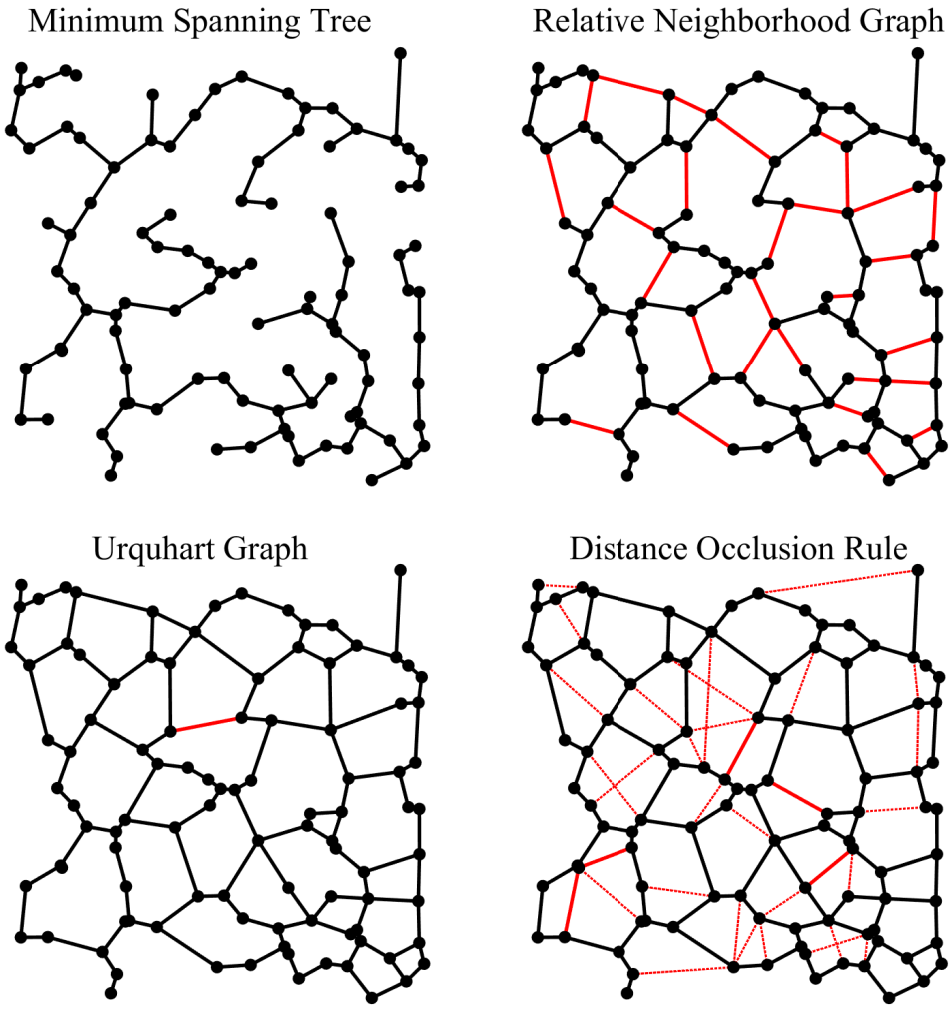


FIGURE 4.4: Neighbourhood graphs constructed on random two dimensional data. Solid lines represent undirected edges and dotted lines are directed edges. Coloured lines show the inclusion of additional edges when compared to a previous structure.

of random two dimensional vertices. The edges of the RNG are differentiated as black edges that form part of the MST and coloured edges that are only found in the Delaunay graph. In the context of ANN search, these additional edges form short connections that can reduce the amount of backtracking needed for exploring the MST. The Urquhart graph is almost identical to the RNG except for the inclusion of one additional edge from the Delaunay graph. This indicates that the edge is not the longest side a triangle in the Delaunay graph, but there is at least one other vertex within the occlusion lens defined by this edge. The last graph is defined by our distance occlusion rule from Equation 4.4. Here, the four solid coloured edges are found the the Delaunay graph but not in the RNG. This occurs because a vertex is

excluding the edge in the RNG, but that same vertex has not formed an edge in our graph (due to occlusion from a shorter edge) and so there is no occlusion here. The dotted edges in our graph represent additional edges that are unoccluded in only one direction. Since a number of these directed edges form crossing paths, this set must include some edges that are not found in the Delaunay graph.

Due to their close relationship with Delaunay graphs, naive construction of a RNG can be computationally expensive. Jaromczyk and Kowaluk [81] proposed an alternative construction method that begins by constructing a supergraph of the RNG. This intermediate graph utilises the properties of the triangle inequality and allows shorter edges to occlude longer ones in a similar way to Equation 4.4. However, the ultimate goal of constructing a RNG only requires the use of undirected edges. As such, this intermediate result will not contain any of the undirected edges found in our graphs.

Arya and Mount [82] built upon this intermediate graph for the purpose of ANN search with sixteen dimensional speech data. Their initial theoretical analysis looks at defining a global vector space that can be translated onto each vertex in a graph. This vector space then defines a set of non-overlapping convex cones that independently determine the inclusion of edges that would fall within them. However, the performance of this theoretical graph is stated to be uncompetitive, and so the authors define a practical alternative by incorporating concepts from RNGs. The resulting structure is a sparse neighbourhood graph (SNG), which used the occlusion method that was previously defined for constructing RNGs. Because the SNG is a directed graph, the edge selection method is now identical to Equation 4.4.

Navarro [67] independently applied the same edge selection method hierarchically, in order to produce an indexing tree for the exact nearest neighbour search problem. Construction of this spatial approximation tree (SAT) begins by computing all directed edges from a randomly chosen starting vertex. All remaining vertices are then assigned to a leaf node of the tree, so that the edge selection can then be repeated from each neighbour of the starting vertex. A tree structure is maintained with each leaf node only considering edges to the vertices that have been assigned to it. Additional layers of the tree are constructed by repeating this process until all

vertices are included in the index. The SATs are then evaluated on natural language data in up to twenty dimensions.

The distance occlusion rule that we have defined in Equation 4.4 has been repeatedly shown to produce desirable structures for indexing datasets of between two and twenty extrinsic dimensions. The equation represents a fundamental application of the triangle inequality and can be applied in an arbitrary number of dimensions. Irrespective of a datasets dimensionality, the three vertices that are considered by the occlusion rule can always be placed on a single two dimensional plane. After independently arriving at this equation, our research introduces additional experiments that explore many new and novel alternatives for constructing and searching this type of index. As a final point of distinction, our research is focussed on the domain of high dimensional visual data. The dataset we use in our experiments have a considerably larger extrinsic dimensionality, and also contain many more samples than the datasets used in existing works.

4.2 Evaluating ANN Performance

When evaluating the performance of an ANN index we are primarily interested in achieving fast and accurate searches. This can generally be achieved by fixing either the search time or and target accuracy and then measuring the free variable. For high dimensional image data, the search time is dominated by the computational cost of calculating pairwise distances. From Equation 2.1 we can see that each distance requires a least three arithmetic operations per dimension, so this cost becomes more dominant as the extrinsic dimensionality increases. The cost of the square root can be easily avoided for comparisons as its inverse is a monotonic function. As such, a theoretical search time can be measured by counting the number of distance computations during the search, rather than the practical measure of recording the elapsed wall clock time. This measure has the major benefit of evaluating the cost of an algorithm in a way that is independent of the hardware being used. Measuring search accuracy then involves allocating an allowable number of distance calculations (bounded by the size of the dataset) and processing a set of queries before comparing

their resulting neighbour sets to a precomputed ground truth. Similarly, because the hardware is being abstracted out of the cost calculation we are not accounting for other important factors such as cache coherency and thread divergence, which can only be effectively measured with wall clock time. As such, we also report wall clock times for our best performing implementation.

A more thorough theoretical evaluation method is to run each query until either the full dataset has been compared to or until no additional progress is being made. During these searches we can keep a record of the order in which each comparison was made. This allows us to produce a full trade-off curve of cost against recall by comparing the ground truth to increasingly large sections of the search records and counting the fraction of completed queries after each number of distance calculations. It is conventional to plot these curves on a log-linear scale with speed-up over linear search against recall. In order to improve the readability of performance at high recall, we instead use a log-log plot to show speed-up over linear search against the error rate. This format provides a clear comparison of the drop-off in performance as the error approaches zero. Here, the speed-up is computed as the size of the dataset divided by the number of distance calculations used and the error rate is fraction of queries that did not achieve the ground truth result. As such, these plots are showing a speed-up relative to a naive linear search and are ignoring possible optimisations in order to aid analysis of the theoretical performance of each method.

One such optimisation that can impact the performance of both the linear search and our own methods is the use of early bailout. When searching a dataset for the nearest neighbour to a particular query, it can be beneficial to partially compute distances until it is guaranteed that a sample can not be the nearest neighbour. During a linear search of a dataset, we expect that finding samples nearer to our query will on average allow for increasingly less computation to be performed for each comparison. Figure 4.5 plots the expected savings when applying this technique to data of varying dimensionality. For randomly sampled data, both the binary and continuous sampling show a similar trend of receiving the most benefit when the dimensionality is in the range of ten to twenty extrinsic dimensions. Outside of this range,

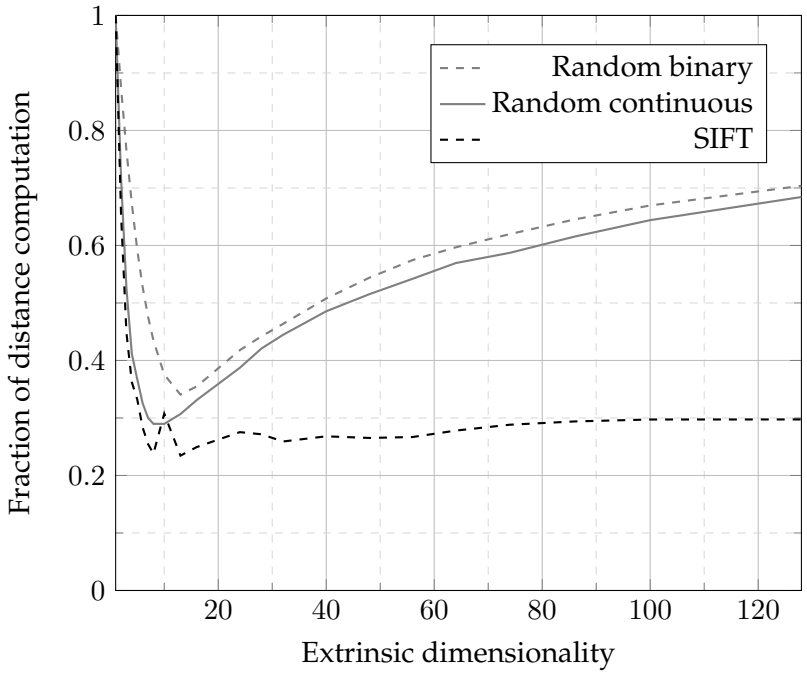


FIGURE 4.5: The expected fraction of each distance calculation that is required when finding a nearest neighbour when using early bailout with data of increasing dimensionality.

we see that early bailout will still provide a diminishing reduction in computation as dimensionality increases. In contrast, the SIFT dataset that has been incrementally truncated shows a consistently large gain in efficiency. As such, it is advisable to include early bailout in any practical implementations where the fully computed distances will not be required for subsequent calculations.

There are a number of additional choices we need to make when evaluating an ANN index. The query set that is used for evaluation should be representative of the type of queries that are expected to be seen in the target area of application. In general, this means using previously unseen queries that are sampled from the same distribution as the dataset, as is the case for the dataset we have outlined in Table 1.2. Additionally, we find that evaluating the search performance with known vectors does also provide useful information about the structure of the index. Based on their inclusion in the indexed dataset, we refer to unseen queries as external queries and the known vectors from our test set as internal queries. Additionally, the size of the datasets we are indexing in our experiments have been chosen predominantly to allow for feasible run times. However, there is still an open question as to how

much data is actually required for adequate parameter tuning.

It is also useful to apply empirical analysis for gauging the costs and benefits of targeting particular hardware. For this purpose, it is easiest to measure the search time of a query using either wall-clock time or the total number of clock cycles used by the process. In the case of wall-clock time, large gains can be achieved through parallelisation that allows multiple queries to be processed simultaneously. This measure also provides an easy way to compare the bookkeeping and memory access overheads for particular implementations of an ANN algorithm. Other measurements that are often of interest for the practical use of an ANN index are the memory requirements for running queries on the index and the memory and time cost of constructing the index. Together, these can determine the feasibility of indexing a particular dataset on a particular hardware platform.

4.3 Searching FANNGs

When performing an ANN search on a traversable graph we are interested in finding samples from a previously indexed dataset $\mathcal{X}$, that are in the same local neighbourhood as a particular query vector $\mathbf{q}$ from the query set $\mathcal{Q}$. The dataset is indexed with a graph $G = (V, E)$ where $V \subseteq \mathcal{X}$ are the vertices of the graph and E is a set of directed edges. For each query, the graph allows us evaluate the distance between a current vertex and $\mathbf{q}$ and then traverse a previously unexplored edge to consider a new vertex. In general, we aim to find the local neighbourhood of $\mathbf{q}$ while evaluating distances from as few vertices as possible.

Due to the long history of research that explores graph structures, the question of how best to travel through an ANN indexing graph during a search has also been explored in detail. However, many algorithms have been shown to perform well and for each of these there are often many possible implementation decisions. It is not immediately clear what search strategy will work the best for the structures we have defined. As such, we aim to build upon and broaden the search strategies that are introduced for the indexing methods we discussed in Chapter 3. In this section

we look at implementing a number of promising search strategies and compare their results when applied to our graphs.

A number of small but advantageous implementation decisions have been used across all of the search strategies we present. Firstly, since our graphs do not change during or in-between queries we implement our search algorithms so that batches of many queries can be run in parallel on the same graph. Due to the nature of high dimensional graphs, memory access for a single query is highly random and so we do not expect to pay a penalty for running concurrent searches. While this optimisation does improve wall-clock time for processing each batch, it does not affect the counting of distance calculations for each of the queries in the batch.

In regards to representation in memory, we load the dataset $\mathcal{X}$ from a single file that contains a concatenation of the vectors that form our graph vertices V. The edges of the indexing graph are also loaded as a block of concatenated vectors, each of which represents a list of edges $\mathbf{e} \in E$. The graph data contains a list of integers for each vertex $v \in V$, where each entry represents an edge from v and stores the integer index of the end vertex. In situations where these edge lists contain a varying number of edges, we enforce uniform lengths by padding all of the edge lists to the length of the longest edge list. The padding is implemented using a reserved value that indicates the end of the list (such as -1 or UINT_MAX). Because of this, iterating to the end of an edge list is achieved either when the reserved value is seen or when we have reached the maximum edge list length for that particular graph. Once loaded, we store both the vertex data and the edge data as a vector of vectors. As each edge is loaded we compute the square of the edge length and then store pairs of destination index and squared distance.

In situations where the indexed dataset contains duplicate entries, we detect these duplicates during index construction and then only include the first of each duplicate set as a vertex in the graph. We achieve this by ensuring that the removed vertex maintain an empty edge list, and that any edge that would point to a removed vertex is instead directed towards the first vertex in its duplicate set. Since we don't allow self-loops in the graph, duplicates can be easily detected by checking that each new edge has a non-zero length. Additionally we store a list of duplicate pairs that each

associate a non-vertex sample with the vertex it is coincident with. This allows us to ignore the issue while executing a query and then if the need arises, we can check to see if any of the query results are present in the duplicate list and return each of the samples in those duplicate sets.

Lastly, it is possible to save some number of unnecessary distance calculations by keeping track of previously visited vertices. In this way, if a search attempts to traverse an edge back to an already visited vertex, then we can detect that a distance has already been computed between that vertex and the query vector. Perhaps more importantly for a graph containing cycles, this lets us guarantee that the search will not become stuck traversing the same cycle within the graph. In situations where memory is not constrained, we can allocate a single bit per vertex to flag when that vertex has been visited. When running multiple parallel queries, each query requires its own set of flags. Ultimately, an interesting dataset will contain a large number of data vectors, and because our goal for each query is to only evaluate distances to a small fraction of these, it is usually faster to maintain a set of flagging indices that need to be reset at the end of each search than it is to reset every flag in-between searches. In situations where constrained memory makes using these flags impossible, it is sufficient to just maintain $U \subseteq V$ the set of indices that have been visited. If each search only visits a small number of vertices, then linearly searching this set before each edge traversal will remain relatively quick. If the set grows too large it becomes preferable to use a data structure (such as a hash table or binary tree) that can offer a more efficient lookup. In all cases, the set U can play the additional role of tracking the best k vertices that have been seen so far. This can be implemented by maintaining U as a (possibly truncated) sorted list that is ordered by ascending distance to the query.

4.3.1 Traversing Local Neighbourhoods

The graphs defined by Equations 4.3 and 4.4 are formed from edge lists than can be constructed independently at each vertex by applying the same simple rules. Due to this, we expect that the structure of these graphs will be mostly uniform. Although we should also expect to see some structural variation to reflect local differences

in the dataset being indexed. The inherent structure of the dataset will ultimately determine the effectiveness of indexing methods and search algorithms. Here we look at properties of the local graph neighbourhoods and how they can inform an efficient search for a query vector.

Figure 4.6 contains four different mappings of ten thousand SIFT vectors relative to a prototypical query vector. The data we show here uses an external query and a graph built with the distance occlusion rule, however we describe any important changes that arise from using an internal query or the angular occlusion rule. Figure 4.6a looks at the length of a vector from each vertex to the query and plots it against

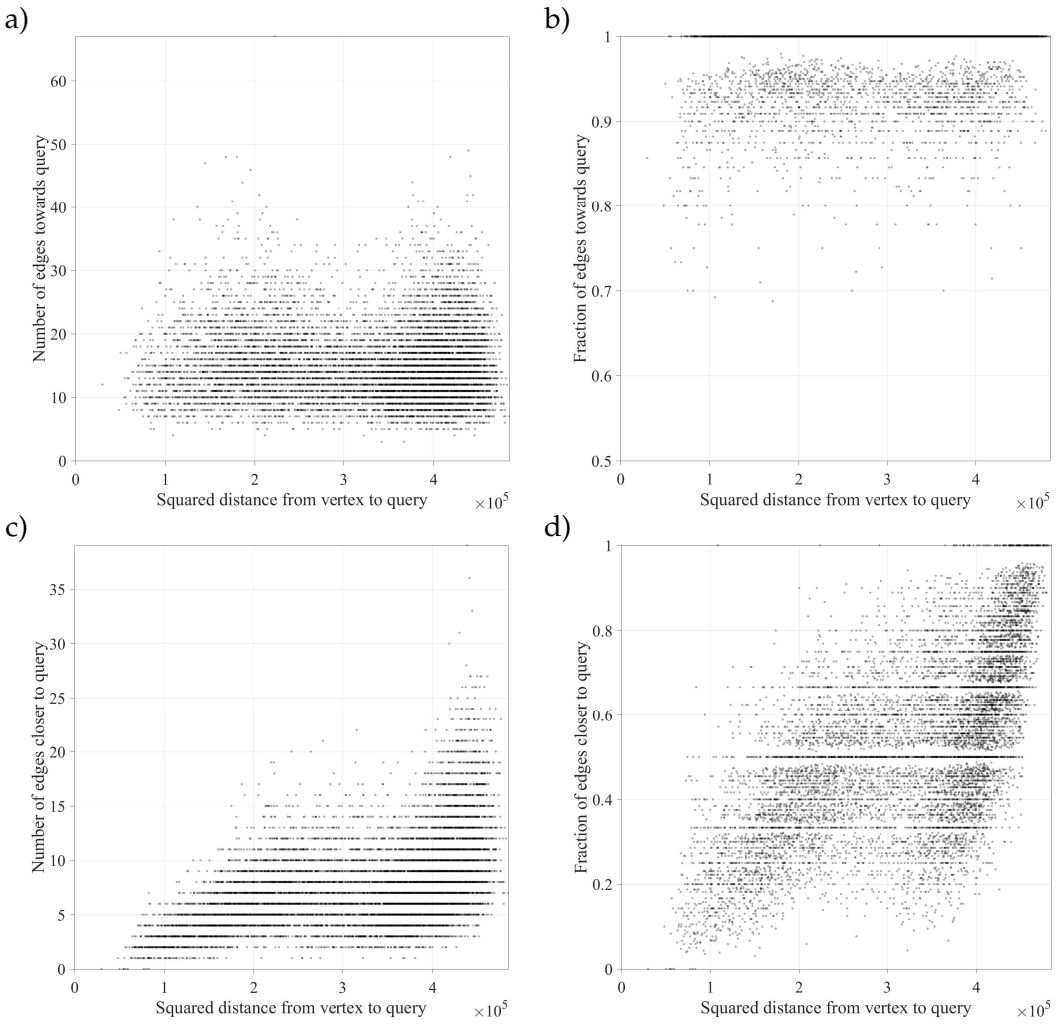


FIGURE 4.6: Distributions of edges from vertices of a graph constructed on ten thousand SIFT vectors using Equation 4.3. a) Number of edges pointing towards the query. b) Fraction of each edge list pointing towards the query. c) Number of edges ending closer to the query. d) Fraction of each edge list ending closer to the query.

the number of edges which start from that vertex and produce a positive dot product with the vector. In other words, we are interested in knowing how proximity to a query will affect the number of edges pointing in the direction of that query. As is expected for a uniform structure, we find that there is noticeable change between the distribution of edges at different distances from the query. The only exception to this is a small amount of bimodal structure due to the two Gaussian clouds that are present in SIFT data. The spread in the number of edges at any particular distance can be accounted for by changes in the local density of the data vectors. If we consider the furthest side of the furthest Gaussian cloud from the query, then we see an overall decrease in the number of edges when comparing to the distribution of the denser central region of the cloud. This is favourable for our graph structures, as edge lists that reflect the changes in local density are an important property for exploiting the low intrinsic dimensionality of the data.

In Figure 4.6b we see the same comparison from the previous plot, however now the number of edges are given as a fraction of the out-degree for each vertex. Since this controls for the effects of edge list sizes, the disappearance of the bimodal structure is further evidence of a relationship between vertex degree and local density. Perhaps it is surprising that around 80% of all vertices have their entire edge list producing a positive dot product. However, this validates a number of properties that were discussed in Chapter 2. Firstly, since this 80% must cover the majority of vertices in both Gaussian clouds we see that a single parameter from along this linear axis has very little impact on the full global structure. This indicates that much like the data, the graph has a uniformity that lacks a hierarchical structure. Across the entire dataset there is no small set of vertices that have a special or unique vantage on the query. This also ties into the concept that the entire dataset is bound between two large hyperspheres that are both centred on the mean of the data. And since every vertex is essentially sitting on the surface of a lower dimensional manifold, edges will connect with local neighbours while still have some component in the direction of most of the other vertices.

While most edges point in the direct of a query, the plots in Figure 4.6c and 4.6d count only the edges with an end point that is closer to the query than the starting

vertex is. These remaining edges are the ones that can be traversed during an ANN search in order to make progress towards the local neighbourhood of the query. In contrast to the previous two plots, we now see a significant overall decrease in the number of edges being counted. There is also a clear relationship that shows as the distance to the query reduces, there is also a consistent reduction in the counts. If we consider a hypersphere centred on the query and with a particular vertex on its surface, then we have seen that for most vertices all of their edges are directed into the hypersphere. Now we see that when the vertex and query are closer together, the hypersphere is smaller meaning that more of the edges will continue through the hypersphere and terminate out the other side. This is easier to achieve in high dimensional spaces, as vectors are more likely to be nearly perpendicular to each other. In the context of ANN search, Figure 4.6d shows us that as the search progresses, each edge list will have increasingly fewer edges that can continue the progress towards the query. The leftmost vertex on the x-axis is the nearest neighbour to the query, however there are also a number of other points with no edges being counted. Each of these other vertices are local minima that can prematurely halt a greedy search. For an internal query, the structure of our graphs provide a guarantee that the query vector will be the only vertex with no edges being counted.

4.3.2 Exhaustive Downhill Search

The most common way to use decision trees and indexing graphs for finding the nearest neighbour of an arbitrary query vector is start at some vertex, test each outgoing edge from that vertex and then follow the edge that terminates closest to the query. Testing each edge and following one of them is repeated until all outgoing edges terminate at vertices that are further away from the query. This process, which is given in Algorithm 4.1, is often referred to as greedy search, however here we refer to it as an exhaustive downhill search due to evaluation of every outgoing edge before moving to the next vertex. The computational complexity of this algorithm is dominated by the distance calculation inside the two loops. For a graph of n vertices each with m extrinsic dimensions and with an average vertex degree given by E_{avg}, it seems that the worst case complexity will be $O(nE_{avg}m)$ if every vertex is visited

during a search. However, because the distance to the query always decreases at each step, and a decreasing distance bounds the number of remaining steps to be no more than the number of vertices within a shrinking hypersphere, then there is a much lower limit on the expected number of loop executions. Regardless of where a search starts and ends, the number of vertices along the search path will always be bounded by the diameter of the graph. At each vertex along this path, there can only be as many distance calculations as there are outbound edges. As such, each search has an equal cost to traversing a path through a weighted graph, where at each vertex all outbound edges have a weight equal to the degree of that vertex. For these graphs, the weighted diameter d_w is the maximum number of distance calculations for an exhaustive downhill search. Hence, the worst case complexity for this search can be expressed as $O(d_w m)$.

Algorithm 4.1: Exhaustive downhill search

Input: graph vertices V
graph edge lists E
starting vertex U_0
query vector $\mathbf{q}$
Output: nearest neighbour $best$

```
1  U ← empty set of visited vertex indices
2  prev ← null
3  best ← {U_0, dist(U_0, q)}
4  U.add(U_0)
5  while prev ≠ best and |U| < |V| do
6      prev ← best
7      foreach e ∈ E[best.index()] do
8          if e.to() ∉ U then
9              newDist ← dist(V[e.to()], q)
10             if newDist < best.dist() then
11                 best ← {e.to(), newDist}
12         U.add(e.to())
13 return best
```

In Table 4.1 we evaluate the 1-nearest neighbour recall (written as R@1) and the associated average query costs for exhaustive downhill searches using our graphs. We compare the graphs generated by each of the occlusion rules on both internal and external queries, and for a range of different starting locations. The random start simply begins each search by uniformly choosing a start vertex from the graph, the

TABLE 4.1: Traversing FANNGs using exhaustive downhill search.

Start location	Angular occlusion				Distance occlusion			
	Internal		External		Internal		External	
	R@1	Cost	R@1	Cost	R@1	Cost	R@1	Cost
Random	1.00	239.8	0.68	240.3	1.00	110.6	0.57	106.8
Medoid	1.00	264.9	0.70	274.0	1.00	121.6	0.54	112.7
Surface	1.00	200.9	0.72	205.7	1.00	105.1	0.56	97.2

surface start always begins from the vertex that is furthest from the centroid of the dataset and similarly, the medoid start always begins from the vertex that is closest to this centroid. The graphs are constructed on a dataset of ten thousand SIFT vectors and the results are all averaged across a fixed set of 100 unique queries. The data presented in the table empirically verifies a number of properties that have previously been discussed. As a direct result of the occlusion rules, both graphs maintain perfect recall across all of the internal queries. The angular occlusion graph consistently has a much higher search cost than the distance occlusion graph due to the higher average vertex degree from the less discriminative angular occlusions. However, it is important to note that the additional edges also enabled a higher recall for external queries on the angular occlusion graph than was seen from the distance occlusion graph. Lastly, there was very little difference in the results from the different starting locations, which reinforces the concept of uniformity throughout the graph and the data space.

Lastly, Figure 4.7 presents a visualisation of the search paths for an internal and external query using the mapping from Figure 4.6d. Just as before, we achieve this by using the distance occlusion rule to index a set of one thousand SIFT vectors and then analysing the index. Figure 4.7a shows the path from a random starting vertex to another randomly selected query vertex. Because we are using a downhill search, each step in the path must move towards the left of the graph and bring the search closer to the query. Since this is an internal query we can guarantee that the search will eventually reach the query vector. In Figure 4.7b we show the same search, but display all of the explored edges regardless of if they are traversed. Each edge is coloured to correspond with the associated step in the previous plot. Throughout the search most of the comparisons are evaluating vertices that are further from the

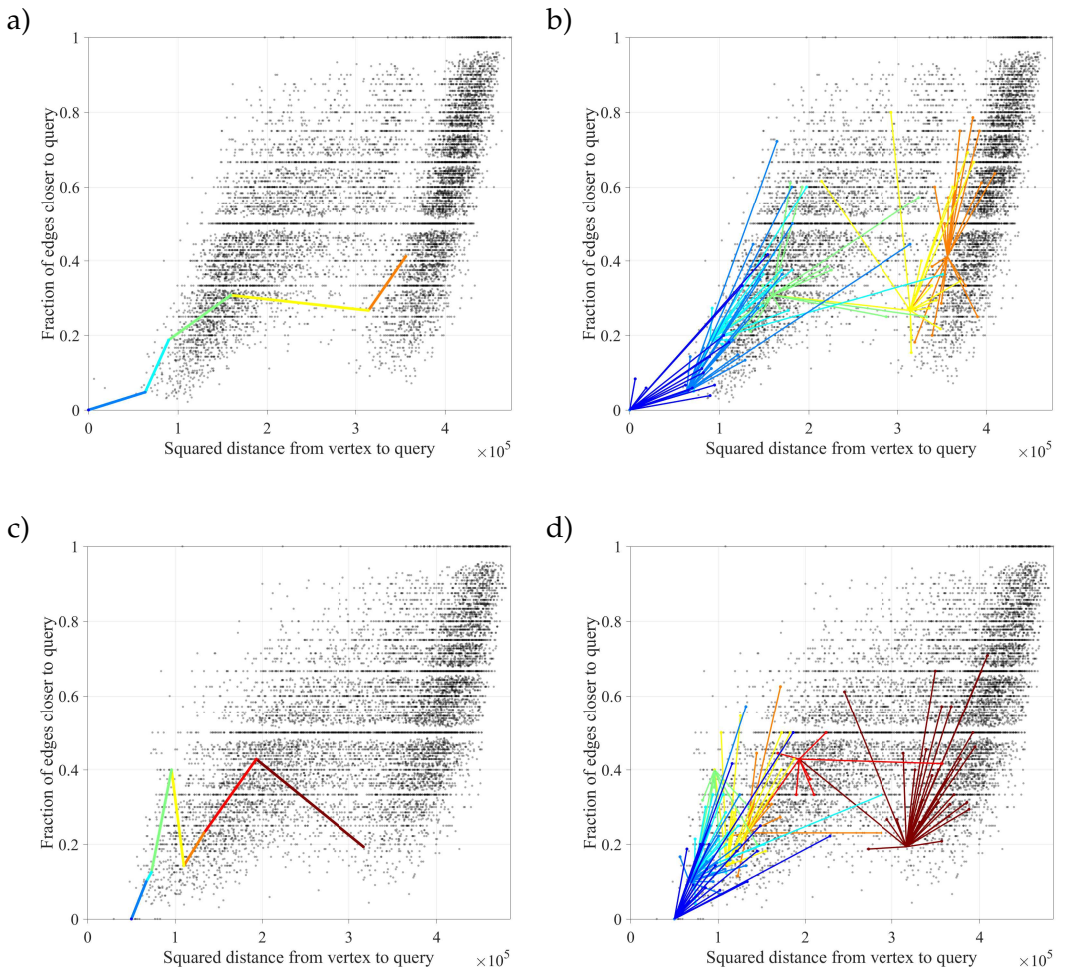


FIGURE 4.7: Exhaustive downhill search paths projected onto a two dimensional parametrisation of ten thousand SIFT vertices. a) Search path towards an internal query. b) All evaluated edges for the internal query. c) Search path towards an external query. b) All evaluated edges for the external query.

query than the current vertex. This is most noticeable towards the end of the search when the vertices have a much smaller fraction of their edge list leading closer to the query. The plots in Figure 4.7c and 4.7d illustrate the search path and comparisons for an external query. The downhill search algorithm continues to ensure that progress is made with each step, however there is no longer a guarantee that the global minimum will be reached. For the query shown, the search path reaches a local minimum and terminates without reaching the global optimum. Ideally we would like to reduce the number of unused comparisons and also include a mechanism for escaping local minima.

4.3.3 Greedy Downhill Search

Due to our use of an ordered edge list at each vertex as well as the specific structure of our graphs, we define a downhill search that is greedier than the exhaustive downhill search that is commonly used with other indexing structures. Algorithm 4.2 provides an overview of the greedy downhill search, which looks almost identical to the exhaustive search in Algorithm 4.1. The key difference between the two algorithms is the addition of line 12 to Algorithm 4.2. This additional statement instructs the search to break out of the inner loop as soon as any edge can bring it closer to the query. By exiting the loop early, the search is able to move to a closer vertex before all of its outbound edges have been checked. While this does mean that some search steps will not progress the maximum possible distance towards the query, every step will still make some amount of progress towards it. For internal queries, the occlusion rules still guarantee the existence of at least one edge that leads closer to every other vertex in the graph. As such, regardless of which vertex we end up at when following edges during an exhaustive search, the new neighbourhood will always contain an edge that allows the search to progress.

Algorithm 4.2: Greedy downhill search

Input: graph vertices V
graph edge lists E
starting vertex U_0
query vector $\mathbf{q}$
Output: nearest neighbour $best$

```
1  U ← empty set of visited vertex indices
2  prev ← null
3  best ← {U_0, dist(U_0, q)}
4  U.add(U_0)
5  while prev ≠ best and |U| < |V| do
6      prev ← best
7      foreach e ∈ E[best.index()] do
8          if e.to() ∉ U then
9              newDist ← dist(V[e.to()], q)
10             if newDist < best.dist() then
11                 best ← {e.to(), newDist}
12                 break
13             U.add(e.to())
14 return best
```

The key goal of the greedy downhill search is to reduce the number of unused distance calculations at each vertex along a search path. In Figure 4.6c we saw that nearly all of the vertices have multiple edges leading closer to a particular query. With a higher fraction of these edges in an edge list, it becomes far more likely that at least one such edge will occur early in the list. This presents an optimisation problem, where for the edge list of every vertex we would like to find an ordering that will minimise the average cost of including that vertex in the search path of a query. This indicates that for our greedy downhill search, unlike the exhaustive search, the search cost will be dependent on the iterating order at line 7 in Algorithm 4.2. Based on this dependency, we can again express the worst case complexity of the search as $O(d_w m)$. However, in this case the weighted diameter d_w is measured on a graph where at each vertex, the outbound edges have a cost equal to their edge list index.

In Table 4.2 we evaluate the greedy downhill search with four different edge list orderings that can be independently computed at each vertex. We compare the graphs generated by each of the occlusion rules on both internal and external queries, and with each different edge list ordering. The shortest length ordering compares the length of each edge in an edge list, and then arranges them in ascending order. Because both of the occlusion rules specify that only shorter edges can occlude longer edges, the shortest length ordering is the natural order when adding edges to an edge list. In contrast, the longest length ordering is simply the reverse of a shortest length ordering. As such, all of the edge lists are sorted in descending order of edge length. Both the largest angle and smallest angle orderings begin by selecting the shortest edge and placing it at the start of the edge list. Subsequent edges are then selected by comparing all of the unselected edges against the current pool of

TABLE 4.2: Traversing FANNGs using greedy downhill search.

Edge ordering	Angular occlusion				Distance occlusion			
	Internal		External		Internal		External	
	R@1	Cost	R@1	Cost	R@1	Cost	R@1	Cost
Shortest length	1.00	121.2	0.63	126.7	1.00	64.4	0.48	60.2
Longest length	1.00	160.2	0.64	177.4	1.00	78.2	0.51	73.8
Largest angle	1.00	97.3	0.69	116.8	1.00	53.1	0.50	47.3
Smallest angle	1.00	164.7	0.64	157.7	1.00	73.2	0.52	64.5

selected edge and adding the one based on a greedy criterion. For the largest angle ordering, an edge is selected if the minimum angle between it and any of the selected edges is the largest. Similarly, the smallest angle ordering selects the smallest maximum angle for the set. Overall, the most notable results come from comparing the different edge list orderings. The impact of the different occlusion rules and query types remains similar to what was seen for the exhaustive downhill search. Results for the internal queries clearly demonstrate a significant reduction in search cost when using the greedy downhill search. At their best, the early loop terminations have halved the average search. This result also indicates that the diameter of the weighted graph has been significantly reduced by the new weighting of the edges. Comparing the different edge list orderings, the largest angle ordering consistently performs the best and the shortest length ordering is second best. While long edges should have the potential to make a large amount of progress towards a query, in practice it is more efficient to favour the shorter edges. Despite this apparent bias, all of the longer edges are still an essential component in guaranteeing that there are no local minima for searches using internal queries. The significant cost reduction from using the largest angle ordering shows an even stronger bias. As edges are being evaluated and found to be leading further away from the query, it is a good strategy to then look in the opposite direction to where the previous edges have been directed.

In Figure 4.8 we visualise the search path of the same internal query that was used for the exhaustive downhill search in Figure 4.7a and 4.7b. Contrasting this to Figure 4.8a shows that applying the greedy downhill search has resulted in more vertices along the search path. Despite traversing across these additional vertices, the total search cost has been reduced by about one third. In Figure 4.8b we can see each vertex that was evaluated throughout the search. When compared to the exhaustive downhill search, many of these intermediate vertices have performed relatively few checks before an edge has allowed the search to progress. The plots in Figure 4.8c and 4.8d provide an additional view of this search. Both plots show the search progress from left to right, and are divided into coloured bands that each correspond to a particular vertex from the search path. Figure 4.8c shows the distance between

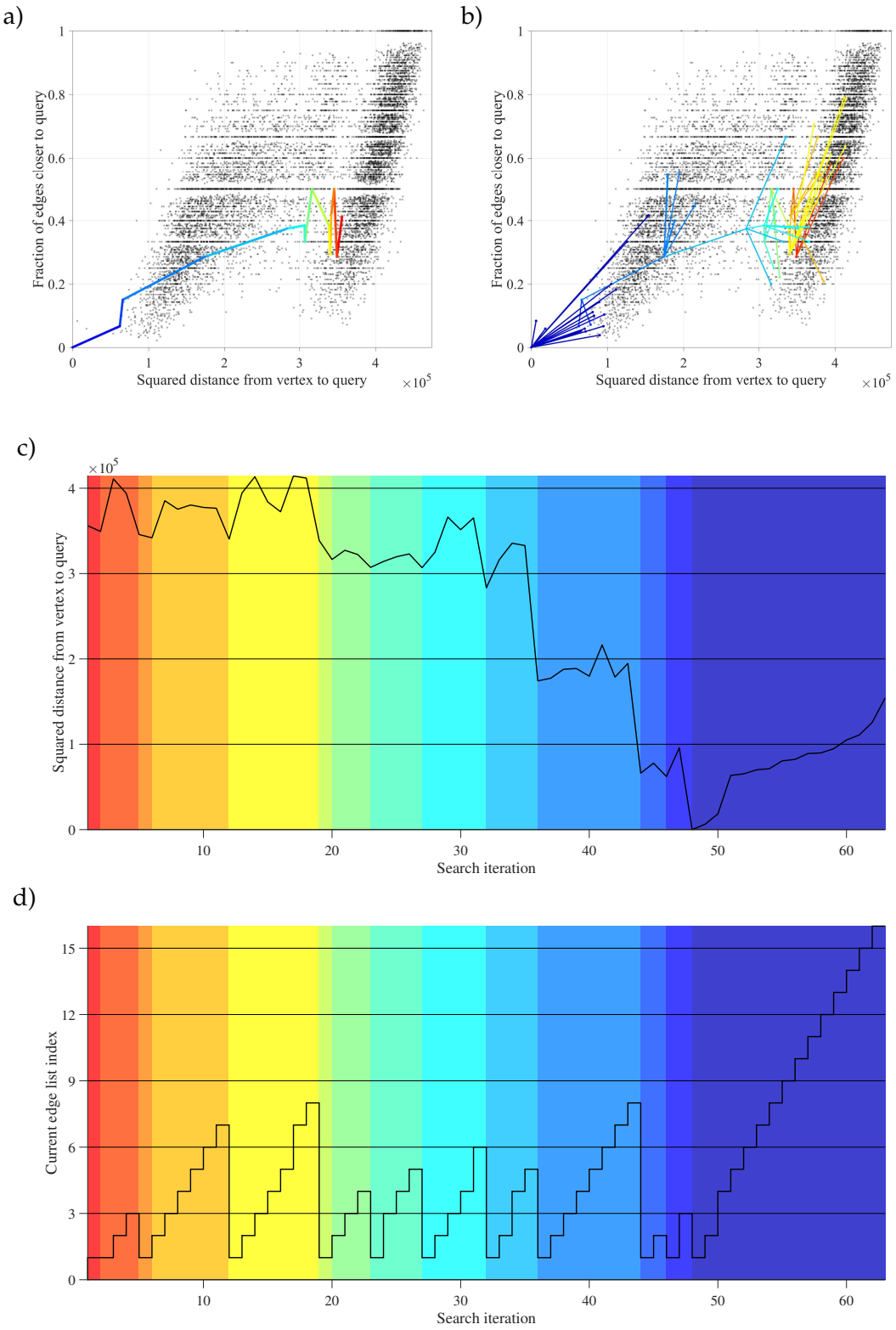


FIGURE 4.8: A greedy downhill search path projected onto a two dimensional parametrisation of ten thousand SIFT vertices. a) Search path towards an internal query. b) All evaluated edges for the internal query. c) Search progress after each evaluated edge. b) Edge list index of each evaluated edge.

the query vector and each of the vertices that were evaluated during the search. A significant portion of the search is spent making frequent incremental progress until later when three large improvements are found. Finally, the global minima is reached at iteration 48 and a large number of comparisons are used to exhaust all of the outbound edges and guarantee that it is a true minima. This larger set of evaluations is typical for the cost of exploring each vertex during an exhaustive downhill search. Figure 4.8d plots the corresponding edge list index for each of the distance calculations shown in the previous plot. Typically the search does not need to evaluate many edges before a better vertex is found and a new edge list is evaluated. Occasionally, such as iteration 18, an edge list contains a vertex that has already been evaluated and so that index is skipped. Overall, the greedy downhill search is able to move quickly though the indexing graph by making only a few comparisons at each vertex. Visiting more vertices along the search path ends up being more efficient than paying a large search cost at fewer vertices. In other words, despite having a longer average path length the weighted path length of a search will be shorter on average for a greedy downhill search that it is for an exhaustive downhill search.

4.3.4 Random Restarts

The graphs we have defined will always contain enough edges to ensure that a downhill search can reach an internal query from any other vertex. However, external queries are samples from the data space that were not considered during the edge selection process. For these queries, we have seen that a downhill nearest neighbour search can become stuck in local minima with no edges directed towards the global optimum. Since these local minima place a limit on the maximum recall that we can achieve with our graph, we would instead prefer to use additional computations to escape the minima and boost the recall. Random restarts are a commonly used strategy for escaping local minima when searching both graph and tree structures.

For a greedy downhill search with random restarts we would like to begin the search from a variable number of vertices throughout the graph. If we sequentially run r repeats of the search then the worst case complexity will be $O(rd_w m)$. We

start each repeat from a random vertex in the graph that has not been checked during a previous search iteration. Similarly, since subsequent repeats can avoid all previously checked vertices, the weighted diameter d_w is expected to decrease as r increases. An updated diameter can be measured by building a weighted graph with only the unchecked vertices and the edges that connect between pairs of these remaining vertices. For the greedy downhill search, each edge in the updated graph can potentially have a lower cost if earlier edges have been removed from the edge list. It is possible for a large number of repeats to essentially partition the vertices into multiple disconnected subgraphs. This will simply result in search paths that terminate at whatever local minima is reachable within their partition, and so d_w is then the largest diameter across all subgraphs.

Table 4.3 contains the results of greedy downhill searches run with an increasing number of random restarts and on a dataset of ten thousand SIFT vectors using only external queries. These experiments were executed on graphs that are constructed with each of the occlusion rules and using both the largest angle edge list ordering and the shortest length ordering. The results show that additional restarts are able to considerably increase the recall for each of the different graphs. However, the additional restarts also result in a growing computational cost. As expected, the average cost of each additional restart does decrease slightly as more vertices have already been checked. Figure 4.9 presents a logarithmic plot of the same performance numbers, but plotted as speed-up over linear search against error rate. This configuration allows for the trade-off between cost and recall be more clearly seen. As the number of random restarts is increased, the computational gain over a linear search goes down and the error rate drops towards the right-hand side of the plot.

TABLE 4.3: Traversing FANNGs using random restarts.

Restarts	Angular occlusion				Distance occlusion			
	Angle order		Length order		Angle order		Length order	
	R@1	Cost	R@1	Cost	R@1	Cost	R@1	Cost
1	0.69	116.8	0.63	126.7	0.50	47.3	0.48	60.2
2	0.84	230.4	0.82	250.8	0.66	95.1	0.62	116.4
3	0.91	344.2	0.86	373.2	0.72	138.1	0.68	170.7
5	0.97	534.3	0.94	597.8	0.83	229.3	0.81	279.3
10	0.99	960.0	0.99	1108.1	0.94	429.6	0.91	528.1

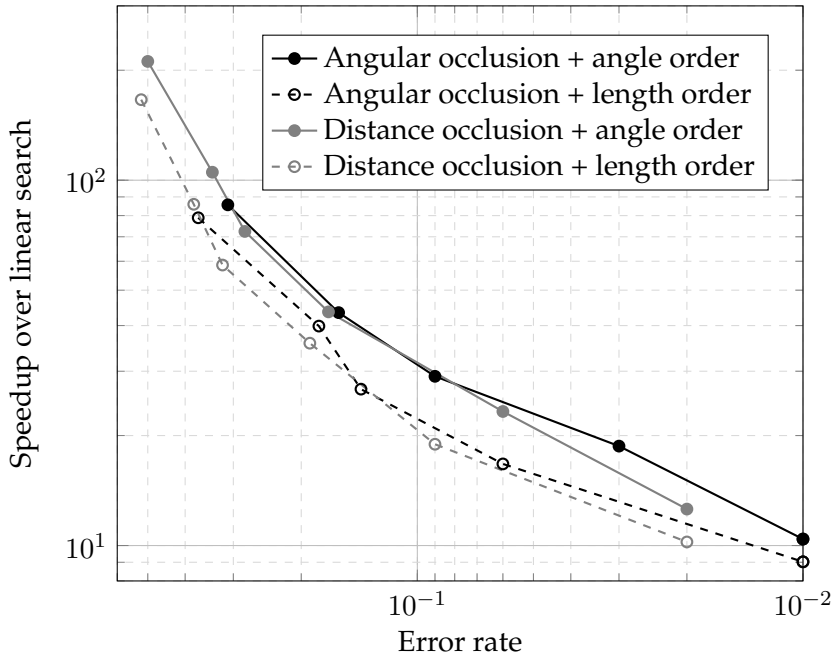


FIGURE 4.9: Search performance trade-off for a variable number of random restarts. Results from between 1 and 10 restarts are shown for each graph type, with additional restarts offering a lower error at the cost of computing more distances.

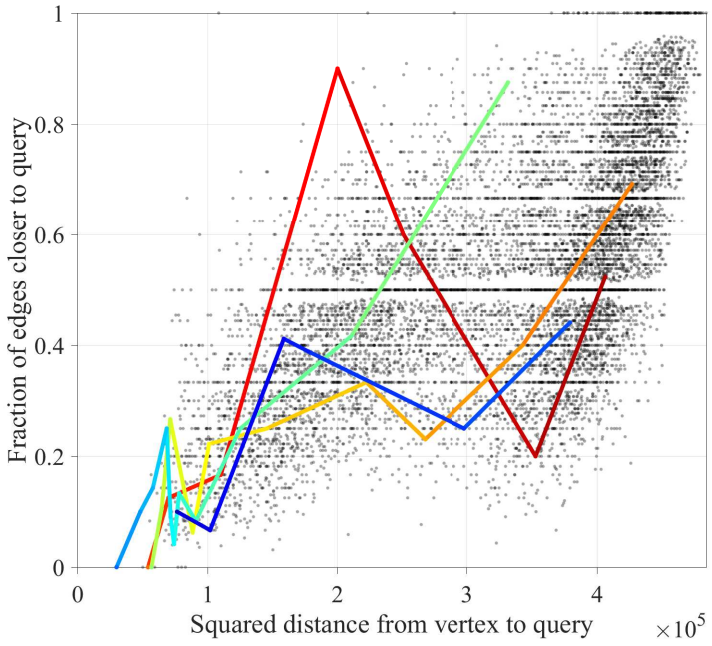


FIGURE 4.10: Exhaustive downhill search paths from random restarts are projected onto a two dimensional parametrisation of ten thousand SIFT vertices. Restarts can escape local minima at the expense of losing considerable search progress.

Each of the curves shows diminishing returns as the number of restarts is increased. For instance, as the speed-up is reduced by one third from 100 to around 30 there is a 40% reduction in error, however, reducing the speed-up by another third from 30 to 10 only yields a 9% reduction in error. Comparing the different graphs types shows that the edge ordering does have a small influence over the trade-off between cost and recall, while the choice of occlusion rule does not.

Lastly, in Figure 4.10 we visualise a search path of the same external query that failed to find the global minimum with the exhaustive downhill search in Figure 4.7c and 4.7d. Four consecutive restarts are shown as overlaid paths on the same plot. Because edges to previously checked vertices are ignored, it is possible for later search paths to terminate above the x-axis as new local minima are created. With each new search path, there is a high probability that the search will begin significantly further from the query than the local minima that prompted the most recent restart. As such, random restarts neglect to take full advantage of progress that has already been made towards finding the local neighbourhood of a query. Additionally, since there is no way of verifying when a global minima is found, any remaining repeats will still be executed and add to the overall search cost.

4.3.5 Greedy Backtracking Search

A less naive approach for escaping local minima during a nearest neighbour search is to backtrack along the previously explored search path. Backtracking searches have already been shown to be an effective way to achieve higher recall when exploring indexing trees and graphs. Commonly a priority queue is used to maintain a sorted list of each vertex that has previously been seen during the search. The queue is sorted by ascending distance between each vertex and the query vector. With this configuration, each search iteration involves removing the first member of the queue, evaluating each of its neighbours and then adding the newly evaluated vertices to the queue. While this approach to backtracking search is building upon the exhaustive downhill search, we would rather make use of our more efficient greedy downhill search. Algorithm 4.3 describes our greedy backtracking search in full. The priority queue P at line 1 stores each vertex as the next unchecked edge list

entry. Newly checked vertices are added to the priority queue with the first edge in their edge list. And then after a vertex is removed from the head of the queue it will be added back with the next edge in its edge list, or if the edge list is exhausted then the vertex can be discarded.

Algorithm 4.3: Greedy backtracking search

Input: graph vertices V
graph edge lists E
starting vertex U_0
query vector $\mathbf{q}$
maximum vertices to visit U_{max}

Output: index of nearest neighbour vertex $best$

```
1  P ← empty priority queue
2  U ← empty set of visited vertex indices
3  best ← {U_0, dist(U_0, q)}
4  P.enqueue(E[start].head(), best.dist())
5  U.add(U_0)
6  while |P| > 0 and |U| < U_max do
7      e, sourceDist ← P.dequeue()
8      if e.to() ∉ U then
9          newDist ← dist(V[e.to()], q)
10         if newDist < best.dist() then
11             best ← {e.to(), newDist}
12         P.enqueue(E[e.to()].head(), newDist)
13         U.add(e.to())
14     if E[e.from()].next() ≠ null then
15         P.enqueue(E[e.from()].next(), sourceDist)
16 return best
```

Whenever a local minima is found by the greedy backtracking algorithm, the search will continue to explore from the closest unexplored vertex that is still in the priority queue. If that minima is in fact the global minima, then the k-nearest neighbours for that query will be approximated as the local neighbourhood is expanded. In order to bound the search time, a maximum limit U_{max} is placed on the number of distances that will be computed during a search. Once all of this computation has been exhausted, the search is immediately terminated and the closest observed vertex is returned. Controlling the search cost via U_{max} allows for management of the trade-off between recall and computational cost. Because U_{max} places an upper bound on the number of distance calculations during a search and is also bounded by the number of vertices n, the worst case computational complexity for the greedy

backtracking search is simply $O(nm)$. For the average case, Section 4.5.4 examines the relationship between U_{max} and n.

Figure 4.11 shows the trade-off as greedy backtracking searches are run on a dataset of ten thousand SIFT vectors over a range of different search costs and using external queries. Results were collected for each of the occlusion rules and using both the largest angle edge list ordering and the shortest length ordering. The best results from the greedy downhill search are also shown for comparative purposes. Compared to the previous search strategies, the greedy backtracking search demonstrates a consistently higher speed-up across a wide range of error rates. In contrast to the greedy downhill search results in Figure 4.8, it is now the occlusion rule rather than the edge list ordering that has the most impact on the search performance. Additionally, because the distance occlusion graph is now found to be outperforming the angular occlusion graph, it is likely that the greedy backtracking search is able to take advantage of the lower average vertex degree in the distance occlusion graphs. This concept is explored further in Section 4.4.2.

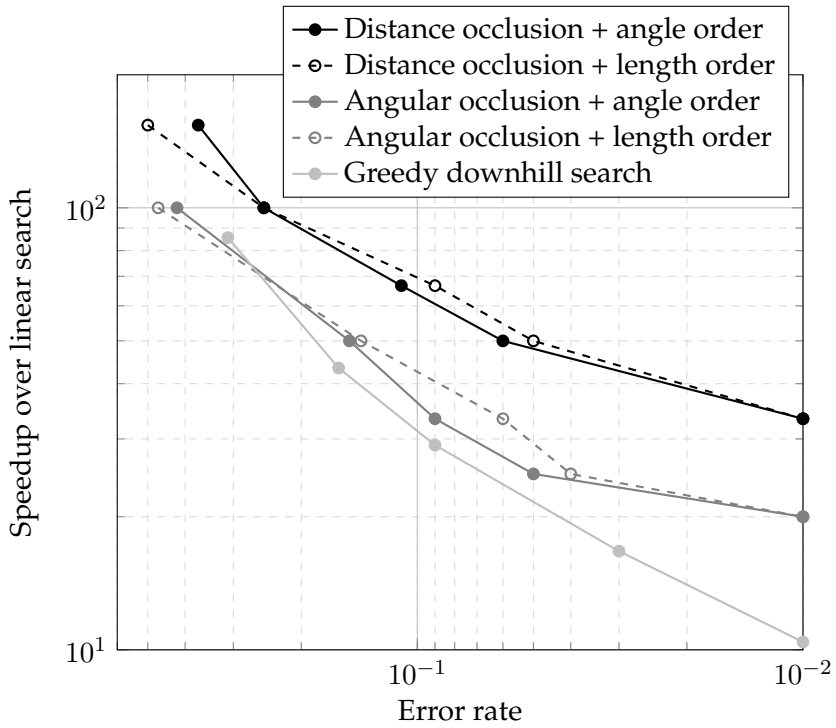


FIGURE 4.11: Search performance trade-off for greedy backtracking search with various graph structures. Almost all results exceed the best performance achieved with greedy downhill search.

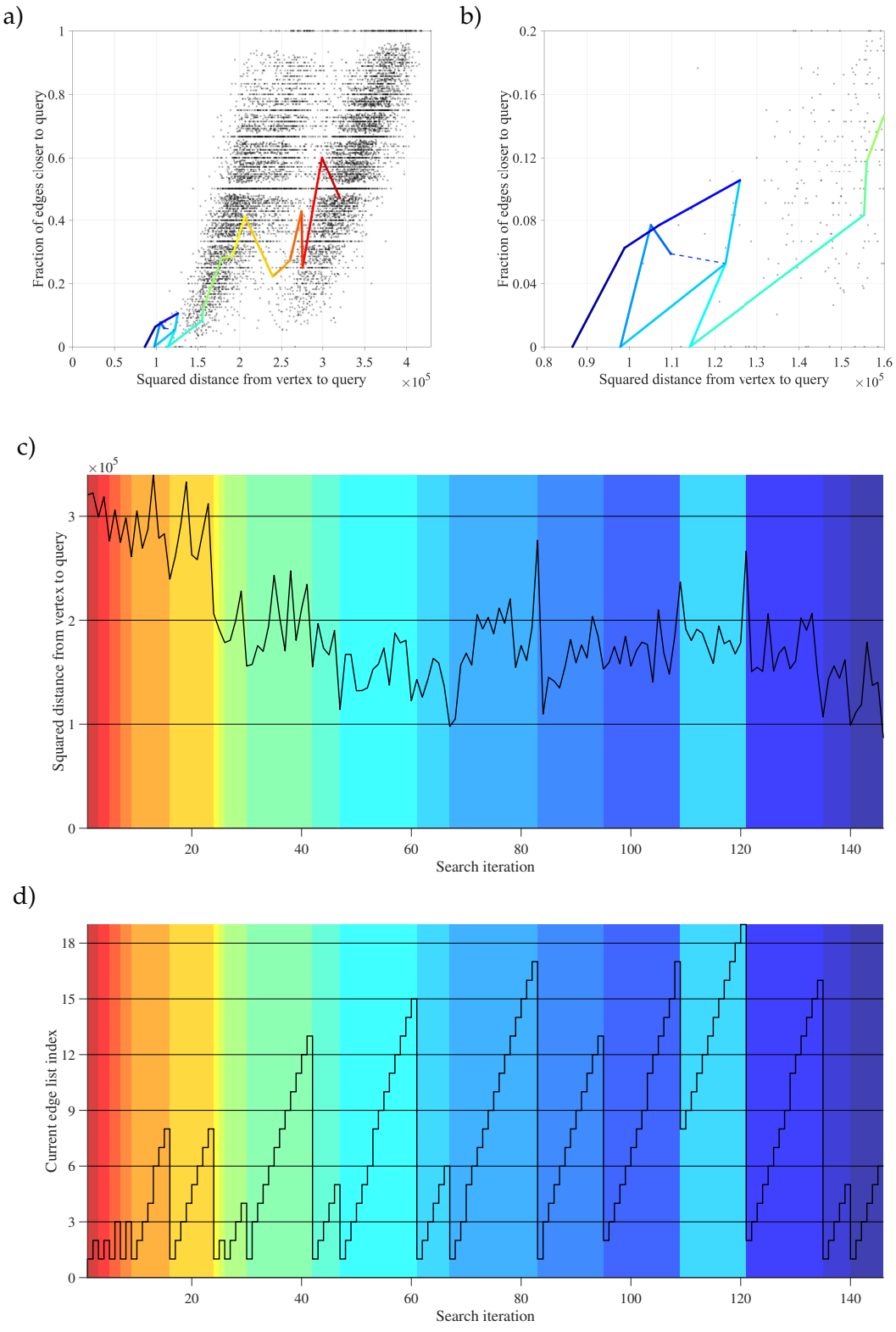


FIGURE 4.12: A greedy backtracking search path projected onto a two dimensional parametrisation of ten thousand SIFT vertices. a) Search path towards an external query. b) Backtracking steps to escape local minima. c) Search progress after each evaluated edge. b) Edge list index of each evaluated edge

In Figure 4.12 we visualise the progress of a greedy backtracking search starting from a random vertex and finishing when it reaches a difficult external query. In this case, the search is not considered to be a path or even a walk because the backtracking steps are able to instantaneously return the search to a previously checked vertex. Figure 4.12b shows an enlargement of the search progress from around iteration 40 until completion. Unlike the random restarts seen in Figure 4.10, the backtracking search is able to remain a short distance from the query while it expands out from previously seen vertices. Considering the additional information presented in the plots of Figure 4.12c and 4.12d, we can see that at iterations 60, 83 and 95 an edge list has been exhausted without a closer vertex being found and so the search continues from the neighbour that was found closest to the query. Then when an edge list is exhausted at iteration 108 the search instead resumes from a previously visited vertex that still has some unexplored neighbours. Because this step does not correspond to following an edge of the graph, it is represented in Figure 4.12b as a dashed line. Iterating through the edge list of the previously visited vertex is resumed from where it had stopped at iteration 67. Once the edge list of the revisited vertex is also exhausted, its neighbour from iteration 62 becomes the closest vertex in the priority queue. From here there is now a path to the nearest neighbour of the query that will avoid all local minima and could be followed using just a greedy downhill search.

4.4 Constructing FANNGs

The occlusion rules from Equations 4.2 and 4.4 give rise to interesting graph structures that are the topic this chapter. These graphs are formed by defining a list of outbound edges to represent a local neighbourhood around each vertex. A naive approach for constructing the edge list $E[i]$ of a given vertex v_i is to first build a complete list of edges E_{max}, from v_i to each other vertex. The edges in E_{max} are stored in ascending order of edge length so that the shorter edges are considered before the longer ones. Once constructed, iteratively comparing each edge in E_{max} to each edge in $E[i]$ will determine which edges from E_{max} are occluded and which can be added to $E[i]$. Initially the list $E[i]$ will be empty, so the first edge in E_{max} will

always be unoccluded. On subsistent iterations as more edges are added to $E[i]$, the longer edges in E_{max} become increasingly more likely to be occluded by at least one of the shorter edges in $E[i]$. A full overview of the naive graph construction is given in Algorithm 4.4.

Algorithm 4.4: Naive graph construction

Input: graph vertices V
Output: graph edge lists E

```
1  foreach v_i ∈ V | 1 ≤ i ≤ |V| do
2      E_max ← empty edge list
3      foreach v_j ∈ V | 1 ≤ j ≤ |V| ∧ i ≠ j do
4          E_max.insert(edge(j, dist(v_i, v_j)))
5      E[i] ← empty edge list
6      foreach e_1 ∈ E_max do
7          occluded ← false
8          foreach e_2 ∈ E[i] do
9              if e_2 occludes e_1 then
10                 occluded ← true
11                 break
12         if not occluded then
13             E[i].append(e_1)
14     E[i].sort()
15 return E
```

For a graph of n vertices, Algorithm 4.4 must perform n iterations of the outer loop at line 1. Each iteration involves building a sorted list of $n-1$ elements, iterating through the list and comparing each element to a growing set of unoccluded edges and then lastly performing an optional reordering of the edges at line 14. By default the final edge lists will be sorted in ascending edge length order. The computational complexity of building the sorted list is $O(n \log n)$ and because the list is sorted by distance, there is an additional cost from the dimensionality m for each insertion. If none of the edges are found to be occluding each other, then a total of $1/2(n-1)n$ unique comparisons will be needed and each requires a distance calculation. This gives a complexity of $O(n^2 m)$ which in practice will only apply for the degenerate case where the vertices are the n corners of an $(n-1)$-simplex. In all cases, each possible edge will undergo a number of comparisons equal to the final degree of the vertex. As such, a tighter bound on the occlusion complexity is $O(nE_{avg}m)$ with E_{avg} being the average vertex degree across the entire graph. Similarly, the reordering of the

unoccluded edges will have a cost of at most $O(E_{avg}^2 m)$. Assuming $E_{avg} \geq \log n$, the naive graph construction is dominated by the occlusion checks with an overall complexity of $O(n^2 E_{avg} m)$. In practice, graph construction time can also be reduced by parallelising the work load. Construction is embarrassingly parallel as each iteration of the outer loop can be computed independently. For each independent thread, the space complexity is $O(n)$ for storing the distance from v_i to each other vertex and the indices that form the sorted list C. When completed, the space complexity for storing the entire graph is $O(n(E_{avg} + m))$ due to the cost of both the edges and the vertices.

We are interested in using fast implementations of the occlusion rules, as the checks at line 9 of Algorithm 4.4 are a significant component of the graph construction times. For the distance occlusion rule in Algorithm 4.4, each comparison can be optimised simply by comparing squared Euclidean distances in order to avoid computing square roots. If sufficient memory is available then construction can also be accelerated by pre-computing all pairwise distances. While this optimisation results in a space complexity of $O(n^2)$, it also reduces the computational complexity to $O\big(n^2(E_{avg} + m)\big)$. For the angular occlusion rule in Equation 4.2 it is difficult to avoid the cost of computing the dot products, however the comparison can still be rearranged and then squared to give

$$(v_j - v_i).(v_k - v_i) > \frac{1}{2} \|v_j - v_i\| \|v_k - v_i\| \tag{4.5}$$

$$(v_j - v_i).(v_k - v_i) \left|(v_j - v_i).(v_k - v_i)\right| > \frac{1}{4} \|v_j - v_i\|^2 \|v_k - v_i\|^2. \tag{4.6}$$

This configuration avoids the cost of inverting the distances and also removes the need to square root them so that precomputed squared distances can still be used. When squaring each side of the equation it is important to preserve the sign on the left-hand side so that the relationship still holds.

4.4.1 Intrinsic Dimensionality and Vertex Degree

In Section 4.1 we saw that using our occlusion rules with a graph construction method such as Algorithm 4.4 has the favourable property of creating graphs with relatively low average vertex degree. For a small dataset of SIFT data, the average vertex degree has remained in the range of 10 to 50 neighbours despite these vertices sitting in a 128 dimensional space. In contrast, the k-nearest neighbour graphs discussed in Section 3.4 require neighbourhoods with hundreds or even thousands of edges in order to operate effectively. Shown again in Figure 4.13, the analysis we presented in Section 2.3.4 demonstrates that the intrinsic dimensionality of SIFT data is much lower than the number of extrinsic dimensions needed to represent the data. As such, we are interested in the relationship between the low vertex degree of our graph structures and the low intrinsic dimensionality of the data. It is possible that occluded outbound edges are pruned so that the remaining edges approximate a span of the local intrinsic neighbourhood around a vertex.

To observe the impact of intrinsic dimensionality on the average vertex degree of our graphs, several datasets of random vectors were generated with increasing extrinsic

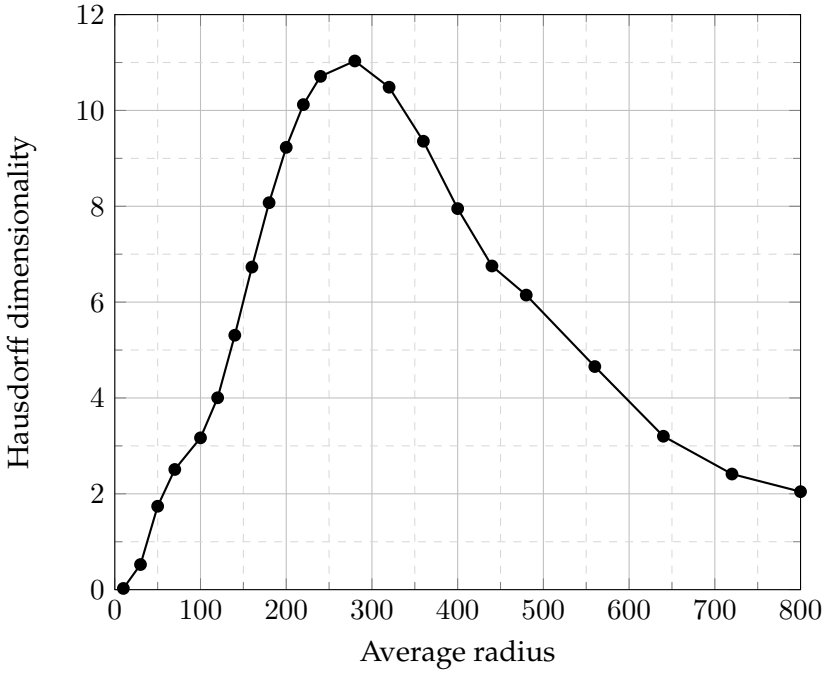


FIGURE 4.13: Hausdorff dimensionality of one million SIFT vectors. The peak dimensionality is measured to be approximately 11.

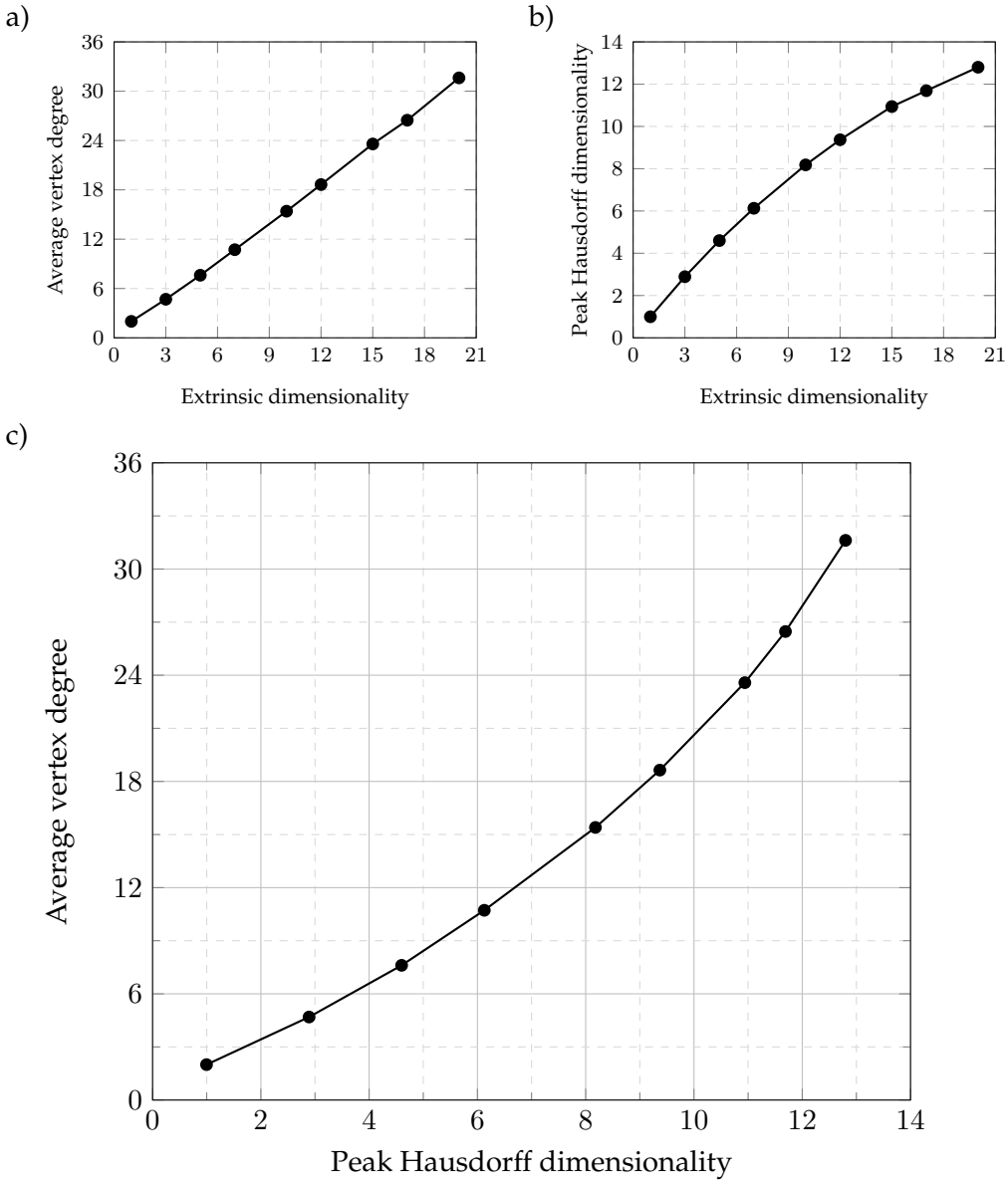


FIGURE 4.14: Relationships between the average out-degree of indexing graph vertices and the dimensionality of synthetic datasets. Each dataset contains samples generated uniformly at random from within an m-cube.

dimensionality by sampling uniformly from within an m-cube. These datasets were then indexed using the distance occlusion rule given in Equation 4.4. Figure 4.14a plots the roughly linear relationship between the length of the random vectors and the average degree of vertices in the index. Since we are attempting to fill these vector spaces with random samples, we expect that the intrinsic dimensionality should roughly follow the full dimensionality of the space. However, we have previously seen that the number of samples in a dataset can impact the intrinsic dimensionality of the data. So, as the number of extrinsic dimensions increases it will become harder for random samples to fill the space. We observe this property in Figure 4.14b with the peak Hausdorff dimensionality measured for each dataset. Constant increases in the number of extrinsic dimensions results in diminishing increases for the intrinsic dimensionality. This bias is then also seen in Figure 4.14c where we plot the average vertex degree against the measured intrinsic dimensionality. However, this relationship should still provide a rough guide for the expected degree of graphs constructed on non-synthetic datasets. Unlike random vectors, actual feature mappings will place a bound on the intrinsic dimensionality of their feature spaces. This should then correspond to a bound on the average degree of vertices in our indexing graph. We see that Figure 4.14c suggests an intrinsic dimensionality of around 11 produces an average vertex degree of about 24, which is within the expected range that we have seen for our graphs.

4.4.2 Variable Out-Degree

Average vertex degree has had a significant influence on the maximum recall for downhill searches as well as the computational cost of backtracking searches. Extending our initial edge selection rules to allow for a variable vertex degree will provide a means of tuning these search properties. The angular occlusion rule in Equation 4.2 produced graphs with a higher average vertex degree than those from the distance occlusion rule. To lower the average degree, we can modify the angular rule so that each edge covers a larger occlusion region. One possible formulation of

this is to exclude edges when

$$\frac{(v_j - v_i).(v_k - v_i)}{\|v_j - v_i\| \, \|v_k - v_i\|} > \cos(\pi/3 + t). \tag{4.7}$$

Instead of just excluding edges within an angle of $\pi/3$ of a shorter edge, a variable angle of t is used to extend the occlusion region. An example of this stronger rule can be seen in Figure 4.15. In contrast to the edges seen in Figure 4.1, $e(v_i, v_n)$ now prevents the inclusion of $e(v_i, v_j)$. As such, $e(v_i, v_k)$ is then unoccluded and will form part of the final graph. Overall, we expect that larger occlusion regions will result in a lower average vertex degree.

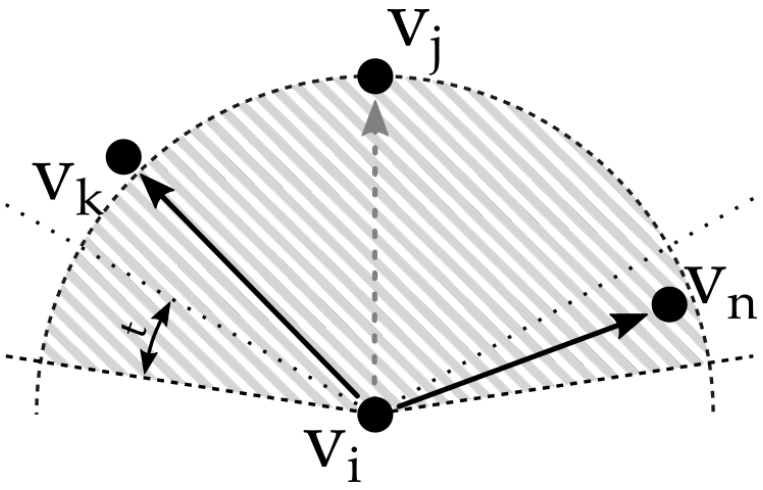


FIGURE 4.15: A simple example of the variable sized occlusion region defined by Equation 4.7. Larger occlusion regions will typically result in fewer total edges.

Table 4.4 presents a summary of ANN search results from graphs constructed with increasingly large values of t. Each graph is constructed on a dataset of ten thousand SIFT vectors and the edge lists all retained the natural ordering of ascending edge lengths. As expected, the additional parameter t is effective at tuning the number of edges included in a graph. As t increases the average vertex degree rapidly drops towards that of the distance occlusion rule and then slows as it approaches zero. The greedy downhill searches for internal queries show that when $t > 0$, the removal of edges that would be selected by Equation 4.2 will also remove the guarantee of avoiding local minima during a downhill search. For both the internal and external queries, larger values of t result in shorter downhill searches as the increasingly

TABLE 4.4: Traversing graphs with increasingly strict occlusion rules.

Occlusion threshold (t)	Average degree (E_{max})	Greedy Downhill Search				Greedy Backtrack Search			
		Internal		External		Internal		External	
		R@1	Cost	R@1	Cost	R@1	Cost	R@1	Cost
0	49.9	1.00	121.2	0.63	126.7	0.87	120	0.61	120
5	22.9	0.86	74.0	0.48	73.1	0.96	120	0.75	120
10	11.8	0.50	42.0	0.36	44.1	0.91	120	0.73	120
15	6.8	0.15	24.9	0.20	26.5	0.73	120	0.70	120
20	4.3	0.10	17.3	0.03	17.1	0.65	120	0.60	120
25	3.0	0.01	9.5	0.01	9.9	0.42	120	0.50	120

abundant local minima are found faster. Decreasing recall is also seen due to the additional local minima, while lower search costs are a combination of shorter search paths and fewer edges towards each local minima. The variable t is providing a trade-off between recall and search cost, and as such, the performance of the greedy backtracking search can be compared using a fixed number of search iterations. As was previously seen, the backtracking search is able to use additional computations to escape local minima and achieve higher recall results than a downhill search. Again we see that a lower average vertex degree can facilitate a more efficient search, however this benefit appears to have a limit. At $t = 5$, the backtracking search reaches a maximum recall performance that outperforms both the higher and lower degree graphs.

4.4.2.1 Exact Nearest Neighbour Search

The distance occlusion rule in Equation 4.4 can be used to build ANN graphs for an arbitrary query vector, however a downhill search is only guaranteed to find the nearest neighbour when using internal queries. We have also seen that both modifying the occlusion rule or using a backtracking search will provide a trade-off between recall and search cost. While Equation 4.7 reduces the number of edges in a graph, here we formulate a modified distance occlusion rule that increases the number of edges in order to guarantee that external queries will avoid local minima when using a downhill search.

For the fixed radius nearest neighbour problem from Equation 3.1, let us assume that

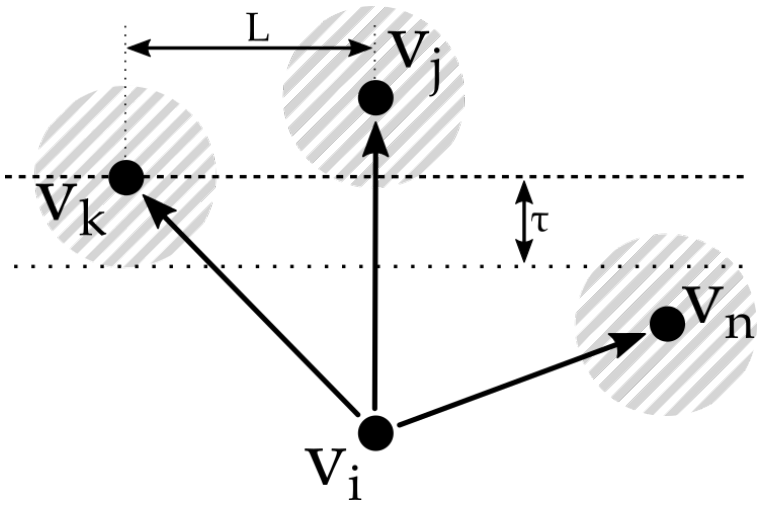


FIGURE 4.16: A simple example of exact nearest neighbour regions. All queries that fall within the shaded regions need to find their nearest neighbours with a downhill greedy search.

true neighbours will have a pairwise distance that is less than some limit τ. As such, we would like to give a guarantee of 100% recall when using a downhill search on a query that is within τ distance of its nearest neighbour. Figure 4.16 illustrates this for a value of τ such that the hyperplane bisecting edge $e(v_i, v_j)$ is tangential to the hyperspherical region around v_k. In this configuration, any query vector within τ distance of v_k must be nearer to v_j than v_i and so the edge $e(v_i, v_k)$ is unnecessary for a downhill search. If we then consider a case where v_k is moved slightly closer to v_i, part of the nearest neighbour region around v_k would then be closer to v_i than it is to v_j. Now a downhill search for a query in this part of the region will not follow the edge $e(v_i, v_j)$ and if the edge $e(v_i, v_k)$ is not present, then v_i will be a local minimum. This indicates that v_k sits at the limit of where the occlusion boundary between v_i and v_j should be moved in order to prevent local minima from prematurely halting these searches.

Guaranteeing that a downhill search will find nearest neighbours within a distance of τ from any vertex requires moving each occlusion boundary by a distance of τ. As can be seen in Figure 4.16, using Euclidean distance and the Pythagorean theorem then gives

$$dist(v_i, v_k)^2 = \left(\frac{1}{2} dist(v_i, v_j) + \tau\right)^2 + L^2 \tag{4.8}$$

and

$$dist(v_j, v_k)^2 = \big(\frac{1}{2}dist(v_i, v_j) - \tau\big)^2 + L^2 \tag{4.9}$$

which through subtraction yields

$$dist(v_i, v_k)^2 - dist(v_j, v_k)^2 = 2\tau dist(v_i, v_j). \tag{4.10}$$

Rearrangement then provides an inequality for the updated occlusion boundaries, and so the modified criteria for occluding edges can we written as

$$dist(v_j, v_k)^2 < dist(v_i, v_k)^2 - 2\tau dist(v_i, v_j). \tag{4.11}$$

This now ensures that an edge $e(v_i, v_j)$ will only occlude an edge $e(v_i, v_k)$ if it is guaranteed that there are no queries where $dist(\mathbf{q}, v_k) < \tau$ and $dist(\mathbf{q}, v_i) < dist(\mathbf{q}, v_j)$.

Table 4.5 summarises the results of greedy downhill searches on graphs of 100 thousand SIFT vectors that are built using the modified occlusion rule in Equation 4.11 and with various values of τ. Each τ is chosen to be a fraction of τ_{max}, which has been calculated as the largest distance between any two nearest neighbours in the indexed dataset. This value of τ_{max} is based on the assumption that the nearest neighbour distances in the dataset are representative of the distances between queries and their closest neighbours. The recall results verify that the less restrictive occlusion rule is capable of removing local minima and enabling downhill searches that achieve perfect recall on sets of external queries. We also see that building graphs in this way has significantly increased the average vertex degree. Higher degree vertices will result in an increased space complexity for storing a graph and an increased search time from additional edges at each step along a search path.

TABLE 4.5: Search performance of variable degree graphs.

τ/τ_{max}	Average degree	Recall	Average query cost
0.00	24.3	0.710	110.5
0.38	127.5	0.998	360.0
0.50	232.0	0.999	547.8
0.75	753.5	1.000	1315.5
1.00	2182.9	1.000	3123.2

4.4.2.2 Undirected Edges

Section 4.1.1 introduced the undirected graph structure called Relative Neighbourhood Graphs (RNGs). Despite being designed for low dimensional data, RNGs can still be constructed in higher dimensions due to an edge selection rule that has similarities to our occlusion rules. We are interested in exploring whether these similarities will also give RNGs a structure that is effective at indexing high dimensional data for ANN searches. Additionally, we use two alternative methods to produce an undirected graph from our own directed structures. Our first method is to convert each directed edge into an undirected edge by adding an additional edge in the opposite direction to each existing edge. These edges only need to be added if there is not already an edge in both directions between two vertices. Our second method is to remove any edge that does not already have a corresponding edge that travels in the opposite direction.

In Table 4.6 we evaluate the performance of each of the undirected graphs and compare them to a directed graph that is constructed using the distance occlusion rule. Each of the graphs is constructed on a dataset of ten thousand SIFT feature vectors and the edges are ordered by ascending edge length. As should be expected for our two undirected graphs, the method that is adding edges results in a higher average vertex degree and removing edges causes a lower average. The RNG has the lowest average vertex degree and this can be explained by the more restrictive edge selection rule for constructing this graph. Looking at the ANN search performance, the edges in the RNG are not sufficient for avoiding local minima during a downhill search. The effect of local minima can also be seen in our graph that has had some of its edges removed. While the guarantee of finding internal queries is maintained

TABLE 4.6: Search performance of graphs with undirected edges.

Edges	Avg. degree (E_{avg})	Greedy Downhill Search				Greedy Backtrack Search			
		Internal		External		Internal		External	
		R@1	Cost	R@1	Cost	R@1	Cost	R@1	Cost
Directed	15.3	1.00	86.1	0.43	79.5	1.00	53.3	1.00	99.9
Edges added	19.6	1.00	88.7	0.51	85.0	1.00	49.3	1.00	94.3
Edges removed	6.9	0.26	39.5	0.14	36.1	1.00	74.1	1.00	96.0
RNG	4.9	0.12	25.5	0.05	23.8	1.00	89.5	1.00	117.5

for the undirected graph with additional edges, the higher average vertex degree results in a larger computational cost when compared to the directed graph.

4.4.2.3 Truncated Edge Lists

Sections 4.4.2 and 4.4.2.1 have both demonstrated that vertex degree has a significant influence on the query cost of ANN searches. If the average vertex degree E_{avg} is too high, then exploring the larger neighbourhood at each visited vertex results in a higher overall search cost. Lower degree graphs generally offer better search performance, however if E_{avg} is too low then additional backtracking can greatly increase the number of vertices visited during each search. A simple method for reducing the average vertex degree of a graph is to truncate the edge list of each vertex to a limit of T edges. Previously, Figure 4.3 illustrated that despite having an average vertex degree of around 50, the maximum vertex degree of the full angular occlusion graph is over 200. Truncating to a lower maximum vertex degree should allow for a reduction in the average query cost of greedy backtracking searches, with minimal impact on the recall rate.

Table 4.7 details the reduction in E_{avg} that occurs with increasing levels of edge list truncation. These values are computed for a dataset of ten thousand SIFT feature vectors, with the truncation being applied to the edge lists of a distance occlusion graph. While the reduced average vertex degree does result in lower search costs, the removal of edges is also seen to introduce local minima into the graphs. For the internal query vectors, even a small reduction in E_{avg} results in a significantly lower

TABLE 4.7: Edge list truncation for lower degree vertices.

Maximum degree (T)	Average degree (E_{avg})	Greedy Downhill Search Internal R@1	Internal Cost	External R@1	External Cost
24	14.0	0.85	53.2	0.10	38.7
20	13.6	0.74	47.7	0.10	35.7
16	12.7	0.56	38.5	0.08	30.0
12	10.9	0.38	28.5	0.03	21.4
8	7.9	0.18	18.8	0.01	13.4
4	4.0	0.02	8.5	0.00	6.7

recall. Additionally, the greedy downhill search is no longer adequate for ANN searches with unseen feature vectors.

Figure 4.17 plots the search performance of the increasingly truncated graphs using greedy backtracking searches with external queries. Because this search algorithm is capable of escaping local minima, this method achieves a significantly higher recall at a given cost when compared to the downhill search. In fact, halving the maximum vertex degree from 24 down to 12 now results in a very small drop in recall. For the greedy downhill search, this same increase in truncation reduced the recall by more than half. Eventually, the backtracking search is unable to efficiently compensate for the increasing number of local minima that are associated with a larger amount of truncation. With just eight edges per vertex, the performance results are moderately impacted, and with only four edges in each edge list the graph has only twice the efficiency of an exhaustive search.

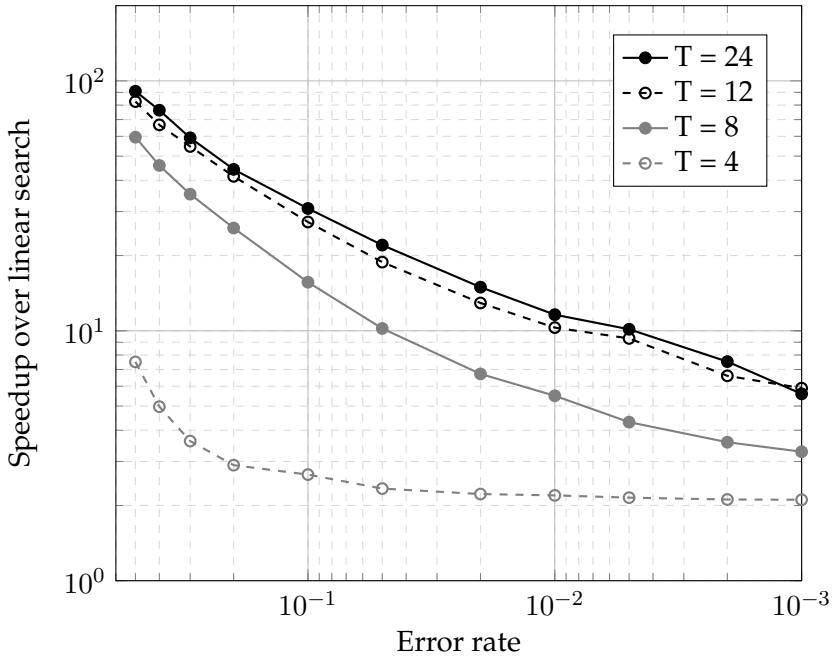


FIGURE 4.17: Search performance trade-off for greedy backtracking search using external queries and with increasing levels of edge list truncation.

Edge list truncation can be applied either during the offline graph construction or dynamically at search time by passing T as an additional parameter to one of the search algorithms from Section 4.3. The major advantage of truncating during graph construction is the reduction in computational and memory costs that are associated

with E_{avg}. Truncating when searching is also beneficial as it allows for a single graph to be adjusted to target various levels of recall. This can be achieved by adding an additional comparison to line 14 of Algorithm 4.3 so that an edge list is not enqueued when e is the T^{th} edge in $E[e.from()]$.

4.4.3 Approximate Graphs

While we are primarily interested in maximising the speed and recall of ANN searches, it can also be beneficial to reduce the complexity of the offline construction of our graphs. This is particularly important for constructing graphs with a larger number of vertices, as the n^2 term can become prohibitively costly. In this section we provide algorithms and analysis for two alternative methods of graph construction. Instead of constructing the exact graphs that are defined by our occlusion rules, each approximate method takes advantage of the stochastic nature of an ANN search. Because each search already includes a trade-off between search cost and recall any differences in the approximate graphs will simply adjust the trade-off curve. Relaxing the need for an exact graph allows us to make use of dynamic programming techniques, since already computed distance information can be shared locally instead of globally. However, we also expect that a larger approximation of the full graph structure will result in a larger degradation of search performance.

Due to the slower spread of information when constructing approximate graphs, it becomes harder to detect duplicate vectors in the dataset. As these graphs are constructed, surrounding vertices will connect to only one vertex in a duplicate set and each of the duplicates will likely have a different edge list. In order to robustly handle the presence of duplicate vectors, we look for coincident vertices every time a new pairwise distance is computed. Whenever a distance of zero is found, all but one of the duplicates is removed from the graph. For consistency and ease of implementations, we always retain the vertex that appears first in the dataset. Removing a vertex from the graph then requires that the edge list at each vertex is searched for possible edges that need redirecting from the removed vertex to the remaining duplicate vertex. In practice, it is faster to maintain a lookup table of duplicates. Initially each vertex points to itself in the lookup table, however when a duplicate is

found the removed vertex can be directed to the remaining one. This table is then used every time the destination of an edge is needed. Instead of simply using the destination vertex stored in an edge list, the lookup table is used to redirect the edge if needed. At the end of the graph construction process, any redirected edges can be stored using their correct destination. As such, the lookup table is not required for traversing a graph during an ANN search, however we still retain the duplicate information so that full duplicate sets can be included in the neighbourhoods of queries.

4.4.3.1 Traverse-Add Graph Construction

Our first method for approximate graph construction initialises a graph of vertices with empty edge lists and then stochastically adds edges that satisfy our local neighbourhood criteria from Equation 4.3. In order to find an edge that will enable efficient traversal of the graph, we first traverse the graph with a greedy downhill search from a random vertex and towards a random internal query. If a local minimum is found during the search, then Algorithm 4.2 will return a vertex that has no edges leading closer to the query. We then insert an edge to the query into the edge list of the vertex that the search terminated at. This new edge will remove the local minima and allow that same downhill search to reach the query. Every time a query vertex is reached by a search there is no need to add an additional edge to the graph. As more edges are added to a graph it becomes more likely that a random downhill search will not encounter a local minima and so changes become less frequent. In order to fully satisfy our neighbourhood criteria it is also necessary to restrict each downhill search to vertices that are within the exclusion region between the current best vertex and the query vertex. This is achieved by adding an additional angle or distance test to line 10 in Algorithm 4.2 so that each visited vertex will be in the shaded regions seen in Figures 4.1a and 4.2a.

Algorithm 4.5 provides an overview of how edges are selected and added to a graph. Graph construction is achieved by repeatedly running the algorithm with different starting and query vertices. We construct the graph for a variable number of iterations p, where each iteration uses each vertex as a starting vertex and query vertex

exactly once. This is implemented with each iteration generating a random permutation of the vertices in order to provide a query vertex for each starting vertex. Each time the edge selection algorithm inserts an edge into to an edge list, the occlusion rule is applied to all longer edges that were already in that edge list. Occluded edges are removed from the graph so that a low average vertex degree E_{avg} can be maintained. In some cases the removal of an edge can reintroduce local minima for a previously successful downhill search. These local minima can be addressed in subsequent iterations of construction, although it is expected that at least some local minima will be present in the finalised approximate graph. Due to the tunable number of construction iterations p and the use of a greedy downhill search, the computational complexity of this method is $O(pn(d_w + E_{avg})m)$ where d_w is the diameter of a weighted graph. These weights are assigned by edge length in ascending order, but other edge orderings can be applied once the graph is finalised.

Algorithm 4.5: Traverse-add edge selection

Input: graph vertices V
graph edge lists E
starting vertex U_0
target vertex V_q
Output: directed graph edges E

$best, d_{best} \leftarrow GreedyDownhillSearch(V, E, U_0, V_q)$
if $best = q$ **then**
  return E
$e_{new} \leftarrow edge(q, d_{best})$
$E[best].insert(e_{new})$
foreach $e \in E[best] \mid e.length() > e_{new}.length()$ **do**
  if e_{new} **occludes** e **then**
    $E[best].remove(e)$
return E

Table 4.8 demonstrates the refinement of an approximate graph for increasing values of p. With a dataset of ten thousand SIFT feature vectors, these approximations of a distance occlusion graph include progressively more edges as the number of iterations is increased. Similarly, the same increase is seen in the percent complete column, which indicates the percentage of downhill searches that reached their query vertex during the most recent construction iteration. We see that the increase in average

TABLE 4.8: Approximate graphs from traverse-add construction.

Build iterations (p)	Average degree (E_{avg})	Percent complete	Greedy Downhill Search Internal R@1	Internal Cost	External R@1	External Cost
25	7.6	15.4	0.17	29.9	0.17	27.2
50	10.6	63.8	0.65	43.0	0.28	39.6
75	11.9	83.5	0.89	52.3	0.47	45.9
100	12.6	90.7	0.91	54.2	0.42	47.5
125	13.0	93.9	0.94	55.0	0.40	48.8
150	13.2	96.1	0.96	55.8	0.42	48.0

vertex degree slows considerably once a large fraction of the searches are completing and new edges have become harder to find. Evaluating the approximate graphs on greedy downhill searches demonstrates once again that the cost of internal downhill searches will increase as the average vertex degree increases.

Figure 4.18 shows the search performance of increasingly refined approximate graphs using greedy backtracking searches with external queries. During the earlier iteration of the graph construction, many new edges are added to remove local minima. The backtracking search is able to compensate for these local minima, and is still

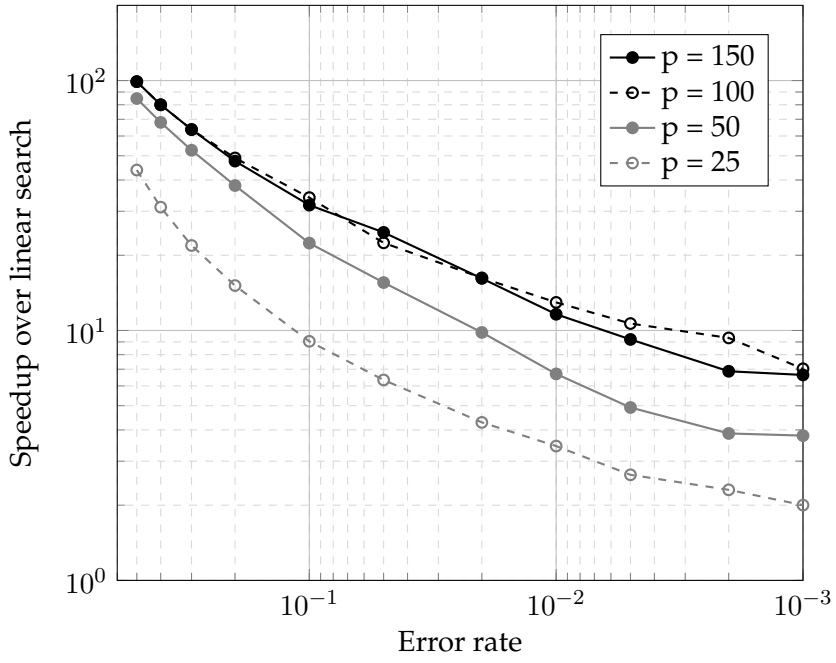


FIGURE 4.18: Search performance trade-off for greedy backtracking search with approximate graphs constructed using increasing traverse-add iterations.

able to achieve high recall results with an incomplete graph. However, the cost of this additional backtracking results in a higher overall search cost. As local minima are removed from the graph, the same recall values can be reached with progressively fewer search iterations. Then, when around 90% of the traverse-add iterations are completing, the advantages of the greedy backtracking search begin to diminish in comparison to the additional cost of higher degree vertices.

By monitoring the percentage of completed searches on each iteration of the construction, it is possible to terminate the building process early once a target percentage has been reached. Experimentally we find that 90% completion will generally provide a good balance between the computational cost of construction and the ultimate search performance of the graph. Lastly, additional gains in efficiency can be achieved by using two simple heuristics when choosing starting and query vertices for the downhill searches. The first is an additional search to ensure that local minima are also avoided in the opposite direction of each edge that is added to the graph. Since the edges are directed, whenever an edge is inserted at line 5 of Algorithm 4.5 an additional search can be run from the query vertex back towards the local minima that was found. The second and the most effective heuristic is an additional search designed to reduce the impact of removing occluded edges. Whenever an edge is removed at line 8 of Algorithm 4.5 an additional search can be executed to ensure that the vertices that where joined by the removed edge can still be reached with a downhill search. Because the additional searches could result in further additional searches, we implement this optimisation with a work list which maintains a set of start and query vertex pairs that are still to be tested.

4.4.3.2 Self-Query Graph Construction

The second method for approximate graph construction begins with a connected graph as an initial starting condition. This pre-built graph is then traversed in order to rebuild a more complete edge list at each vertex. The new edge lists approximate the edge lists of the full graph, and while this method can be applied to any connected graph, closer approximations are produced from graphs that contain edges between local neighbours. As such, several iterations of the traverse-add

graph construction will produce a good starting graph. The edge list for each vertex v_i is then rebuilt by replacing the full candidate list from line 2 of Algorithm 4.4 with an approximate edge list that can be generated with a function call of $GreedyBacktrackingSearch(V, E, v_i, v_i, k)$. The goal of the backtracking search is to expand the local neighbourhood around v_i and return a small candidate set of p approximate nearest neighbours. Once a candidate set has been found, the final edge list can be generated by applying an occlusion rule to remove additional edges. Ignoring the construction of the starting graph this method reduces the computational complexity of Algorithm 4.4 from $O(n^2 E_{avg} m)$ to $O(npE_{avg} m)$.

TABLE 4.9: Approximate graphs from self-query construction.

Search iterations (p)	Average degree (E_{avg})	Greedy Downhill Search Internal R@1	Internal Cost	External R@1	External Cost
0	7.6	0.17	29.9	0.17	27.2
100	9.2	0.41	39.2	0.23	31.2
200	10.8	0.54	43.6	0.32	38.6
300	11.8	0.62	50.2	0.35	42.3
400	12.5	0.74	54.6	0.37	45.0
500	13.0	0.78	58.1	0.41	47.3
600	13.5	0.83	59.6	0.48	48.7

Table 4.9 presents the details of approximate graphs constructed from increasingly large self-queries on each vertex. The initial connected graph was built to index ten thousand SIFT vectors using 25 iterations of the traverse-add construction method. The distance occlusion rule from Equation 4.4 was then applied to all of the candidate edge lists. We can see from these results, that the approximate graphs retain progressively more edges as the number of search iterations is increased. However, unlike the traverse-add method, the increase in average vertex degree continues to grow fairly steadily as the construction cost increases. This is due to the stochastic nature of the traverse-add algorithm, where the frequent addition of shorter edges can remove significant numbers of longer edges. In contrast, the self-query method performs a single occlusion pass, and so longer edges that would occur in a complete graph do not need to be continually rediscovered. Evaluating these approximate graphs on greedy downhill searches offers lower recall and increased costs when

compared to approximate graphs with the same average vertex degree, but produced using the traverse-add method. Again this is likely due to the type of edges that are being found by each construction method.

Figure 4.19 shows the search performance of approximate graphs constructed with increasingly large self-queries, and using greedy backtracking searches on external queries. Using only a small number of search iterations for the graph construction can already provide many of the local edges that help to avoid local minima. Increasing the search size allows for the discovery of longer unoccluded edges that results in denser graphs. While there is some advantage to be gained from these additional edges, the greedy backtracking search quickly becomes hampered by the additional cost of the higher average vertex degree.

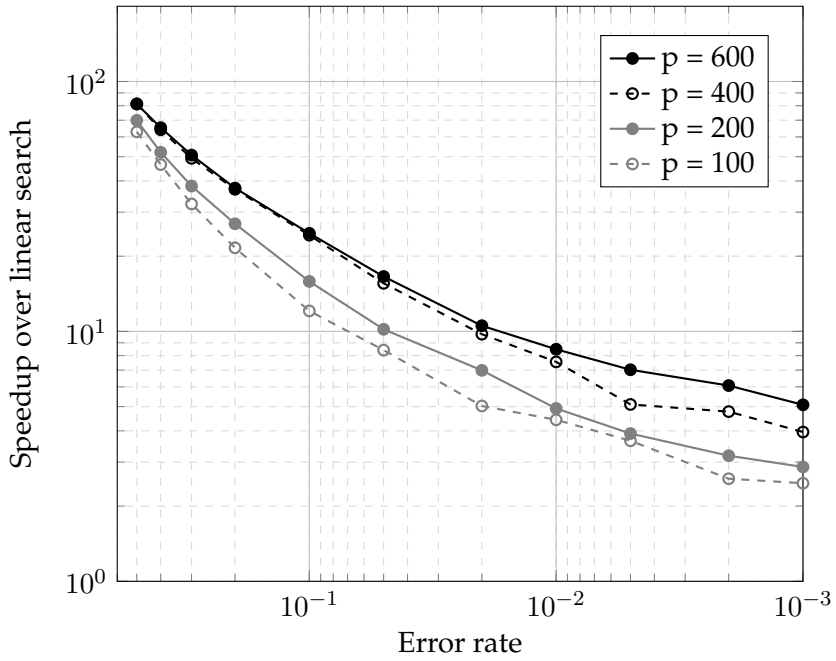


FIGURE 4.19: Search performance trade-off for greedy backtracking search with approximate graphs constructed using increasing self-query search iterations.

Compared to the traverse-add construction method, the self-query construction will produce considerably different approximations of a full graph. While the traverse-add algorithm is designed specifically to find edges that remove local minima, a self-query edge list is likely to contain a dense subset of the corresponding complete edge list. Ultimately, the self-query edge lists are a better candidate for applying the

truncation methods discussed in Section 4.4.2.3. Additionally the numerous searches that are required for the self-query construction can produce their candidate edge lists in parallel without modifying the underlying graph structure. As such, these independent searches are well suited for GPU acceleration.

4.5 Quantitative Results

This chapter has explored many variations of our novel approach for ANN search with high dimensional visual data. We utilise an edge occlusion rule to construct sparse indexing graphs that enable efficient traversal of the data they index. Sparse edges allow the search paths to exploit the local intrinsic structures of a dataset, but without needing to directly compute the lower dimensional manifold that contains the embedded samples. Using a graph structure also minimises the impact of hyperplane boundaries that contribute to the large backtracking costs when exploring indexing trees. Our ordered edge lists and greedy backtracking algorithm further reduce the average search cost by allowing for very few outgoing edges to be evaluated before the search can progress towards a query. Our implementation is able to directly trade-off the computational cost and the expected recall of a search by setting a limit on the number of distance comparisons allocated to finding a particular query.

In this section we evaluate the performance of our graphs, which are constructed using the distance occlusion rule from Equation 4.4 and contain truncated edge lists that are sorted in ascending order of edge length. For larger datasets we construct an approximate graph by repeatedly applying the traverse-add edge selection in Algorithm 4.5 until at least 90% of each iteration provides no additional improvement to the graph. We then finalise our approximate graphs with a single call to the self-query construction method, in order to find a candidate set of the thousand closest neighbours to each vertex. These candidate lists are then reduced using the distance occlusion rule before a final truncation.

4.5.1 Comparative Performance of k-Nearest Neighbour Graphs

Figure 4.20 presents the performance of our full graph and a truncation with $T = 25$. Truncating our graph provides a small gain in search performance, while also reducing the memory footprint. To illustrate the advantages of a sparse edge list we compare our graphs against two k-nearest neighbour graphs. All four of the graphs are constructed on 100k SIFT vectors and are evaluated using our greedy backtracking search algorithm. While the vertices in the truncated graph and the 25-nearest neighbour graph have equal out-degree, the truncated graph is able to include longer edges in the place of shorter occluded edges. As the figure shows, using just the twenty-five nearest neighbours of each vertex lowers the computational efficiency substantially. Even with $k = 250$, the graph is consistently less efficient than our method. The larger k-nearest neighbour graph also highlights the difference in space complexity, with the better performing k-nearest neighbour graph using ten times more memory than our best graph.

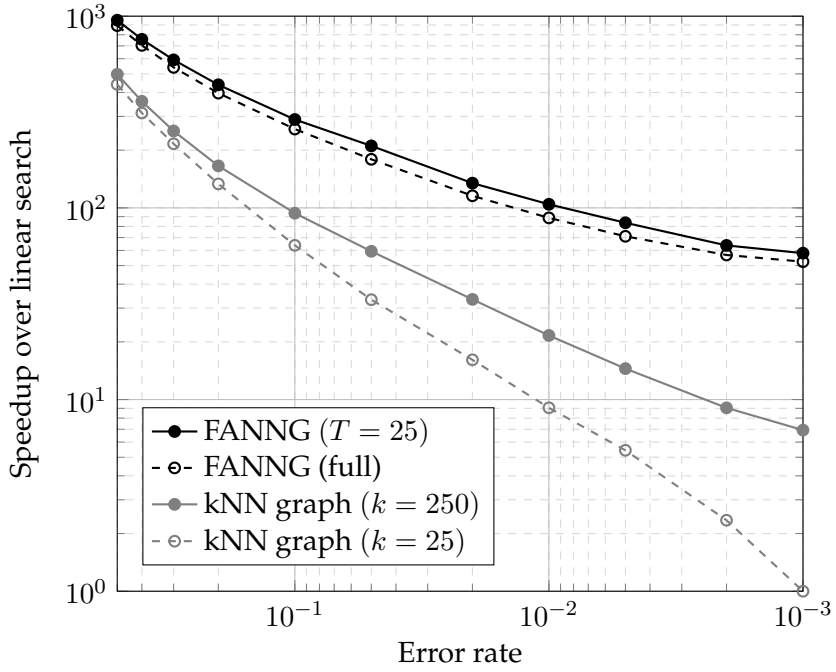


FIGURE 4.20: Comparative performance of k-nearest neighbour graphs and both our full and truncated graphs from the distance occlusion rule.

In contrast to the results in Section 4.4.2.3, we find the efficiency of our queries can increase when truncating graphs that index larger datasets. Specifically, for larger

datasets of SIFT features a maximum degree of between 25 and 32 will result in faster searches at a fixed recall rate. In practise the optimal truncation level does appear to depend on the recall rate begin targeted. Higher recall rates are typically more efficient with slightly higher vertex degrees, while lower recall rates are more efficient with additional truncation. Ultimately, the benefits of truncation amount to only a small percentage difference, and a single truncation value is typically sufficient across a large range of recall values.

4.5.2 Comparative Performance at High Recall

Here we compare our graphs with two state-of-the-art ANN methods that perform well at high levels of recall. With those methods being navigable small world graphs (NSWG) [83] are the k-means trees from FLANN [52]. Results are presented using k-nearest neighbour overlap, which provides the average fraction of the top k ground truth nearest neighbours that are returned in the top k results of each ANN search. This measure requires the ANN methods to efficiently locate multiple samples from the local neighbourhood of each query. Figure 4.21 compares our self-query graph

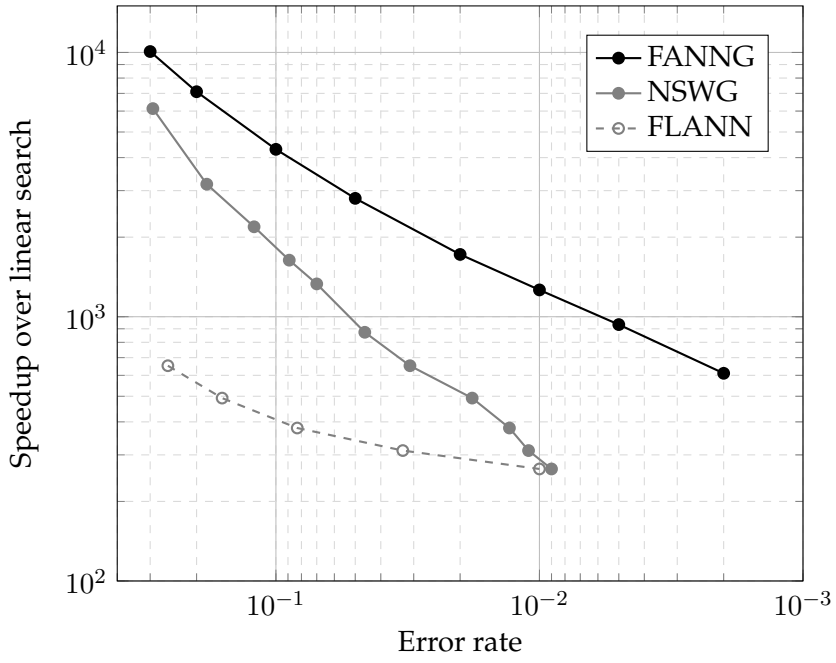


FIGURE 4.21: Comparison of ANN methods using 10-nearest neighbour overlap on a dataset of five million SIFT vectors.

and greedy backtracking method to both FLANN and NSW graphs using 10-nearest neighbour overlap on a dataset of five million SIFT vectors. The performance of our method is seen to surpass the other graphs structures with more efficient queries across a wide range of recall values. The local edge structures that are produced by our graphs are designed to efficiently arrive in the neighbourhood of a query. Our greedy backtracking search is then able to expand this local neighbourhood and return an accurate set of nearest neighbours.

4.5.3 Comparative Performance on a GPU

We evaluate our GPU implementations for both our greedy backtracking search and for an exhaustive linear search. Due to the trade-offs that are present in the GPGPU instruction set, the maximum efficiency of our implementations is limited to specific length features that align with the number of threads in a warp. Additionally, the priority queue at line 1 of Algorithm 4.3 is truncated in order to accelerate the search speed, but at a cost of limiting the maximum achievable recall. This is also due to the merger of this priority queue with the set of visited vertices. As such, we assume that the tests on lines 6 and 8 will only need to see the best set of visited vertices as the queue fills with the final set of neighbours.

For both search methods, we evaluate performance using a GTX Titan X GPU and with a dataset of one million SIFT vectors. Each result in Table 4.10 is generated by batching ten thousand query vectors into a single kernel call and then return timings that are averaged over these queries. These timing results include all overhead costs, such as the cost of maintaining a priority queue. Even with these additional costs,

TABLE 4.10: Comparative performance on a GPU.

Dataset size	Method	Recall	Average query time (μs)
1M	Linear search	1.00	473.7
	FANNG	0.95	1.3
		0.90	0.8
5M	Linear search	1.00	2433.0
	FANNG	0.90	1.7

our method is approximately 600 times faster than the linear search when operating with a 90% recall rate on one million dataset samples and approximately 1400 times faster on a dataset of five million samples. For the one million sample dataset, targeting a nearest neighbour recall of 95% results in search time that is approximately 350 times faster than the linear search.

4.5.4 Performance at Scale

We further evaluate the search performance of our graphs with datasets of increasing size and with varying dimensionality. Figure 4.22 plots the number of distance calculations needed to consistently achieve particular recall rates as the size of a dataset increases. Each set of feature vectors contains between 100k and 20M SIFT vectors. From the results, we observe that each of the curves are approximately linear on the log-log plot. Additionally, a very similar slope is also seen at each of the evaluated recall levels. As such, we conclude that our method scales with a power law complexity that is determined by this slope. We find that the cost complexity, and hence the time complexity, of our method is given by $O(n^{0.2})$.

Figure 4.23 compares the performance of our graphs on two different types of high dimensional visual data. For this comparison we use datasets of one million SIFT vectors and one million GIST vectors. The GIST vectors, which can summarise entire images, have 7.5 times the number of extrinsic dimension found in SIFT vectors, which summarise local image regions. As such, we find that both the intrinsic dimensionality and the average vertex degree are larger for the GIST dataset. The resulting plots indicate that searching this high complexity index is consistently less efficient. The computational cost of the ANN queries remains consistent across a wide range of recall values, with the SIFT index remaining around three times more efficient than searches on the GIST index.

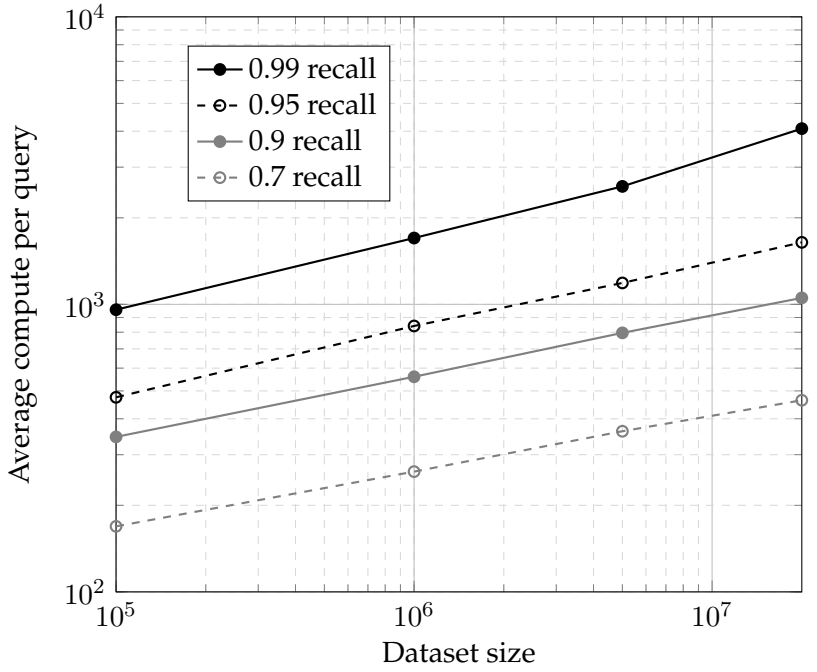


FIGURE 4.22: The growing number of distance calculations required to maintaining a fixed recall across datasets of increasing size.

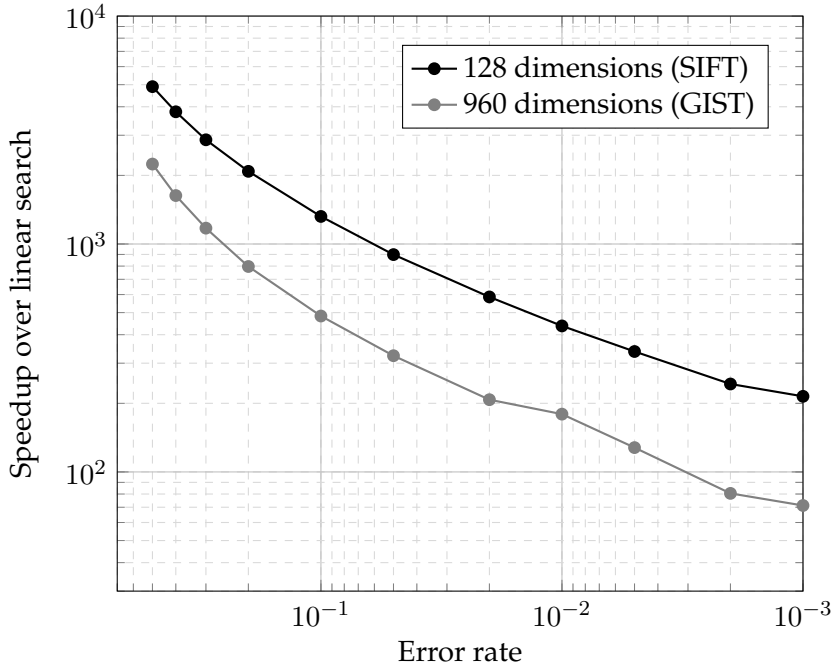


FIGURE 4.23: The consistent performance of our indexing graphs when applied to datasets with widely differing dimensionality.

Chapter 5

Indexing Binary Strings

In Chapter 4 we introduced a novel approach for indexing high dimensional visual data. We now consider the applicability of these indexing graphs for performing efficient nearest neighbour searches with binary data. Many approximate nearest neighbour (ANN) search algorithms are defined for general metric spaces, however this does not mean that they will have equivalent speed and efficiency across all metrics. Binary vectors are a good example of this, because they are often used as information dense visual features, but typically they require specialised indexing schemes. Unlike existing methods, our approach attempts to utilise the intrinsic structure of high dimensional binary data in order to build a traversable graph that allows for efficient queries in the Hamming space. We evaluate the performance of our graphs across multiple types of binary feature vectors and various dataset sizes.

When real valued data is represented on a digital computer, many individual bits of information are used to form independent integer or floating point values. Each of these values could define the measurement of a continuous property that represents a particular extrinsic dimension of a feature vector. In contrast to this, each bit of information in a binary feature vector is assumed to be independent. As such, all binary feature vectors must ultimately be constructed through a series of simple questions that each only have two possible outcomes. Figure 5.1 illustrates this concept in one, two, three and four dimensions. Each additional dimension is adding a new question that can be answered with either a 0 or a 1. Concatenating the answers to m different questions will yield a binary feature that corresponds to a particular corner of a unit hypercube in $\mathbb{R}^m$. Compared with real valued data, it is a severe

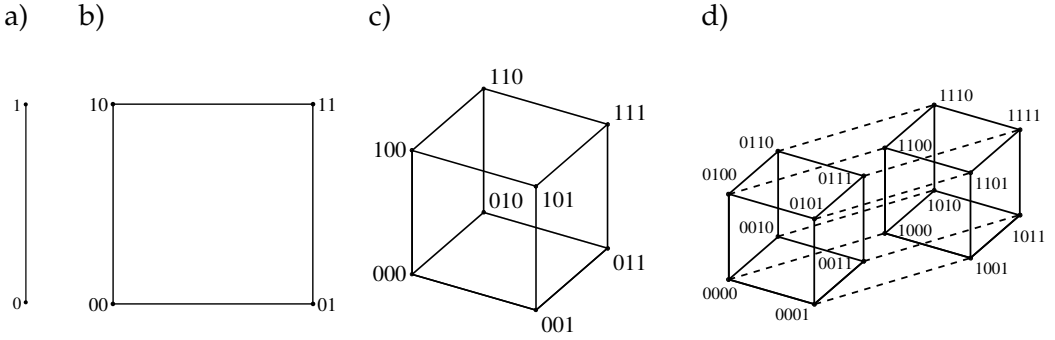


FIGURE 5.1: Compared to real valued data, binary vectors exist only at the corners of an n-cube. a, b, c, d) The full set of binary vectors in 1, 2, 3 and 4 dimensions respectively.

restriction to no longer be able to produce feature vectors throughout the interior of these hypercubes. On the other hand, binary representations do in general use fewer total bits per feature and can also enable more efficient usage of the digital hardware found in modern computers. Maximising this efficiency requires the use of a distance function that can fully utilise the fast mathematical operations that are available on modern hardware.

Previously in Section 4.1 the construction of our traversable indexing graphs made use of a distance function $dist : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}$ that was required for evaluating the occlusion rules. For binary feature vectors, we propose the use of the Hamming distance $dist_H : \mathbb{B}^m \times \mathbb{B}^m \rightarrow \mathbb{Z}^*$. This distance not only acts on the restricted representation of the binary feature vectors with $\mathbb{B}^m \subset \mathbb{R}^m$, but also maps onto the smaller domain of non-negative integers $\mathbb{Z}^* \subset \mathbb{R}$. Even so, the Hamming distance function

$$dist_H(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{m} (\mathbf{x}_{i,k} - \mathbf{x}_{j,k})^2 = popcount(\mathbf{x}_i \oplus \mathbf{x}_j). \tag{5.1}$$

is still a sufficient metric for the construction and traversal of FANNGs. For binary vectors $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{B}^m$ the Hamming distance computes the sum of absolute differences between each extrinsic dimension. This is equivalent to counting the minimum number of edges in Figure 5.1 that would need to be traversed in order to travel between the coordinates of the two vectors. In practice, it is far more efficient to compute the absolute differences using the exclusive-or operator in $\mathbb{B}^m$, and then summing the results with a population count. Due to its widespread applicability,

the population count function is available as a native operation in many modern CPU and GPU instruction sets.

While the Hamming distance is sufficient for applying the occlusion rule defined in 4.4, it is not immediately obvious how this new metric will affect the efficiency of querying a FANNG. As was discussed in Section 4.1, there are many possible ways to construct functional sets of edge lists but not all of them will be equally efficient for performing ANN searches. Most notably we can observe that when the Hamming distance function in Equation 5.1 is applied to an m dimensional dataset, it will in fact map $dist_H : \mathbb{B}^m \times \mathbb{B}^m \rightarrow \{0, \ldots, m\}$. This then means that there are only a total of $m + 1$ possible distances that can exist between any two feature vectors in that dataset. As such, it is far more likely for graph vertices or graph queries to have equidistant nearest neighbours. This will most likely result in less efficient edge lists and less effective results from the priority queue used by the greedy backtrack search in Algorithm 4.3.

5.1 Related Publications

Binary feature vectors have a number of benefits over real valued features, which describe an image region using a vector of either integer or floating point values. Trzcinski et al. [84], [85] illustrate how the construction of binary feature vectors can be more computationally efficient, use considerably less memory for storing the computed features and also compute distances using an efficient Hamming distance implementation. However, due to the limited model capacity of binary feature vectors, each dimension does need to be constructed for maximum information density. Otherwise, it will become increasingly common that matching feature vectors do not represent matching visual data.

Well constructed binary features such as ORB [8], BinBoost [13], BRISK [11] and FREAK [12] have been used in a wide range of computer vision application areas from simultaneous localisation and mapping [86] to object classification [87]. The visual features used in these systems behave as a form of data compression, by summarising the information dense regions of an image with a collection of compact

binary vectors. In many applications this process will shift the computational bottleneck from the processing of large amounts of raw image data to the matching of large sets of feature vectors.

Binary feature vectors are relatively small (typically with tens, hundreds or thousands of bits) in comparison to the image data that they are constructed from (hundreds, thousands or tens of thousands of pixels). However, the extrinsic dimensionality of these binary features is still too large for any known algorithm to guarantee exact matching from a large dataset with sub-linear time complexity. As such, it is currently infeasible to compute exact matches on large datasets when real-time performance is required. Hence, it is common practice to make use of an approximate search strategy for situations where an exhaustive linear search is deemed to be too costly. Once again, the goal of using an ANN search is to sacrifice a small degree of accuracy for a large gain in search speed.

Many of the best performing ANN algorithms have an inherent dependence on indexing vectors of real valued data. This means that these methods are unable to construct an index for a dataset of binary vectors. Efficient decision tree algorithms including the k-d trees and k-means trees proposed by Muja and Lowe [52] contain this dependency. k-d trees make use of repeatedly partition a dataset by applying a threshold to an extrinsic dimension. This become a less meaningful parameter when there are only two possible values per dimension. Similarly, k-means trees and other clustering techniques rely on computing the arithmetic mean for each extrinsic dimension. This again proves ineffective in a binary domain, although in some cases it may be possible to use an alternative function, such as the mode. Andre et al. [88] show that an inability to define effective clusters can also impact product quantisation techniques is a similar way. Here, the compact nature of binary feature vectors leaves little or no room for dimension reductions that will not have a detrimental effect on the search quality. This property extends to many other ANN techniques [89]–[91] that once again are aiming to directly reduce larger feature vectors into a compact binary representation.

Hashing techniques such as locality sensitive hashing (LSH) [29], multi-index hashing [92] and spherical hashing [93] use many specialised hashing functions that map feature

vectors to a hash space where similar hashes are likely to correspond to closely located samples. As such, these techniques are able to perform better on binary data where the feature vectors have only a fewer bits difference from their nearest neighbours. This property does not tend to be the case for most visual features that are generated from natural images. Instead many hashing algorithms are able to trade off memory footprint for recall accuracy simply by increasing the number of hash tables that are constructed and then used for queries. With additional hash tables it becomes more likely that one of the hashes will end up closer to a near neighbour. Conversely, additional candidate matches can increase the amount of linear searching that needs to be performed in the original feature space in order to select the best of the candidates.

Tree techniques, most notably hierarchical clustering [94], provide a traversable search tree that contains a constrained number of edges between nodes to reduce the number of unnecessary distance calculations that are needed during a query. Each node in the tree represents an actual sample from the given dataset which helps to avoid many of the issues that other search trees have with binary data. However the intrinsic hierarchical structure of the trees comes at an additional cost similar to the effects of a poor hash. Each time a decision is made to travel down a particular branch of the tree there is a significant probability that the nearest neighbour to the current query does not exist down that branch despite it taking the search in partially the correct direction. The backtracking cost of taking an incorrect branch decreases the further down the tree the search is, but high costs exist for early missteps. Much like use of additional hash tables, the need for backtracking can be reduced by building and then simultaneously searching multiple different trees. As the number of trees increases, so does the minimum search cost due to the simultaneous nature of the traversal.

5.2 Dimensionality Analysis

Binary feature vectors that are represented in $\mathbb{B}^m$ will only be located at the corners of a unit hypercube when viewed in $\mathbb{R}^m$. Because an m dimensional hypercube will

have exactly 2^m corners, there is an absolute limit of 2^m unique feature vectors that can be represented in such a space. As such, binary feature spaces with too few extrinsic dimensions will not be able to represent enough unique samples for certain feature matching applications. We are interested in knowing if a particular extrinsic dimensionality is sufficient for handling non-trivial feature matching problems. Additionally, we would like to investigate whether a suitably sized binary space will contain the same intrinsic structures that we have utilised for ANN searches with real valued datasets.

For this preliminary investigation, we constructed the random binary dataset described in Table 1.3. In order to understand the intrinsic structures of this data, we once again applied the Hausdorff dimensionality method described in Section 2.3.1. As with the real valued data, our first step is to compute all pairwise distances between the feature vectors in our dataset. When computing these distances for our binary data, it is important to compute the Euclidean distance in $\mathbf{R}^m$. This is equivalent to computing the square root of the Hamming distance with

$$dist_E(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{m} (\mathbf{x}_{i,k} - \mathbf{x}_{j,k})^2} = \sqrt{dist_H(\mathbf{x}_i, \mathbf{x}_j)}. \tag{5.2}$$

Using the Euclidean distance for the dimensionality of the binary data allows us to evaluate how well this data is able to fill the full Euclidean space. In particular, we can directly compare these results with our previous analysis of real valued feature vectors.

In Section 2.1, we initially looked at the distribution of feature vectors in high dimensional Euclidean spaces. As seen in Figure 5.2a, randomly sampling the interior volume of a unit 32-cube will result in an sparsely populated central region with the shaded annulus depicting the probable range of distances between the sampled locations and the centre of the cube. This largely empty central region arises because the majority of a hypercube's volume is contained within its many corner regions. Additionally, there are slightly over four billion unique corners in the 32-cube, so the probability that multiple locations are sampled from a single corner region is extremely low (assuming that the dimensionality is sufficiently large for the dataset

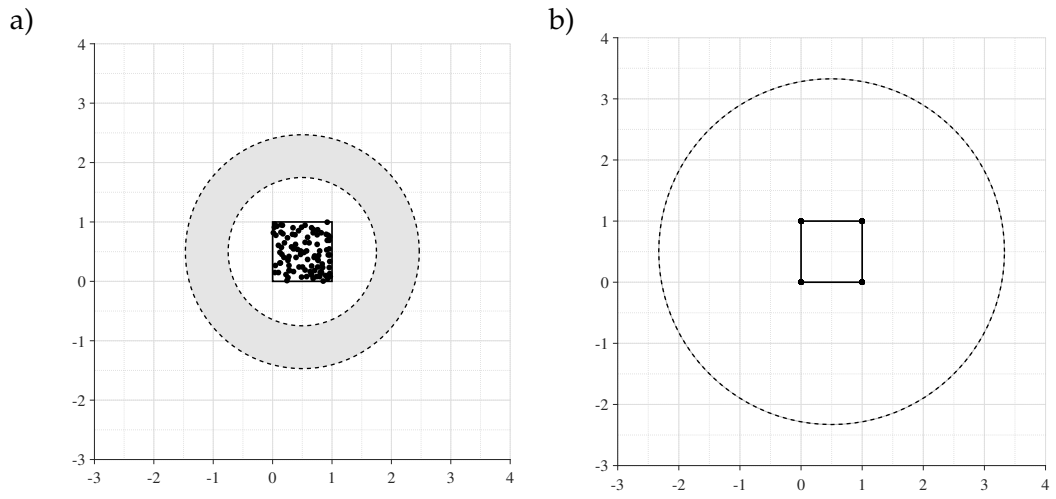


FIGURE 5.2: a) Uniform random data in a unit 32-cube will largely occupy the shaded region when considering Euclidean distance from the centre of the cube. b) Moving each sample to the nearest corner results in a binarised representation of the data.

size). Figure 5.2b provides a corresponding diagram with each vector mapped to the corner that it was closest to. This mapping moves all of the data as far as possible from the centre of the hypercube, and results in a uniform set of distances. The data is no longer bounded within two hyperspheres, but is instead located entirely on the surface of a single hypersphere that intersects each of the hypercubes corners. If we consider the information that is represented by the pairwise distances between each of the sampled vectors, it is possible that the now binarised data will have lost a critical portion of this information.

Figure 5.3 plots the probability distribution of pairwise distances for data sampled from both the interior volume and the corners of a unit hypercube. Again we can see that for this 32-dimensional data, it is extremely unlikely that any two real valued or binary vectors will be located within the same corner (with a pairwise distance less than one). Scaling the real valued data to have the same mean value as the binary data highlights the identical shape of these distributions. While the binarised data does result in a discrete number of possible distances, the information about pairwise relationships has been summarised rather than discarded. Additionally, the left tail of the binarised distribution will still allow for local neighbourhoods to be represented within the data. This is an important requirement for indexing binary

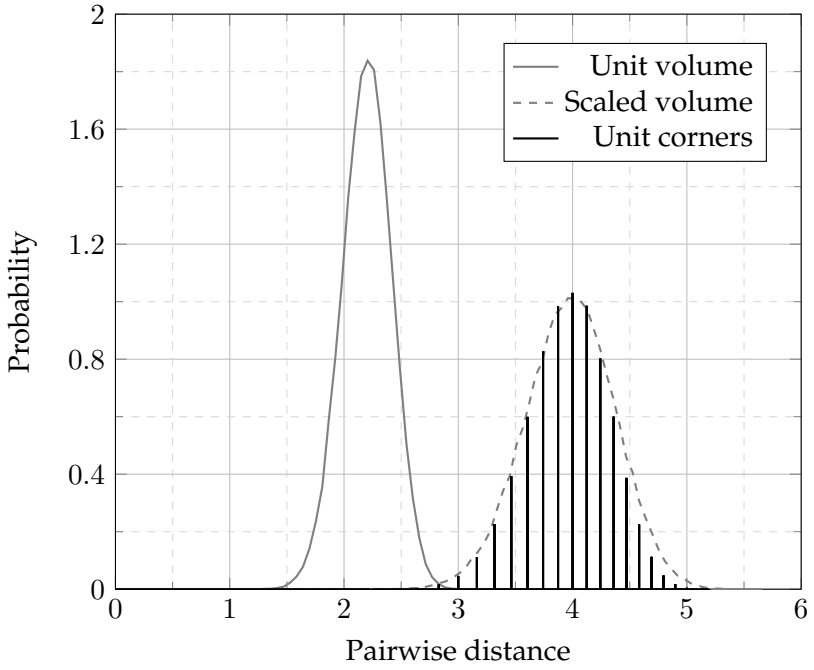


FIGURE 5.3: Probability density functions for pairwise Euclidean distance calculated between locations uniformly sampled from the interior volume and the corners of a 32-cube.

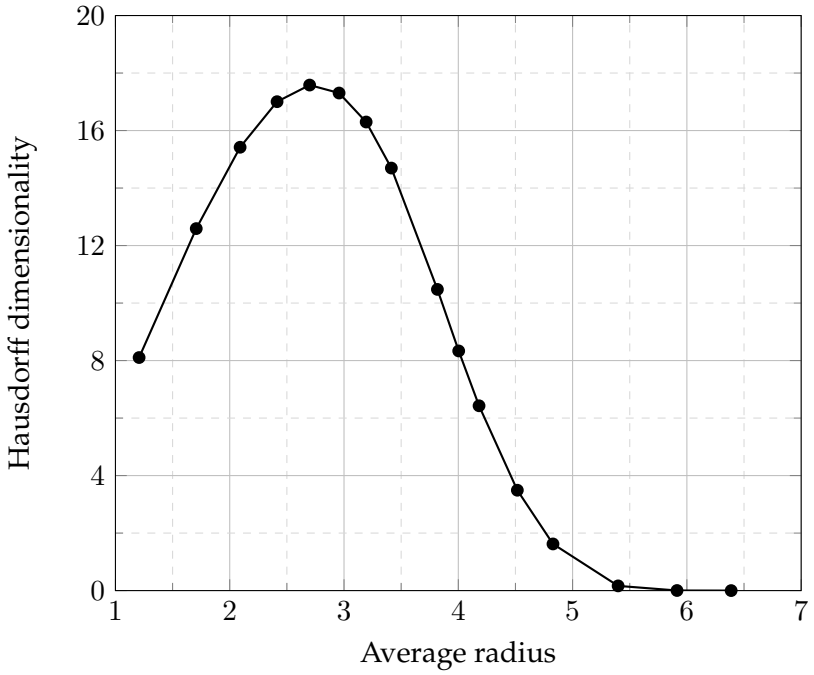


FIGURE 5.4: Hausdorff dimensionality for 32 dimensional random binary data measured at varying distance scales.

vectors with a FANNG.

Figure 5.4 plots the Hausdorff dimensionality of one million random binary vectors, each with 32 extrinsic dimensions. The plotted curve is seen to exhibit the same characteristic shape as the dimensionality plots for vectors of real valued data. In this case, we find that the peak intrinsic dimensionality is slightly above half the number of extrinsic dimensions. This peak is in fact higher than the peak dimensionality that was seen for the 128 dimensional SIFT data in Section 2.3.4. Unlike the scale change that was used to adjust the mean of the distribution in Figure 5.3, Hausdorff dimensionality is computed from a ratio of distances, so the peak dimensionality will remain invariant to scale changes. Considering that each vector in the random binary data is restricted to one of the corners on a 32-cube, it is unintuitive to expect this potential for filling continuous space. However, the results imply that we can sufficiently approximate a continuous manifold by simply sampling the corners of a high dimensional cube.

5.2.1 Impact of Dataset Size

We can utilise the symmetrical structure of binary data to gain a better understanding of how the size of a dataset influences our measure of intrinsic dimensionality. While we have seen that feature representations do not entirely fill the space they are embedded in, each dataset is only a subsampling of the full representation. If this sampling is too coarse, then we expect to measure a lower intrinsic dimensionality than the feature can offer.

Since m dimensional binary data is limited to only $m+1$ possible pairwise distances, we can count the exact fraction of pairwise distances that could exist for each possible distance value. By comparing cumulative sums of these fractions we can then compute the maximum achievable Hausdorff dimensionality for that particular extrinsic dimensionality m. Plotting the peak Hausdorff dimensionality for increasing values of m gives rise to the dashed line seen in Figure 5.5. This line represents an

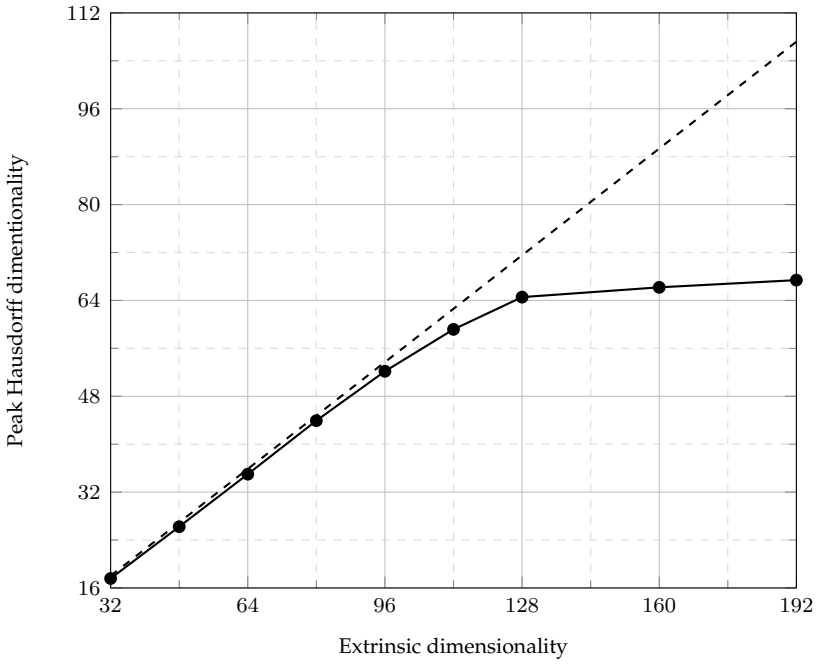


FIGURE 5.5: Peak Hausdorff dimensionality of random binary data is limited when the datasets have a fixed size at one million samples.

upper bound for the intrinsic dimensionality that can be represented in a binary feature space. In order for a dataset to approach this limit, the data would need to be an extremely fine sampling of a feature that can fill the entire extrinsic space.

The random binary vectors we have been using are an example of a dataset that does have the potential to fill the entire extrinsic space. As such, we generated sets of one million random vectors in $\mathbb{B}^m$ for increasing values of m. By using a fixed number of vectors across each of the datasets, we can observe how effectively a sampling of that size can fill different sized feature spaces. The peak Hausdorff dimensionality for each of these datasets is shown as a solid line in Figure 5.5. We can see that for dataset with around 100 or more extrinsic dimensions, the space filling capacity becomes increasingly limited by the fixed size of these datasets. Prior to this point, the dimensionality of the datasets is instead limited by the binary representation of the data.

As previously discussed in Section 2.2, high dimensional feature spaces quickly become infeasible to fill with sampled data. Once the 'blessing of dimensionality' takes over, the intrinsic structure will always be limited by the dataset size. Beyond this

point, advancements to hardware and algorithms should be focussed on efficiently handling larger datasets, rather than ones of higher dimensionality. Using a feature representation with a higher dimensionality could allow for a more natural representation of intrinsic structures. However, going too far beyond this point would be wasteful due to the computational costs of computing higher dimensional distances.

5.2.2 Impact of Vertex Degree

The edge selection criteria discussed in Section 4.1 has been shown to enable efficient ANN queries when using real valued data. In Section 4.4.1 we have also seen that the average vertex degree of a FANNG is somewhat correlated with the intrinsic dimensionality of the data being indexed. Our analysis of random binary data has shown that, perhaps unintuitive, the intrinsic structure of high dimensional binary vectors should also be suitable for our edge selection algorithms. However, since the binary data has been measured to have a higher intrinsic dimensionality than the real valued data, we expect a corresponding increase in the number of retained edges when indexing binary data.

Figure 5.6 plots the results of applying the distance occlusion rule from Equation 4.4 to each dataset of one million random binary vectors. Because these datasets have an increasing number of intrinsic and extrinsic dimensions, we can use these

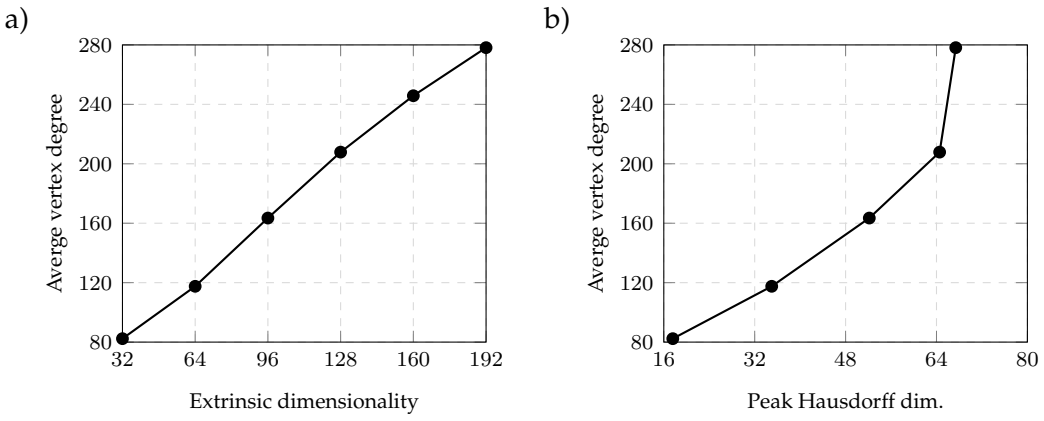


FIGURE 5.6: Relationships between the average out-degree of indexing graph vertices and the dimensionality of synthetic datasets. Each datasets contains samples generated uniformly at random on the corners of an m-cube.

constructed graphs to observe the relationship between dimensionality and average vertex degree. As was seen for real valued data, the results in Figure 5.6a show a linear growth in the number of edges as the extrinsic dimensionality increases. Figure 5.6b then combines this information with Figure 5.5 to show the relationship between average vertex degree and Hausdorff dimensionality. The linear region of this curve should provide a rough indication of the average vertex degree that is expected for a dataset with the corresponding intrinsic dimensionality.

5.3 Quantitative Results

In this chapter we have offered some analysis of the structure and characteristics of high dimensional binary data. Binary representations are often favourable due to their compact nature, although specially tailored algorithms are typically required in order to fully exploit this property. We believe that both real valued and binary data contain the same intrinsic structures that are utilised by our FANNGs. As such, our construction and search algorithms from Chapter 4 should be applicable for indexing binary data. Here we present experimental results to evaluate the performance of FANNGs on commonly used types of binary features. Details of these datasets are given in Table 1.3. Performance is evaluated across multiple feature types, as well as a wide range of dataset sizes.

We compare our method with two widely used ANN algorithms by computing the trade-off between search accuracy and the computational cost of each approach. Again we report these results in terms of error rate and speed-up over a linear search on a log-log plot, as this allows for clearer comparisons across a wide range of recall values. Additionally, we explore the scaling complexity for FANNGs constructed on binary datasets that contain an increasing number of feature vectors. Lastly, we consider the overhead costs of our method by report query times obtained with a GPU accelerated implementation of our ANN search algorithm.

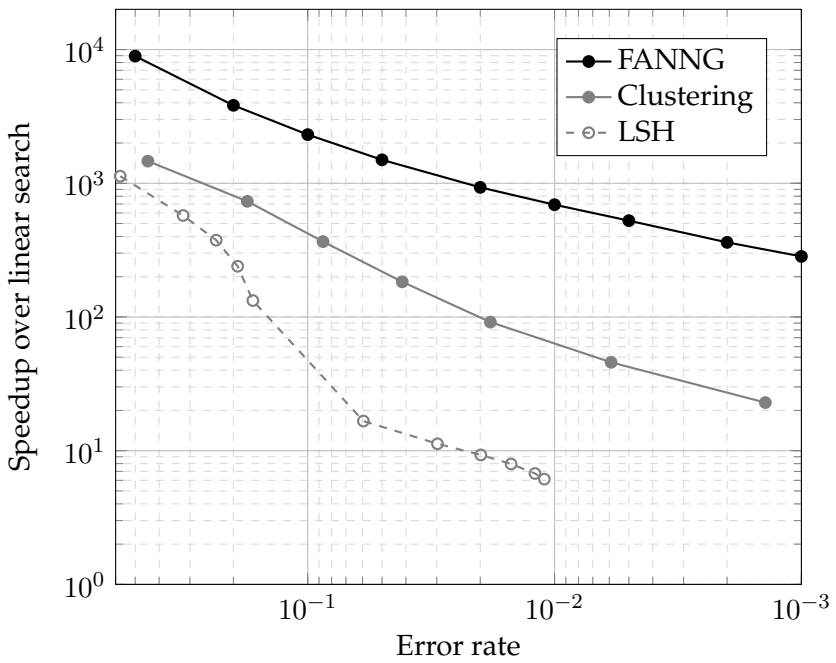
5.3.1 Comparative Performance on Various Binary Features

We compare the performance of FANNGs against each FLANN [94] method that the authors have described as being the most suitable for indexing binary data. The library contains an indexing tree method called hierarchical clustering, as well as an optimised implementation of locality-sensitive hashing (LSH). Performance results for the hierarchical clustering and LSH were obtained from the best results of a parameter sweep that was run for each dataset. All of the FANNG results were collected by tuning a single parameter to specify the desired recall rate. The edge lists of each FANNG were truncated to the initial average edge list length for that particular graph.

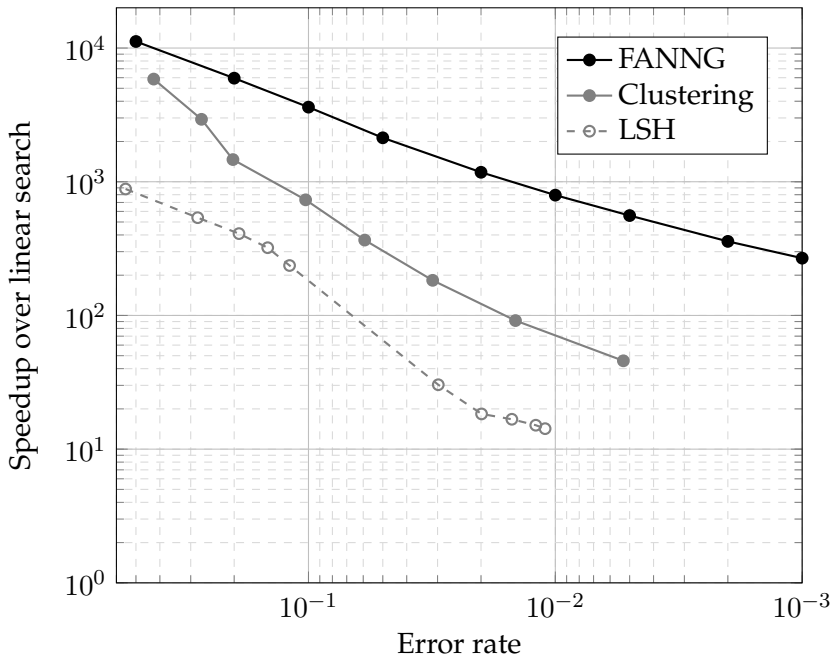
Figure 5.7 plots the nearest neighbour error rate across all algorithms and datasets. All speedup measurements are given as a ratio of the number of extrinsic distance calculations used per query between the ANN algorithm and an exhaustive linear search. The FANNG results give a clear indication of good performance at all error rates and across all four datasets. This behaviour supports our hypothesis that high dimensional binary and real valued data do contain similar structures that can be exploited by our edge selection criteria. The performance gains of FANNG over the hierarchical clustering and LSH algorithms comes at the cost of a longer offline building phase and a greater memory footprint. This places FANNG as the most favourable binary indexing scheme for applications without significant memory constraints and with either sufficient time for pre-building the indexing graph or a sufficiently large number of queries so that the build time becomes negligible.

Comparing the performance of each search algorithm across the different feature types, we see very little impact from choosing one representation over another. The most notable difference is for the 512 bit datasets, where LSH performs poorly at higher error rates, while both the FANNG and hierarchical clustering methods are slightly more efficient at lower error rates. These results indicate that on average there is a larger number of differing bits between a query vector and the ground truth nearest neighbour for a BRISK or FREAK feature vector, when compared to the other feature types. Since there is little difference between the ANN performance of each

a)


FANNG
Clustering
LSH
Speedup over linear search
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
Error rate

b)


FANNG
Clustering
LSH
Speedup over linear search
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
Error rate

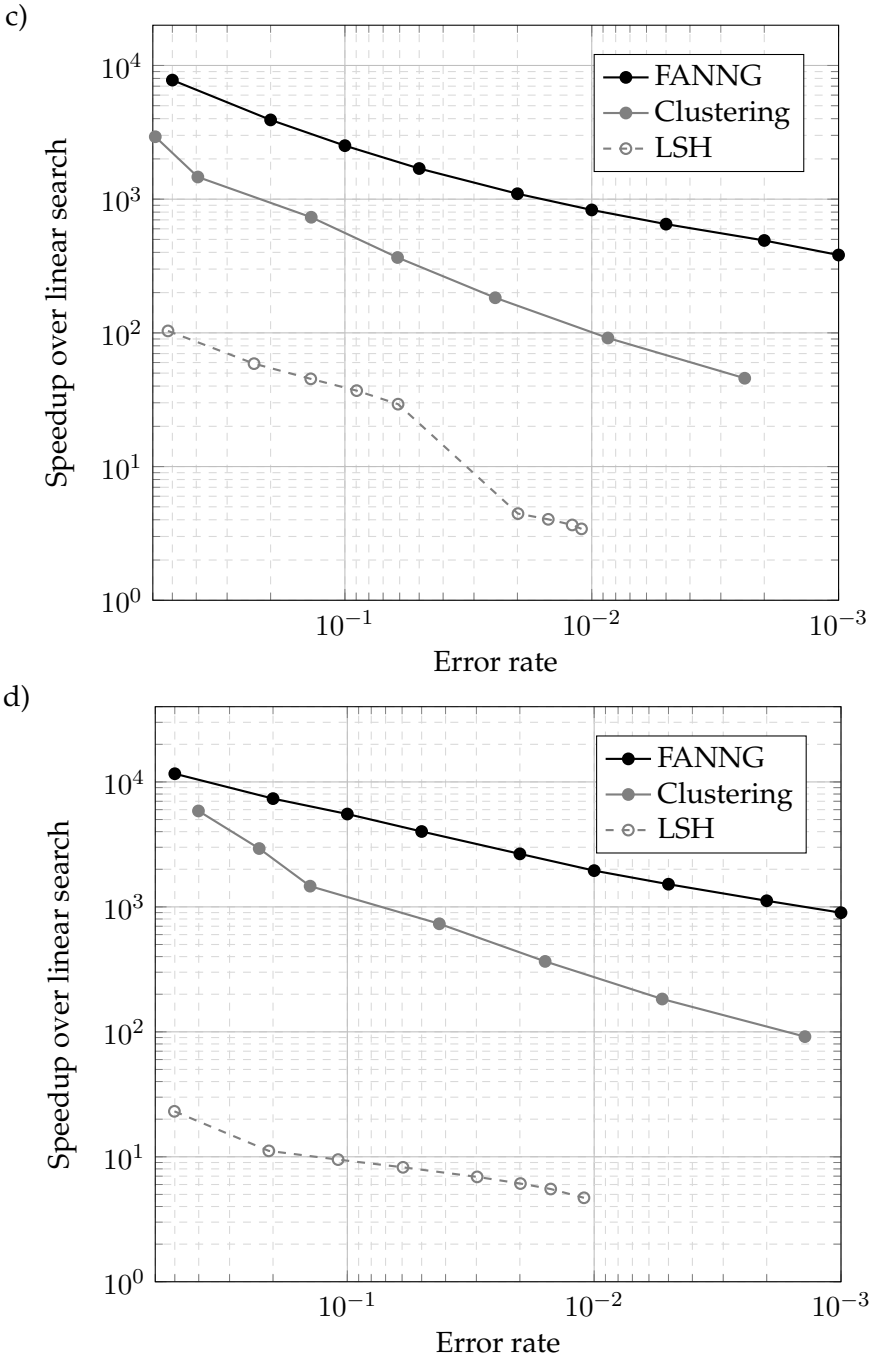


FIGURE 5.7: Comparative performance of ANN search methods applied to binary datasets. a,b,c,d) 1.5M ORB, BinBoost, BRISK and FREAK feature vectors respectively.

binary feature, selecting a feature type should instead be based on the capacity for each feature representation to preserve local distance relationships when mapping feature vectors from an image patch to the Hamming space. For our experiments, we have assumed these relationships have been successfully represented within our datasets. This concept of accurate feature representation will be explored in-depth throughout Chapter 6.

5.3.2 Comparative Performance on a GPU

Table 5.1 presents wall clock timings from our accelerated GPU implementations of our ANN search and an exhaustive linear search. These timings include all overhead search costs, such as maintaining a truncated priority queue of the vertices currently being explored. As with the truncation of the FANNG edge lists, aggressively truncating the priority queue can limit the possibility of reaching higher recall values. For a dataset of one million BinBoost feature vectors, we found that a maximum queue length of 128 was sufficient for achieving recall rates of up to 99.9%.

Search times were computed with batches of ten thousand query vectors using a GTX Titan X GPU. While operating with a 90% recall rate, our method was able to perform each query at about 150 times the speed of the linear search. This amounts to a quarter of the performance difference that was observed for the real valued data in Section 4.5.3. However, the Hamming distances computed for the binary data are considerably faster than the squared Euclidean distance calculations for the real valued vectors, so any overhead costs will be more pronounced. While the binary vectors can be successfully indexed using our approach designed for real

TABLE 5.1: Comparative performance on a GPU.

Method	Recall@1	Average query time (μs)
Linear search	1.000	179.6
FANNG	0.999	10.9
	0.990	3.9
	0.950	1.6
	0.900	1.1

valued data, larger average vertex degrees when indexing the binary data results in significant additional search costs.

5.3.3 Performance at Scale

Figure 5.8 plots the number of distance calculations needed to maintain a fixed rate of recall as the size of a dataset increases. These results were collected by constructing FANNGs on increasingly larger portions of a dataset with 50M BinBoost feature vectors. For each dataset size, ground truth nearest neighbours were identified using an exhaustive linear search for each of the query vectors.

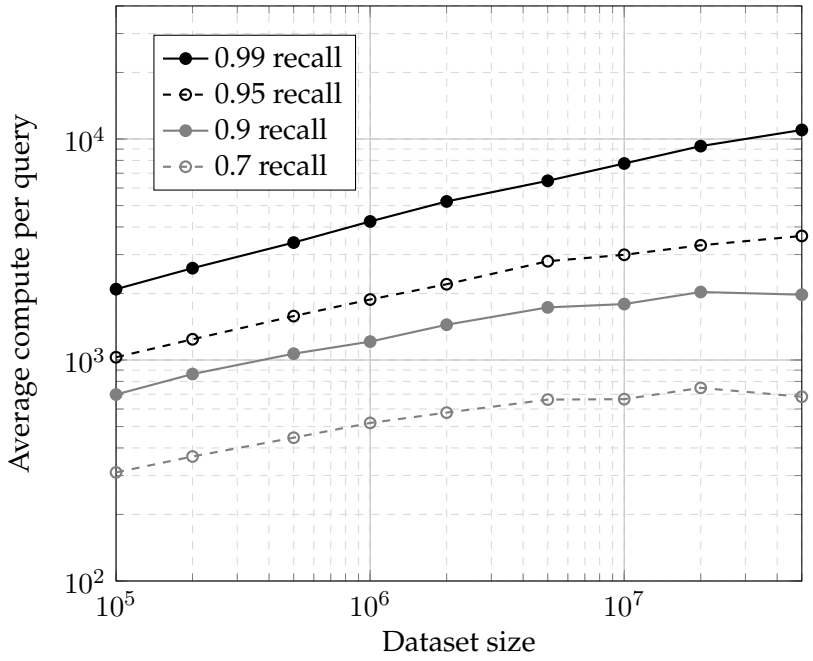


FIGURE 5.8: The growing number of distance calculations required to maintaining a fixed recall across binary datasets of increasing size.

From the plotted curves, we see that the 99% recall plot and significant portions of the other plots are all approximately linear on a log-log plot. As such, the results for binary feature vectors are consistent with the results seen in Section 4.5.4 for real valued feature vectors, having a power law relationship between the search performance and the dataset size. Since distance calculations represent the majority of both the runtime and the computational cost, we can say that for a dataset of n feature vectors, the cost complexity and the time complexity are roughly between $O(n^{0.15})$ and $O(n^{0.25})$.

Chapter 6

Learning from Visual Data

Concurrent advancements in computer hardware and large scale data collection are the current driving force behind substantial advancements in machine learning techniques. Artificial neural networks were originally formulated in the 1980s, however their applicability to challenging real world computer vision tasks only arose in the 2000s. LeCun et al [95] first demonstrated the potential for convolutional neural networks (CNNs) to compete with human experts in vision related tasks. Figure 6.1 illustrates the fundamental structure of LeNet and its predecessors. Information is extracted locally from image data and then combined at multiple scales to allow for output layers that can consider both local and global features. Krizhevsky et al [96] adapted this architecture for large scale image learning with AlexNet.

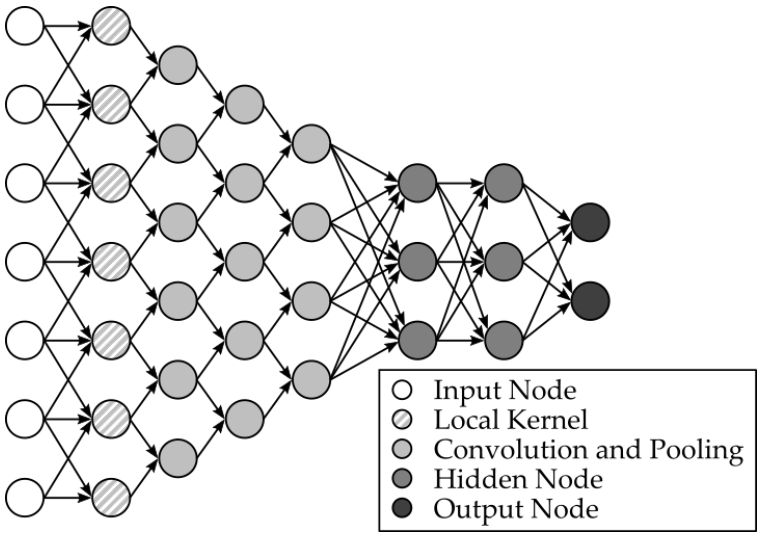


FIGURE 6.1: CNNs extract visual information using layers of functions that combine, threshold and downsample the data.

Training a network of this size was achievable in part due to the incorporation of rectified linear units (ReLU)[97] as a computationally efficient non-linear function. Simonyan and Zisserman [98] achieved a further significant reduction in error with their VGGNet architecture by utilising softmax classification and smaller filters that allow for additional network layers. Similar benefits were seen by Szegedy el al [99] with GoogLeNet and He el al [100] with ResNet. Both architectures allow for small sub-networks and increase the depth of the network. Deeper networks typically require more training data and longer training times, however they are capable of reaching higher levels of performance on a range of vision tasks.

These networks represent a powerful data driven approach to learning a complex and configurable non-linear function f. In the context of computer vision, these functions have the capacity to replace entire pipelines such as the mapping of visual data from an image space to a feature space. Learning these complex functions currently requires an enormous amount of training data and occurs during an off-line training phase. Training involves optimisation with a known objective function using stochastic gradient descent. Here, batches of known images $\mathcal{X}$ are presented

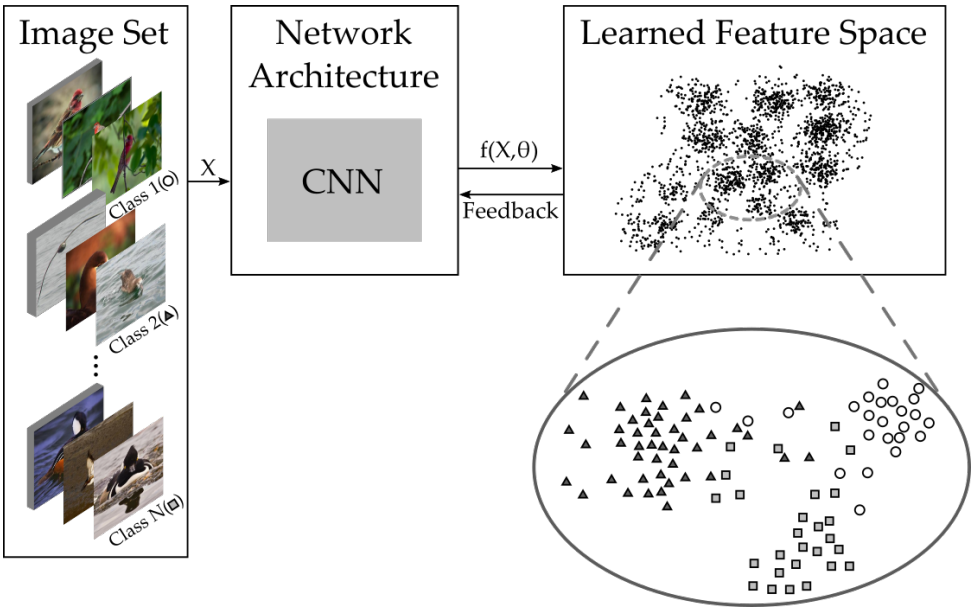


FIGURE 6.2: Large datasets of images can configure a CNN to produce a well structured feature space.

to the network and the current network weights θ determine a mapping to the feature space $f(\mathcal{X}, \theta)$. The objective function can then be used to determine the quality of the resulting feature vectors and will provide feedback for adjusting the network weights. Once trained, the CNN model can be applied to previously unseen data using the finalised network weights. Figure 6.2 illustrates the training pipeline with an objective function that is aiming to produce clusters of different image classes in the feature space.

In this chapter we explore two different methodologies for learning robust feature spaces that utilise the information offered by nearest neighbour correspondences. Our first method is built upon the triplet network architecture, where an anchor sample from a particular image class acts as a reference point for two other samples in the feature space. While many triplets of images can be formed, only a small fraction of them are useful for refining the structure of a feature space by adjusting θ in a way that will improve the mapping of correspondences into the feature space. By constructing a FANNG over the embedded data, we can consider all possible triplets and select those that will progress the learning process. Following this, our second learning method replaces the concept of a triplet, by evaluating the entire neighbourhood around the anchor samples. Here, our optimisation function encourages the formation of class specific Gaussian clusters around each anchor. Once again we construct a FANNG over the embedded data, this time as a means of approximating the computationally expensive Gaussian distances. Lastly, both of our methodologies are evaluated on transfer learning tasks. This determines how effectively each of the learned feature spaces are able to represent correspondences between previously unseen images.

6.1 Triplet Networks

In order to produce robust feature embeddings, current methodologies will commonly use a triplet model to minimise the relative distance between samples from the same class and to maximise the relative distance between samples from different classes. As illustrated in Figure 6.3, class clusters will naturally form in the

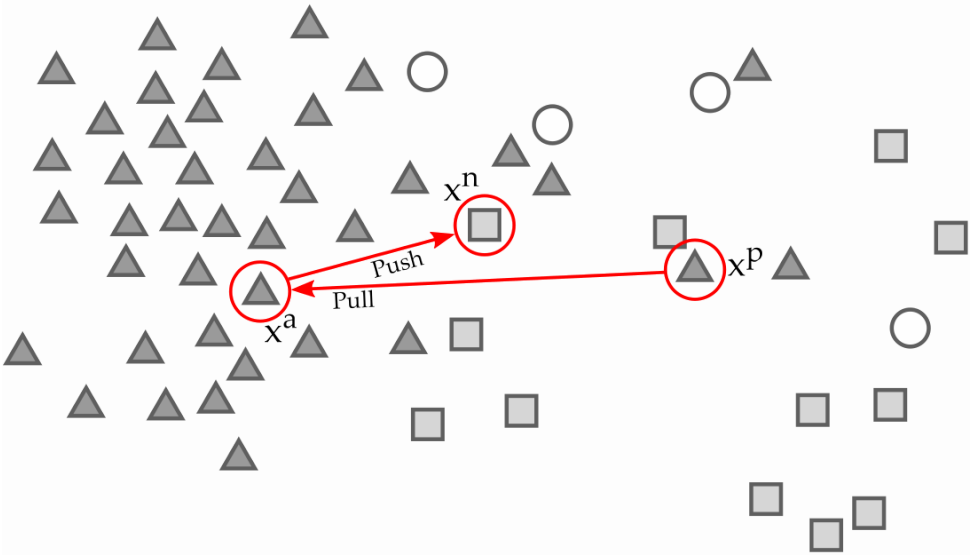


FIGURE 6.3: The relative position of three embedded samples informs the formation of a more robust feature space.

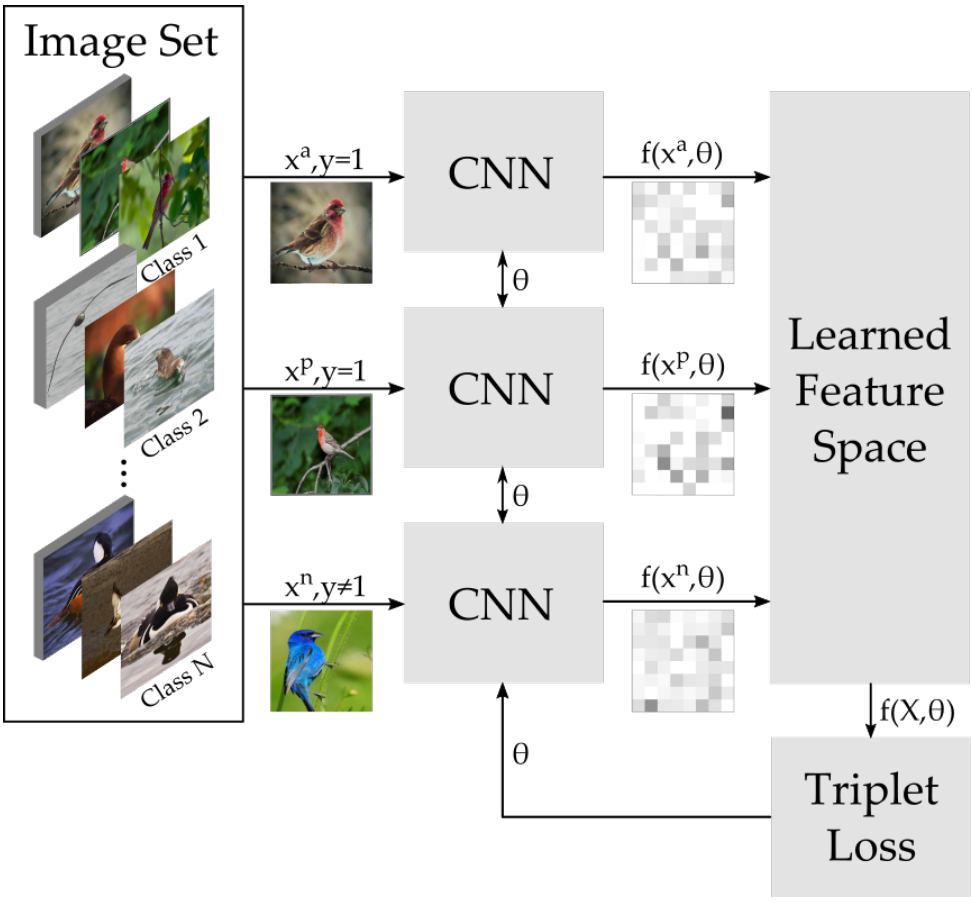


FIGURE 6.4: Conceptually the triplet model uses three CNNs to map a triplet of images into the feature space. In practice, their shared network parameters allows for a single CNN to be used.

embedding space when samples of the same class are pulled together and samples of different classes are pushed away.

As shown in Figure 6.4, triplet networks consist of three CNNs that are trained together using triplets of samples. Each triplet is constructed to contain an anchor sample, a positive sample of the same class as the anchor, and a negative sample of a different class. These networks utilise a loss function that penalises large relative distances between the anchor and positive samples and small relative distances between the anchor and negative samples. Since all samples are selected from the same dataset and will be mapped to the same feature space, the three networks are typically constrained to share a single set of parameters.

In order to describe the triplet architecture, we first define $\mathcal{T}$ a training set of n images in C classes where $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}\ \forall i \in \{1, ..., n\}$. Here the image $\mathbf{x}_i \in \mathbb{R}^{u \times v}$ is pre-labelled as a member of class $y_i \in \{1, ..., C\}$. We then denote the m dimensional feature embedding of sample $\mathbf{x}_i$ by $f(\mathbf{x}_i, \theta)$, where $f : \mathbb{R}^{u \times v} \times \mathbb{R}^k \rightarrow \mathbb{R}^m$ and $\theta \in \mathbb{R}^k$ represents the k parameters shared by the three CNNs. These parameters include values such as weight matrices, bias vectors and normalisation parameters. Each CNN contains the same l layers and is defined by

$$f(\mathbf{x}, \theta) = t_{\text{out}} \circ r_l \circ h_l \circ t_l \circ r_{l-1} \circ h_{l-1} \circ t_{l-1} \circ ... \circ r_1 \circ h_1 \circ t_1(\mathbf{x}). \tag{6.1}$$

On the i^{th} network layer $t_i(.)$ denotes a linear transformation, $h_i(.)$ represents a normalisation function and $r_i(.)$ is a non-linear activation function such as a ReLU. Furthermore, the linear transform $t_i = [t_{i,1}, ..., t_{i,j_i}]$ is an array of j_i pre-activation functions.

For the triplet loss function, we first define a triplet as an anchor sample $\mathbf{x}^a = \mathbf{x}_i$ (from class y_i), a second sample from the same class $\mathbf{x}^p = \mathbf{x}_j$ (with $i \neq j$ and $y_i = y_j$), and a sample from a different class $\mathbf{x}^n = \mathbf{x}_k$ (with $y_i \neq y_k$). The loss function for each triplet is then defined by

$$L(\mathbf{x}^a, \mathbf{x}^p, \mathbf{x}^n, \theta) = \max\left(0, 1 - \frac{dist\left(f(\mathbf{x}^a, \theta), f(\mathbf{x}^n, \theta)\right)}{dist\left(f(\mathbf{x}^a, \theta), f(\mathbf{x}^p, \theta)\right) + \eta}\right). \tag{6.2}$$

Here the distance function is Euclidean distance, and η is the desired distance margin

between class embeddings. The loss function itself is a hinge loss applied to the relative distances between the anchor and the other two samples.

6.1.1 The Vanishing Gradient Problem

The vanishing gradient problem is a well known issue that impedes the training of triplet networks. In order for learning to continue it is important that the training regime continually provides the network with triplets that will teach new information or reinforce prior information. Learning progresses due to computed loss values that are propagated back through the network and cause changes to the network parameters. Importantly, the loss function in Equation 6.2 will always have zero magnitude when the negative sample is at least the margin distance further from the anchor than the positive sample. We refer to these triplets as being 'well ordered' due to the desirable structure of having the positive sample closer to the anchor and negative sample further from the anchor. The vanishing gradient problem occurs when a network is provided with too many well ordered triplets, and so the training halts prematurely due to the lack of a strong training signal.

Producing well ordered triplets is in fact the primary goal of the training. So it is natural to find that as the training progresses and class clusters begin to form, the vast majority of triplets will quickly become well ordered. This idea is illustrated in Figure 6.5a, where the large regions of easy positive samples and easy negative samples are shown relative to a particular anchor. In general, the region of easy positive samples can be represented as the interior of a hypersphere centred on an anchor and with a radius that just excludes the closest negative sample. Similarly, the easy negative samples can be bounded on the outside of a hypersphere centred on the anchor and with a radius that just contains the furthest positive sample.

Figure 6.5b provides an example of a triplet with no easy samples. In this case the positive sample is further away from the anchor than the negative sample. As such, this triplet will result in a non-zero loss that can be used for training the network. If we consider the cases where the two hyperspherical boundaries that separate the

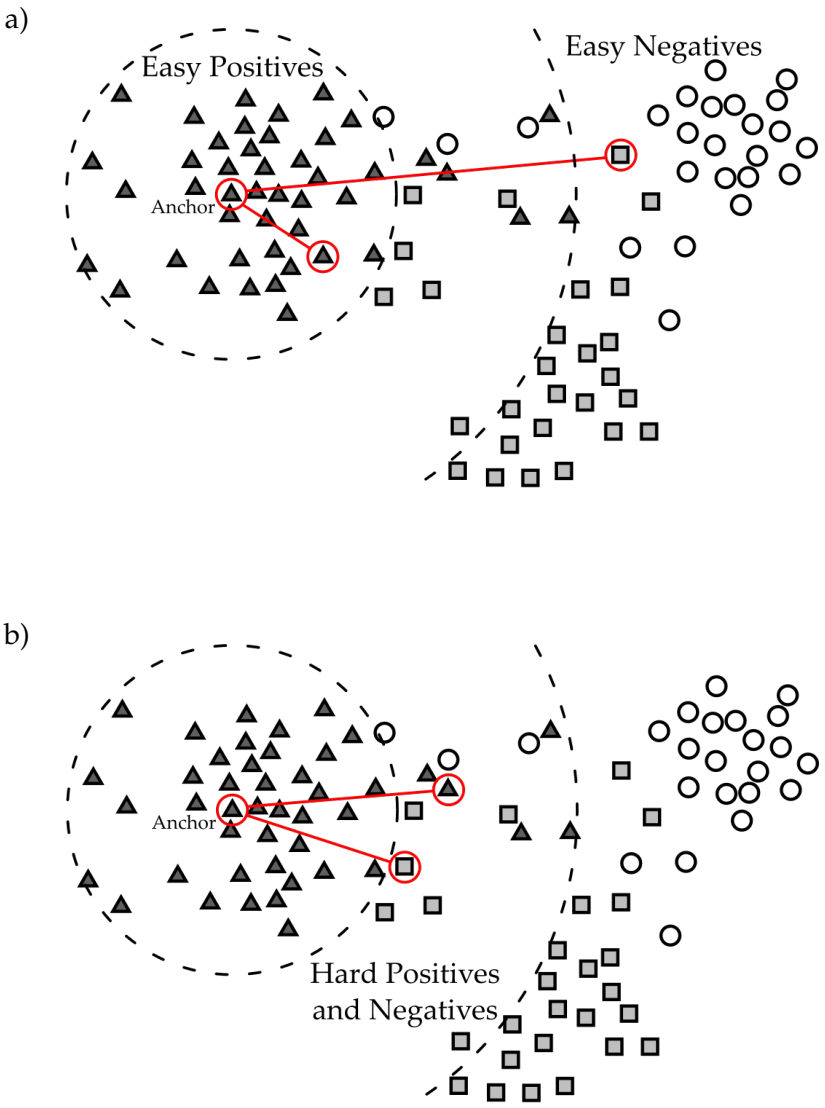


FIGURE 6.5: Useful triplets must contain samples from poorly defined regions of the embedding space. a) Well ordered triplets contain at least one easy sample and will not progress the training. b) Triplets that contain both hard positive and negative samples can help to shape the embedding space.

easy and hard samples are more than the margin distance apart, then for any chosen anchor, a triplet will always be well ordered as long as at least one of the other two samples is an easy sample. When we consider just the hard samples, it is still possible to construct triplets that will not advance the training. However, we can guarantee that all of the desirable triplets that produce non-zero losses will also exclusively contain hard samples.

To mitigate the vanishing gradient problem and ensure that more triplets contain hard samples, it has become common practice to mine the feature space for desirable samples. Figure 6.6 illustrates for a particular anchor, how the current feature space embeddings can be used in the selection of the other two samples. However, there are two major challenges when mining for hard samples. Firstly, every step of the stochastic gradient descent will likely result in at least some changes to the feature space. These frequent changes will modify the embedding location for many of the previously mined samples and could also invalidate the results of that previous

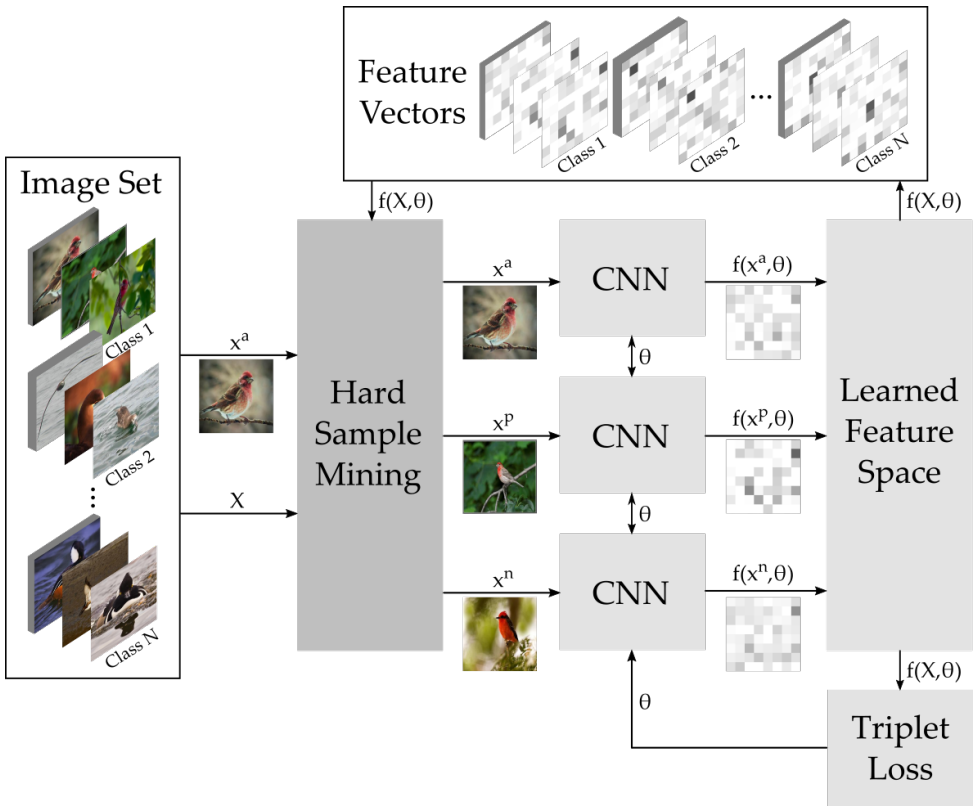


FIGURE 6.6: Hard sample mining is able to utilise information from current feature correspondences in order to construct triplets that will continue to refine the feature space.

mining. The second major challenge is that for a training set with n samples, the upper bound on the number of possible triplets is n^3. While deep CNN training is typically improved with larger training sets, naively evaluating each possible triplet is currently infeasible even for datasets of moderate size. Despite these challenges, current triplet mining heuristics can still achieve considerably better results than using randomly selected triplets.

Triplet networks have been found to perform favourably in domains where a relatively small amount of training data is available across a relatively large number of classes. This training scenario creates a large imbalance in the difficulty of mining hard negative samples compared to mining hard positive samples. With few training samples per class it is easy to evaluate all positive samples for a particular anchor, but the large number of other classes will all contribute to a multitude of negative samples. With just a small amount of training, it is reasonable to expect that vast majority of negative samples will already be easy. As such, it is advantageous for mining to only focus on identifying hard negative samples.

6.1.2 Related Publications

The development of deep metric learning models for the formation of robust feature spaces is a core component of many computer vision systems. The main advantage of these models is their ability to leverage information from large datasets of labelled images in order to learn feature spaces, where samples from similar classes tend to be close together, while samples from different classes are more likely to be far away from each other. This approach has proven to be effective in scenarios where there is an extremely large number of classes and a low number of samples per class. Under these conditions, the implementation of traditional classifiers becomes far more challenging.

A popular approach to learning feature embeddings is to utilise a deep learning model that is based on a triplet network [101], [102], which is an extension of the earlier Siamese network [103]. However, the impact of the vanishing gradient problem has necessitated the development of importance sampling techniques that stochastically

under sample the set of all possible triplets. The success of these techniques relies on generating enough samples so that a certain fraction of the hard positive and negative samples are statistically likely to be used for training. Because of the high computational complexity of finding hard positive and negative samples, an alternative approach is to instead incorporate additional loss functions that take into account the global structure of the embedding space.

Shrivastava et al. [104] formulate hard sample mining as a relabelling of the bootstrapping problem [105], where the idea is to initially train the feature model with triplets from input data that is well separated, and then gradually introduce more challenging positive and negative samples as training progresses. One of the major challenges of this approach is the myriad of possible ways in which this training regime could be formulated. Such a regime is required to efficiently sample the training set in a way that allows for the selection of effective training samples. However, the definition of what constitutes an effective or challenging sample is ill-defined. To this end, Wang et al. [102] described a method for building triplets based on a manual annotation of sample relevance. Importance sampling can then be used to construct the final triplets. This method is ultimately limited by its reliance on the manual annotations.

A more recent method from Simo-Serra et al. [106] utilises image class labels in order to determine and then gradually introduce the hardest sample pairs to a siamese network. This is achieved by randomly sampling the training set for pairs of samples, and then sorting them based on their feature space distance. Anchor-positive pairs are sorted in descending order, while anchor-negative pairs are sorted in ascending order. As such, training pairs can then be formed from the head of each list. Han et al. [107] introduce a similar reservoir sampling method for selecting positive and negative samples, however this methods does not use any form of importance sampling.

The FaceNet architecture from Schroff et al. [108] introduces a triplet training approach, where batches of training samples are initially selected at random. Pairs of anchor and positive samples are then chosen from within each batch, and the hardest negative samples are used to complete each triplet. These negative samples are

referred to as semi-hard samples, since they are only guaranteed to be the hardest negative samples within their random batches. Additionally, the semi-hard negative sampling is able to improve the robustness of training by avoiding the possibility of over-fitting to outliers in the training set. These outliers can derail mining methods that perform global searches for the hardest samples. Song et al. [109] achieve a similar result by efficiently computing the full pairwise distance matrix over a random subset of the training data. This full matrix then allows for a specialised loss function that utilises all positive and negative sample distances to form a lifted structured embedding for each training batch.

In order to avoid the computational cost of sample mining, Kumar et al. [110] have proposed a global loss function that uses first and second order statistics to encourage the separation of class clusters in the feature space. However, this method does still rely on stochastic sampling of positive and negative samples. Similarly, Ustinova and Lempitsky [111] propose a loss function that minimises the integral of the product between a distribution of negative similarities and a cumulative density function of positive similarities. Finally, Song et al. [112] introduce a loss function that optimises using normalised mutual information (NMI) as a global cluster quality metric.

6.2 Smart Mining for Triplet Embeddings

To increase the efficiency and effectiveness of finding hard triplets, we first observe that hard sample mining can be framed as an instance of the ANN search problem. When mining for hard samples we are primarily interested in avoiding the computational cost of exhaustively searching through the entire training set. Using FANNGs we can efficiently consider samples across the entire training while only needing to evaluate a small fraction of these training samples. In Chapter 4 we saw that FANNGs are able to trade-off a small decrease in nearest neighbour recall for a large gain in computational efficiency. They are particularly efficient when applied to high dimensional feature vectors and when a very high recall rate is desired. Additionally, FANNGs are built in the full embedding space which allows for a triplet

selection process to reuse the exact distances that have been computed during the ANN search.

As discussed in Section 6.1.2, hard mining has shown to be an effective method for training triplet networks when they would otherwise be impeded by the vanishing gradient problem. Naively, this can be achieved by selecting triplets that provide the greatest violation of the triplet constraint. For instance, given an anchor $\mathbf{x}^a$ with class label y^a, the hardest positive is defined as

$$\mathbf{x}^p = \underset{\substack{(\mathbf{x}_i, y_i) \in \mathcal{T} \\ \mathbf{x}_i \neq \mathbf{x}^a,\ y_i = y^a}}{\arg\max} \ dist\big(f(\mathbf{x}^a, \theta), f(\mathbf{x}_i, \theta)\big), \tag{6.3}$$

and the hardest negative as

$$\mathbf{x}^n = \underset{\substack{(\mathbf{x}_i, y_i) \in \mathcal{T} \\ \mathbf{x}_i \neq \mathbf{x}^a,\ y_i \neq y^a}}{\arg\min} \ dist\big(f(\mathbf{x}^a, \theta), f(\mathbf{x}_i, \theta)\big). \tag{6.4}$$

Semi-hard mining can avoid the costly $\arg\max$ over the entire training set by instead considering randomly selected subsets of the training data. The major disadvantage of using random subsets is that the majority of samples are not being considered in each subset. This means that the hard samples needed to produce non-zero gradients can still be completely missed.

Instead of building triplets from random subsets of samples it is easier to find hard triplets using nearest neighbour sets around each anchor. A small set of nearest neighbours in the current feature embedding is guaranteed to contain samples that will produce desirable triplets. If we consider an anchor $\mathbf{x}^a = \mathbf{x}_i$ from a class with an average number of samples C_{avg}, then we only need to consider the set $\mathcal{S}_i \subset \mathcal{T}$ that contains the C_{avg} nearest neighbours of $\mathbf{x}^a$. The local embedding is very well formed if the entire neighbour set is found to be positive samples (with the closest neighbour being the anchor itself). In this case there are no hard samples, so all triplets will be well ordered when using that particular anchor. In non-degenerate cases there will be at least one negative sample in the nearest neighbour set. Each of these negative samples is guaranteed to be a hard negative and will also indicate the existence of a hard positive sample that is not in the nearest neighbour set. Any combination of

these hard positives and negatives will result in a triplet that produces a non-zero gradient.

One key additional advantage of mining with randomly chosen subsets is the inherent low probability for repeated attempts at learning from outliers. These outliers are often erroneously labelled samples that will form hard triplets but will not improve from repeated attempts at learning. Using nearest neighbour sets will easily find the hardest possible triplets in the dataset, so an additional heuristic is needed to avoid potential outliers. For an anchor $\mathbf{x}^a = \mathbf{x}_i$, we define a smart negative as any negative sample in the nearest neighbour set $\mathbf{x}^n \in \mathcal{S}_i$ such that

$$dist\big(f(\mathbf{x}^a, \theta), f(\mathbf{x}^n, \theta)\big) > \kappa \cdot dist\big(f(\mathbf{x}^a, \theta), f(\mathbf{x}^p_{\text{NN}}, \theta)\big). \tag{6.5}$$

Here κ is a real valued global tuning variable with $\kappa > 1$ and $\mathbf{x}^p_{\text{NN}}$ is defined as the nearest positive sample to $\mathbf{x}^a$. Typically we require that $\mathbf{x}^p_{\text{NN}}$ should be a member of $\mathcal{S}_i$. Together κ and $\mathbf{x}^p_{\text{NN}}$ parametrise an exclusion boundary that contains samples that are considered to be too hard for current triplets.

The relationship between the neighbourhood of samples around an anchor and the exclusion boundary is explored in Figure 6.7. The distance d_{NN} is used to denote the distance between the embedding of an anchor and the nearest positive sample, so the distance κd_{NN} is equivalent to the right-hand side of Equation 6.5. Illustrated is the projection of the hyperspheres centred on an anchor and with a radius of κd_{NN}. The anchor in Figure 6.7a is in a poorly structured margin between class clusters. As such, the negative samples within the large exclusion boundary are considered too hard for current triplet mining. A larger concentric hypersphere is shown containing a few positive samples that are closer to the anchor than the nearest smart negative. When a class cluster is well formed around an anchor, as in Figure 6.7b, most positive samples will be labelled as being too easy because they would always form well ordered triplets. Well formed clusters are more likely to contain anchors with positive samples very nearby, so tighter exclusion boundaries allows for harder negatives to continue progressing the training. As such, training with harder negatives will happen naturally as classes continue to form their own clusters.

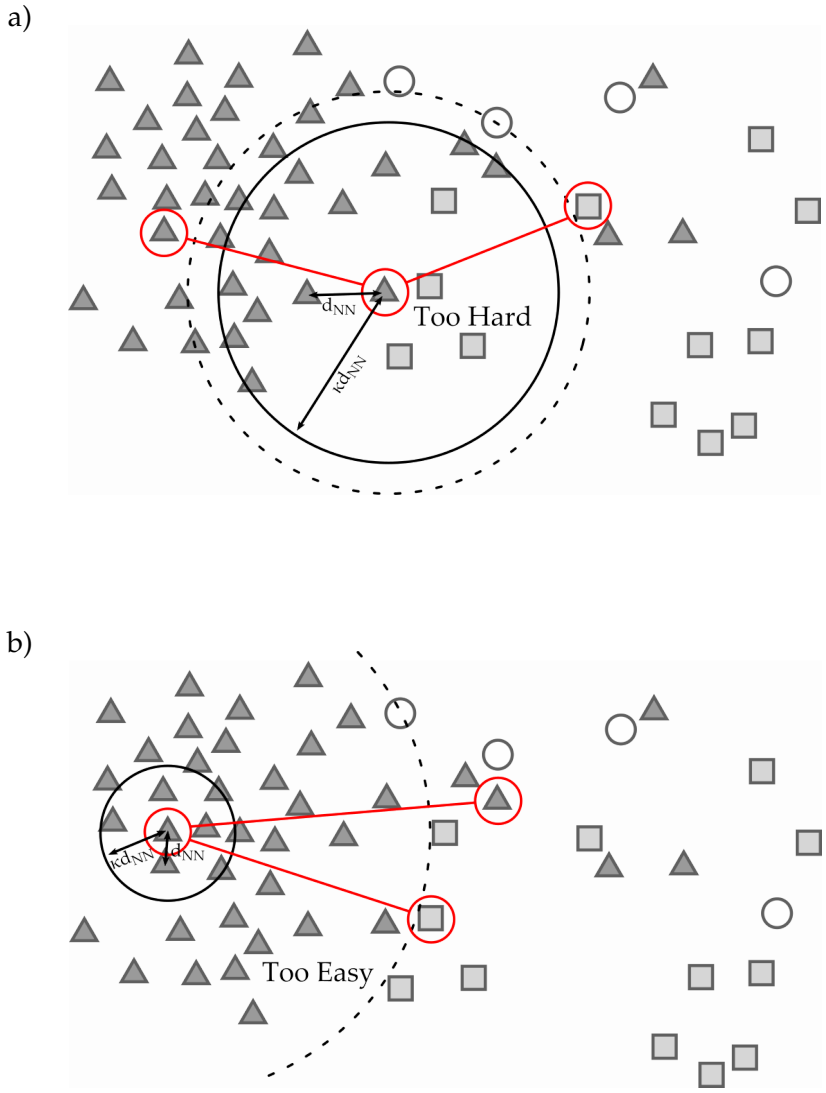


FIGURE 6.7: A simplified projection of the neighbours and mining boundaries around triplet anchors. a) An anchor in a poorly structured margin between class clusters has an exclusion boundary containing negatives samples that are deemed too hard for training. b) A tighter class clustering yields a small exclusion boundary for hard negatives. Easy positive samples are avoided outside the exclusion boundary so that all mined triplets will violate the triplet constraint.

The global variable κ provides a direct means of controlling the rate at which the exclusion boundaries are shrinking. While the distance between the embedding of an anchor and the nearest positive sample provides local information for relative sizing of exclusion boundaries, κ is used to determine the global scale. While training is progressing and adjustments are being made to the embedding structure, the formation of clusters will also be adjusting the size of exclusion boundaries. This complex interaction will very likely change the number of smart negatives that are available at any one time. The nature of the vanishing gradient problem suggests that the overall trend will be a decrease in the number of mined negative samples. In order to counteract this, we would like to decrease κ at a rate that keeps the global number of smart negatives roughly constant. This allows for a bootstrapping behaviour, where previously excluded negative samples can be considered later during training either because nearby positive samples have formed a tighter cluster or because the global constant has been sufficiently reduced.

6.2.1 Implementing Smart Mining with FANNGs

Current state-of-the-art training regimes for deep CNNs utilise stochastic gradient descent to direct convergence towards a more optimum network configuration. Stochastic gradient descent relies on partitioning each training epoch into smaller random batches of training samples. Sequential processing of each batch involves a forward pass to determine the current embedding of samples within the batch, followed by a backwards propagation of the gradients determined by the loss function. As such, the global structure of the embedding space is expected to change with each consecutive batch. Triplet mining methods are in fact aiming to mine the current embedding of samples in this changing space. In order for the smart mining method to remain practical we must balance between the speed of offline mining and the accuracy of online mining.

At one extreme, fully online mining would aim to maintain an up-to-date representation of the embedding space with a forward propagation of the entire training set after each batch has been processed. In contrast, a fully offline method might evaluate the sample embeddings once at the start of training and then use those initial

values throughout the training. While it is possible that a single batch could result in large global changes to the embedding space, this is likely to be a catastrophic behaviour for most training scenarios. Instead we expect that changes to the embedding space are gradual enough that previous embeddings can be used for some period before they become too outdated. For this purpose, we perform offline mining once at the beginning of each training epoch. Here we define an epoch as processing $O(n)$ triplets, or enough batches to allow for each training samples to be the anchor of a small number of triplets. Algorithm 6.1 provides an overview of this training process. The function at line 7 introduces an additional cost of forward propagating all training samples, building a FANNG on the sample embeddings and then constructing a neighbourhood set for each sample. Then the function at line 9 completes the mining process by producing each of the triplets that will be used throughout an epoch.

Algorithm 6.1: Network Training and Testing

Input: image set X
class labels Y
pre-trained weights θ
Output: network configuration θ

1 $X_{train} \leftarrow$ training partition of X
2 $X_{val} \leftarrow$ validation partition of X
3 $error \leftarrow 100\%$
4 $prev \leftarrow 100\%$
5 **while** $error \leq prev$ **do**
6 $\quad prev \leftarrow error$
7 $\quad S \leftarrow ConstructNeighbours(X_{train}, \theta)$
8 $\quad \kappa \leftarrow UpdateKappa(error)$
9 $\quad B \leftarrow ConstructBatches(X_{train}, S, Y, \kappa)$
10 $\quad$ **foreach** $b \in B$ **do**
11 $\quad\quad f(b, \theta) \leftarrow$ forward pass
12 $\quad\quad \theta \leftarrow$ backward pass
13 $\quad error \leftarrow Evaluate(X_{val}, Y, \theta)$
14 $X_{test} \leftarrow$ test partition of X
15 **print** $Evaluate(X_{test}, Y, \theta)$
16 **return** θ

6.2.1.1 Nearest Neighbour Set Construction

At the beginning of each training epoch we perform a full forward pass of the entire training set to generate an up-to-date feature vector for each of training sample. A FANNG can then be efficiently constructed on these vectors by using the traverse-add edge selection method given previously in Algorithm 4.5. As discussed in Chapter 4, the traverse-add algorithm can be repeatedly applied until a specified percentage success rate is reached. Experimentally we found that a success rates of 95% or higher is a broadly suitable target with no noticeable advantage to be gained from fine tuning this parameter. Once the target build percentage is achieved, our approach diverges from the approximate graph construction method that was previously described.

In the context of triplet mining, each indexing graph will enable computationally efficient collection of nearest neighbour sets that closely approximate $\mathcal{S}$. This mining will occur once at the start of an epoch and then a new graph is constructed on the next epoch. As such, it is inefficient to dedicate additional computational resources to refining the graph. Rather than applying the self query graph construction method from Section 4.4.3.2, we instead use the greedy backtrack search from Algorithm 4.3 to immediately generate a nearest neighbour set for each training sample. As with the first stage of the self query graph construction, we obtain $\mathcal{S}_i$ by passing the vertex $f(\mathbf{x}_i, \theta)$ to Algorithm 4.3 as both the query vector and the starting vertex. The algorithm is again modified to maintain and return a sorted list of the nearest vertices that are visited during the search.

Algorithm 6.2 provides an outline of the approximate construction of $\mathcal{S}$. This function adds a tuning parameter $target$ for the index construction and another parameter $cost$ for the ANN search quality. When mining only once per epoch, we have already accepted a reduction in the maximum possible accuracy of the entire pipeline. In particular, the neighbourhoods produced by this mining are only guaranteed to remain exactly the same for the first batch of the stochastic gradient descent. As such, both $target$ and U_{max} are only required to be large enough that they do not

degrade the neighbourhood sets more than the training process itself. The final parameter C_{max} is determined by the training set and is selected such that each neighbour set $\mathcal{S}_i$ is guaranteed to contain at least one positive and negative sample. Each of the iterative loops can be accelerated with a parallel implementation. This is very straightforward implementation for the second loop, since the collection of the neighbourhood sets does not modify the indexing graph.

Algorithm 6.2: Neighbourhood construction

Input: training set T
pre-trained weights θ
target build percentage $target$
maximum vertices to visit U_{max}
neighbourhood size C_{max}
Output: nearest neighbours S

$f(T, \theta) \leftarrow$ full forward pass
$E \leftarrow$ empty set of graph edges
$success \leftarrow 0\%$
while $success < target$ **do**
  $count \leftarrow 0$
  foreach $\mathbf{x} \in T$ **do**
    $E_{new} \leftarrow TraverseAdd(T, E, \mathbf{x}, rand(\mathbf{x} \in X))$
    if $E_{new} = E$ **then**
      $count \leftarrow count + 1$
    else
      $E \leftarrow E_{new}$
  $success \leftarrow 100 \cdot count/|X|$
$S \leftarrow$ empty set of nearest neighbours
foreach $\mathbf{x}_i \in T$ **do**
  $S_i \leftarrow$
  $GreedyBacktrackSearch(T, E, \mathbf{x}_i, \mathbf{x}_i, U_{max}, C_{max})$
return S

6.2.1.2 Triplet and Batch Construction

Each nearest neighbour set $\mathcal{S}_i$ is stored as a sorted array in ascending order by distance from the anchor sample $\mathbf{x}_i$. Each entry in the array holds the index of a neighbouring sample and the embedding space distance between that sample and $f(\mathbf{x}_i, \theta)$. Once a nearest neighbour set $\mathcal{S}_i$ has been computed, class label information is used to separate the neighbours into lists of positive and negative samples.

We perform a single iterative pass over the list of neighbours, while ignoring all negative samples until the first positive sample has been found. The distance to the closest positive neighbour is then multiplied by the global parameter κ to define the local exclusion boundary. Then any future negative samples that satisfy Equation 6.5 are added to the list of valid negatives. The list of positive samples does not exclude any of positive neighbours, however each positive samples is added to this list along with the current number of valid negatives. This additional information allows us to track how many smart negatives can be used with each positive sample to produce hard triplets. This process is achieved without performing any distance calculations, since the required distances have already been computed during the construction of the neighbourhood set.

In order to construct enough triplets for the current epoch, we need to assign ν pairs of samples to each anchor. For each anchor we use the first ν negative samples from the list of valid negatives. And for each negative sample we complete the triplet by using the first positive sample that can be used by the chosen negative sample. In the rare cases where there are fewer than ν negative samples available, random triplets are constructed instead. Additionally, if there are no valid positive samples associated with a chosen negative sample, then a positive sample is uniformly selected at random from the set $\mathcal{T} \backslash \mathcal{S}_i$. Each negative sample is used no more than once with any given anchor during each epoch, however positive samples can be used multiple times with the same anchor. The unique pairings of anchors and negative samples ensures that no triplets are repeated during an epoch. Algorithm 6.3 details the complete triplet selection process. Finally, in order to satisfy the requirements of the stochastic gradient descent, we randomly shuffle the set of triplets to distribute the anchors throughout each training batch.

Algorithm 6.3: Triplet selection for smart mining

Input: training set T
nearest neighbours S
class labels Y
boundary scale κ
triplets per anchor ν
Output: Triplet batches B

$B \leftarrow$ empty set of triplets
foreach $x_i \in T$ **do**
  $P \leftarrow$ empty list of positive samples
  $N \leftarrow$ empty list of smart negatives
  $bound \leftarrow$ **max**
  foreach $\{x_j, dist\} \in S_i : i \neq j$ **do**
    if $y_i = y_j$ **then**
      $P.append(\{x_j, |N|\})$
      if $bound =$ **max then**
        $bound \leftarrow \kappa \cdot dist$
    else if $dist > bound$ **then**
      $N.append(x_j)$
  for $t = \{1, \ldots, \nu\}$ **do**
    if $t \leq |N|$ **then**
      $x^p \leftarrow rand(x_j \in T \backslash P : y_i = y_j \wedge i \neq j)$
      $x^n \leftarrow N[t]$
      for $\{x_j, range\} \in P$ **do**
        if $t \leq range$ **then**
          $x^p \leftarrow x_j$
          break
    else
      $x^p \leftarrow rand(x_j \in X : y_i = y_j \wedge i \neq j)$
      $x^n \leftarrow rand(x_j \in X : y_i \neq y_j)$
    $B.add(\{x_i, x^p, x^n\})$
$shuffle(B)$
return B

6.2.1.3 Automatic Parameter Selection

The global tuning parameter κ is used to adjust the scale of the exclusion boundaries that are centred on each triplet anchor. At the beginning of training we would like large boundaries that exclude most of the harder negative samples. As training progresses, κ should be reduced to allow for increasingly harder negative samples to be mined. As a naive way to achieve this, we use the following linear equation to

control the rate at which κ decreases

$$\kappa = \frac{\kappa_{max} - \kappa_{min}}{\varepsilon_{max} - \varepsilon_{min}}(\varepsilon_{max} - \varepsilon) + \kappa_{min}. \tag{6.6}$$

This linear equation parametrises κ in terms of the current epoch ε. This also requires the hand tuning of the four constants defining each of the initial value scale for the exclusion boundaries κ_{max}, the final scale of the exclusion boundaries κ_{min}, the first epoch that will use mined triplets ε_{min} and an estimate of the final training epoch ε_{max}. With the selection of appropriate values, the triplet mining converges on a model that outperforms random triplets. This relative reduction in test set error validates that our mining methodology is able to address the vanishing gradient problem.

A more robust solution for scaling the exclusion boundaries is to automatically estimate a value of κ that will produce triplets of a suitable difficulty for the current epoch. One possible goal of this estimate would be to ensure that the training set error remains consistent with the current validation set error. A divergence between these two values indicates that either the network is over-fitting when the training error is lower than the validation error or that the mined triplets are becoming too hard when the training error is higher than the validation error. Again the naive approach is to use a linear model, but now we will automatically adjust the following model at the beginning of each epoch

$$\boldsymbol{\kappa} = \alpha\boldsymbol{\delta} + \beta. \tag{6.7}$$

The model parameters α and β are estimated by computing the least-squares solution for a vector of recent training errors $\boldsymbol{\delta}$ and their associated κ. Once we have solved for the current values of the model parameters, we can estimate κ as follows

$$\kappa = \alpha\delta_t + \beta. \tag{6.8}$$

This equation assigns a value of κ for the current training epoch using the solved model parameters and a target training error δ_t. For our example goal of reducing divergence between the training set error and the validation set error, δ_t is assigned

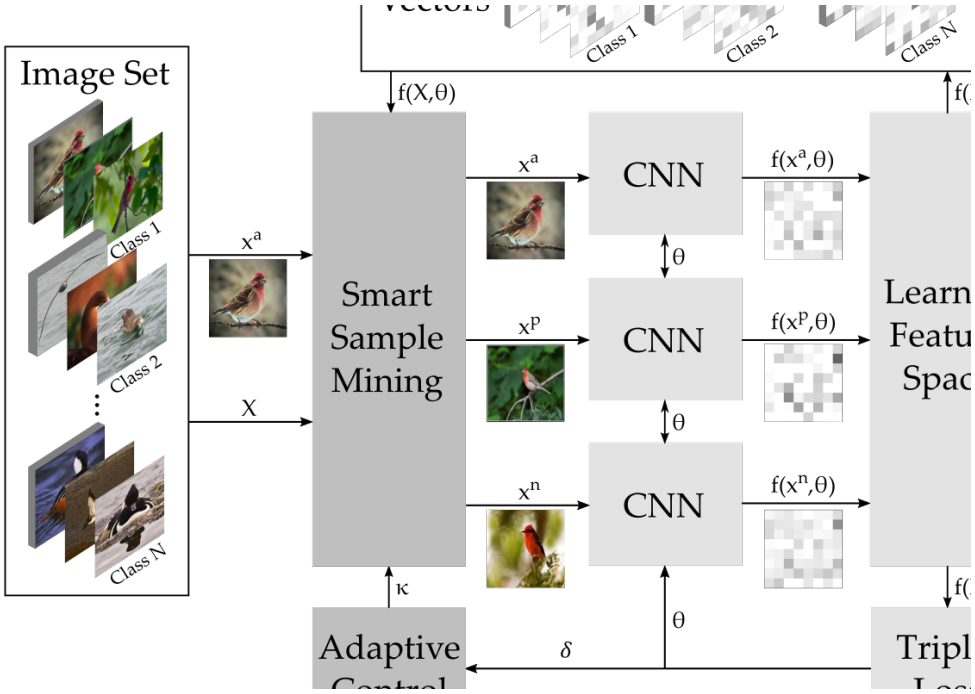


FIGURE 6.8: Our proposed model introduces a smart mining procedure that is capable of quickly searching the set of feature vectors to select effective samples. The hyper-parameter κ is estimated by the adaptive controller based on the training error δ.

as the validation error of the previous training epoch. Figure 6.8 illustrates how this adaptive tuning of κ fits into the triplet network pipeline. We can see that the triplet selection loop is typically closed by feature space mining, but now the adaptive controller is utilising the training errors as additional information for the loop closure.

Typically, as training progresses and the embedding improves, it is expected that both the training and validation error will steadily decrease. While this is desirable for the validation error, a low training error indicates that most of the epoch has been spent processing triplets that did not make an impact on the training. Hard mining does marginally address this issue and will usually result in a training error that is higher than the validation error. However, hard triplets are usually restricted to around 1% of each training batch to avoid over-fitting with hard outliers. Semi-hard mining does allow for more mined triplets to be included in each batch, but the training can quickly and catastrophically diverge if this is pushed too far. Using our adaptive control of κ we can deliberately separate the training and validation errors so that the training error is maintained around a particular target, while the

validation error continues to decrease. This is achieved by assigning δ_t in Equation 6.8 to be the fixed training error that we would like to target. Experimentally we found that a target of between 50% and 75% training error will still provide stable convergence, while also having the additional benefits of producing more accurate embeddings in far fewer epochs. We also found that in order to maintain these high training errors, it is best to use batches that comprise of between 50% and 100% mined triplets. However, the inclusion of as little as 2% mined triplets in each batch is enough to provide some control over the training error.

Figure 6.9 provides a comparison between hand-tuned and adaptive selection of κ. Both models are initialised at the beginning of the third training epoch with predefined values. Triplet mining then occurs on each subsequent epoch and the training results from the previous five epochs are used to update the adaptive model. The training errors seen in this figure indicate the fraction of triplets from each training batch that are producing a non-zero gradient. The validation error is produced by evaluating the current embedding with random triplets from an unseen set of samples. As such, the training error gives a measure of how much work is being done

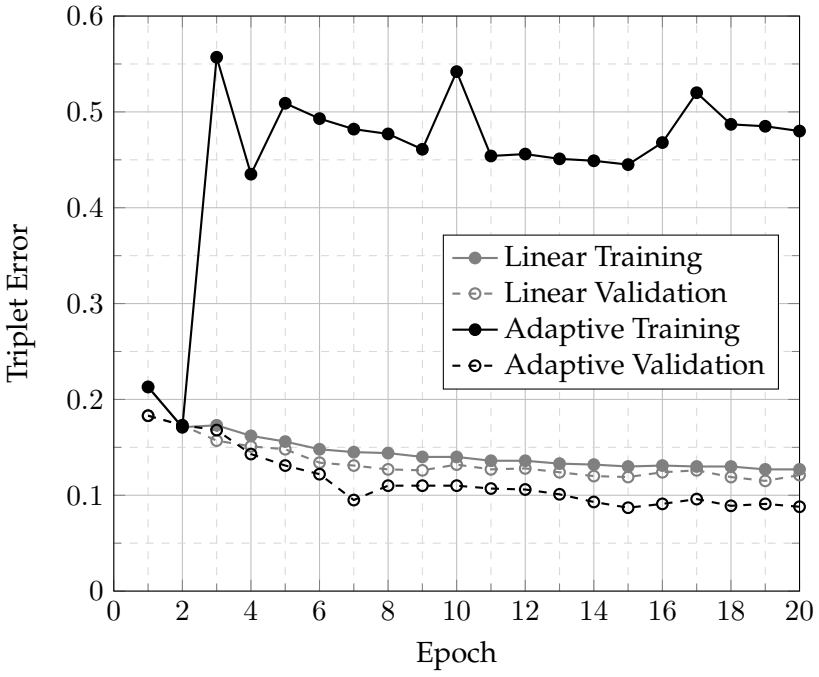


FIGURE 6.9: A comparison of training performance using hand-tuned and adaptive selection of κ. Training and validation error is shown for the first 20 epochs while training on the CUB birds dataset.

to improve the embedding and the validation error provides an inverse measure of the embedding quality. In this figure, we see that the adaptive method results in a higher quality embedding than the one produced by mining with the hand-tuned linear model. Additionally, the higher training error and the steeper descent of the validation error demonstrate that the improved results of the adaptive model can be achieved while also using fewer training epochs. These results validate that the adaptive method is able to select harder triplets, while also avoiding triplets that are so hard that the training diverges.

6.2.2 Runtime Complexity

The computational cost of triplet mining is dominated by the pairwise distance calculations that are used to determine if each triplet is well ordered. As such, a naive hard mining algorithm that selects $O(n)$ triplets from a training set of n samples will have a worst case computational complexity of $O(n^3)$ on any given epoch. If these samples are equally distributed between C classes, then for each anchor evaluating all positive and all negatives samples has complexity of $O(n/C)$ and $O(n - n/C)$ respectively. Given that there are $O(n)$ anchors, the total complexity per epoch can instead be expressed as $O\big(n(n - n/C)n/C\big)$. Now we consider a dataset that contains many classes, each with very few samples. This scenario can be expresses with limit $C \to n$, so then $n/C \to 1$, and the overall complexity reduces to a best case of $O(n^2)$.

The smart mining algorithm adds an additional cost of constructing a nearest neighbour index at the beginning of each epoch. We find that even for feature embeddings with thousands of extrinsic dimensions, the average degree of the graph vertices remains low (fewer than one hundred edges). Typically, exhaustive index construction has a computational complexity of $O(n^3 \log n)$ due to the sorting of all n^2 pairwise distances. However, we can instead guarantee a worst case complexity of $O(n^2)$ by building an approximation of the index. This is achieved by performing at most $O(n)$ construction iterations, with each training sample being used as the starting vertex for the traverse-add edge selection $O(1)$ times per iteration. With the nearest neighbour index, we can construct each of the n neighbourhood sets with far fewer than $O(n)$ steps in each of the greedy backtracking searches. The final step

of constructing the triplets for each anchor can also be performed with worst case complexity $O(n)$ regardless of class distribution. Since each stage of the smart mining process has a worst case complexity of $O(n^2)$, and given that this is the best case complexity for the naive hard mining approach above, we can conclude that our method is more computationally efficient.

Semi-hard mining methods are able to achieve a lower algorithmic complexity by limiting brute force mining of negative samples to within each training batch. If we consider an epoch that is divided into B_ε batches, then each batch will contain $O(n/B_\varepsilon)$ anchors. Additionally, for each anchor there will be an $O(n/B_\varepsilon)$ cost for the $\arg\max$ used to select the negative sample. Hence, the a total computational complexity for an epoch is then $O\big(B_\varepsilon(n/B_\varepsilon)^2\big)$, which simplifies to $O(n^2/B_\varepsilon)$. Although $O(n^2/B_\varepsilon) < O(n^2)$, we note that larger batches (i.e. smaller B_ε) can reduce training error up until performance is limited by the naive use of $\arg\max$. Overlooking this limitation, we can take the limit as $B_\varepsilon \to 1$. As the semi-hard mining complexity approaches $O(n^2)$, the number of triplets considered by the semi-hard mining approaches that of both the naive and smart mining.

Lastly, it is important to consider the number of epochs to train these triplet networks until convergence. In practice, when using GPU accelerated code and the datasets considered in Section 6.4, our triplet selection accounts for less than 1% of the total epoch runtime. Instead, the majority of this time is spent on the forward and backward propagation of the triplets. This represents a significant advantage for our adaptive controller, which is able to considerably reduce the number of epochs needed to reach convergence. When comparing to other mining methods with a similar complexity per epoch, producing a high quality embedding in comparatively fewer epochs will greatly reduce the overall training time. Indexing the entire training set could result in a higher upfront cost for our method, but by utilising the additional information that is provided by the index we are able to learn the embedding at a much faster rate and hence with fewer computations overall.

6.3 Deep Learning with Gaussian Kernel Loss

Triplet loss enables effective feature learning by shaping an embedding space with the pushing and pulling of labelled samples. Additionally, our smart mining method has shown that a relatively small nearest neighbour set around each sample will contain the key information needed for progressing the training. As such, we would like to take full advantage of the information being provided by the ANN index. Our smart mining heuristic is focused only on triplets that will advance the formation of the embedding space. This overlooks the fact that other nearby triplets could be well ordered on one epoch and then unordered on the next. Rather than revisiting these triplets, a more direct approach is to use a training loss that considers the presence of all nearby samples, not just the ones that currently need reordering.

We use the weighted sum of Gaussian distances between a particular sample and all other samples in the training set to express a Gaussian kernel loss. For the labelled training sample $(\mathbf{x}_i, y_i) \in \mathcal{T}$, we compute the current probability distribution over the class labels by summing the influence of samples from each given class and then normalising the result. As such, the probability that x_i is in class y is given as

$$\Pr(y_i = y) = \frac{\sum\limits_{\substack{(\mathbf{x}_j, y_j) \in \mathcal{T} \\ \mathbf{x}_j \neq \mathbf{x}_i,\ y_j = y}} w_j \cdot G\big(f(\mathbf{x}_i, \theta), f(\mathbf{x}_j, \theta)\big)}{\sum\limits_{\substack{(\mathbf{x}_k, y_k) \in \mathcal{T} \\ \mathbf{x}_k \neq \mathbf{x}_i}} w_k \cdot G\big(f(\mathbf{x}_i, \theta), f(\mathbf{x}_k, \theta)\big)}. \tag{6.9}$$

For training CNNs, this equation is implemented in a classification layer that follows from the embedding layer. The classification layer is then followed by a Gaussian kernel loss function that is defined as the negative logarithm of the ground truth probability with

$$L(\mathbf{x}_i) = -\ln\big(\Pr(y_i = y)\big). \tag{6.10}$$

Using this loss function during training will produce an embedding space that maximises the probability of the ground truth labels. Additionally, this allows for Equation 6.9 to be used as a classifier for unseen samples. An overview of our training pipeline can be seen in Figure 6.10.

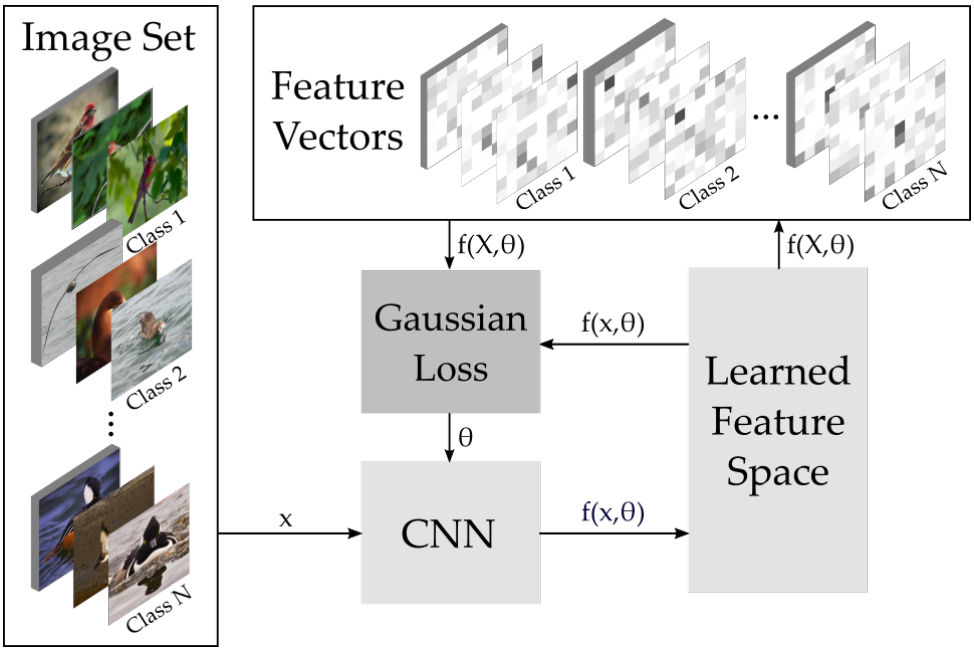


FIGURE 6.10: Gaussian kernel loss is able to utilise information from the full set of current feature correspondences in order to directly refine the feature space.

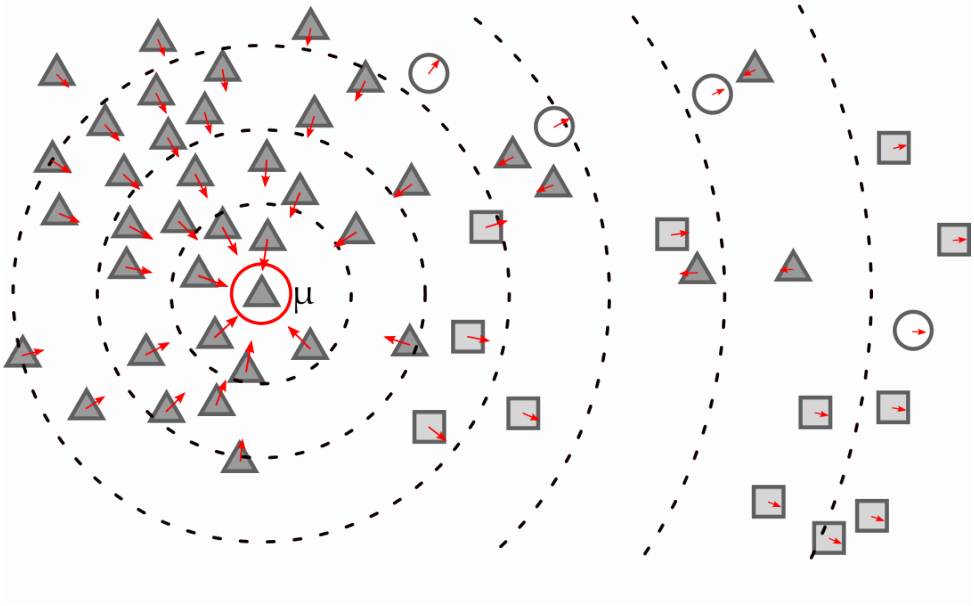


FIGURE 6.11: Level sets of embedded samples are influenced by the Gaussian kernel centred at μ.

In Equation 6.9 the distance function G represents a Gaussian kernel centred at the embedding of each sample in the training set. We use the Euclidean distance between a sample embedding $f(\mathbf{x}, \theta)$ and a Gaussian centre $\boldsymbol{\mu}$ to compute the kernel function as

$$G(f(\mathbf{x}, \theta), \boldsymbol{\mu}) = \exp\left(\frac{-dist\big(f(\mathbf{x}, \theta), \boldsymbol{\mu}\big)}{2\sigma^2}\right). \tag{6.11}$$

Here σ is a global standard deviation value that controls the decay rate for the influence of the kernels. Figure 6.11 illustrates the influence of a particular Gaussian kernel on a local neighbourhood of samples. The influence of each kernel is also adjusted individually in Equation 6.9 where w_i is a learned weight for the Gaussian kernel centred at $f(\mathbf{x}_i, \theta)$.

Importantly, the additional classifier weights $\mathbf{w}$ can be learnt end-to-end along with the other network weights θ. This is due to the differentiability of the classification layer, which allows the network to remain fully differentiable for the backward propagation of gradients. Additionally, for the datasets considered in Section 6.4, the additional memory requirement of an extra weight per sample does not impact our ability to train the network on a single GPU. If this did become an issue, then additional memory could be freed at the expense of model capacity by reducing the dimensionality of the fully connected embedding space layer.

6.3.1 Approximate Computation of Gaussian Kernel Losses

Computing a ground truth probability for the Gaussian kernel loss requires the two summations seen in Equation 6.9. Since the Gaussian kernels are centred on each of the training samples, these summations are computed across the entire training set. This results in the loss function having a computational complexity that is roughly equivalent to the intractable hard negative mining discussed in Section 6.2.2. However, we can utilise the fact that the value returned from the Gaussian kernel function in Equation 6.11 approaches zero as the distance between the embedded sample and the kernel centre increases. As was the case for the hard negative mining, we are much more interested in samples that are embedded closer to the sample being evaluated. In this case, closer samples will often represent a kernel function with

a significant influence over the result of the summations, while distant samples are likely contribute very little.

Utilising the same approach described in Section 6.2.1.1, we can construct an indexing graph over the embedding space and then efficiently approximate a set of nearest neighbours $\mathcal{S}_i$ for each embedded sample $f(\mathbf{x}_i, \theta)$. This then allows us to rewrite the classification function as

$$\Pr(y_i = y) \approx \frac{\sum\limits_{\substack{(\mathbf{x}_j, y_j) \in \mathcal{S}_i \\ \mathbf{x}_j \neq \mathbf{x}_i,\ y_j = y}} w_j \cdot G\big(f(\mathbf{x}_i, \theta), f(\mathbf{x}_j, \theta)\big)}{\sum\limits_{\substack{(\mathbf{x}_k, y_k) \in \mathcal{S}_i \\ \mathbf{x}_k \neq \mathbf{x}_i}} w_k \cdot G\big(f(\mathbf{x}_i, \theta), f(\mathbf{x}_k, \theta)\big)}. \tag{6.12}$$

Figure 6.12 plots trade-offs between the size of $\mathcal{S}_i$ and the error in Equation 6.12 at various times throughout the training phase. In all cases, we find that increasing the size of the nearest neighbour sets will offer diminishing returns for the classification accuracy. This is because each $\mathcal{S}_i$ is constructed with a greedy backtracking search that expands outward from the associated $f(\mathbf{x}_i, \theta)$. As such, the probability

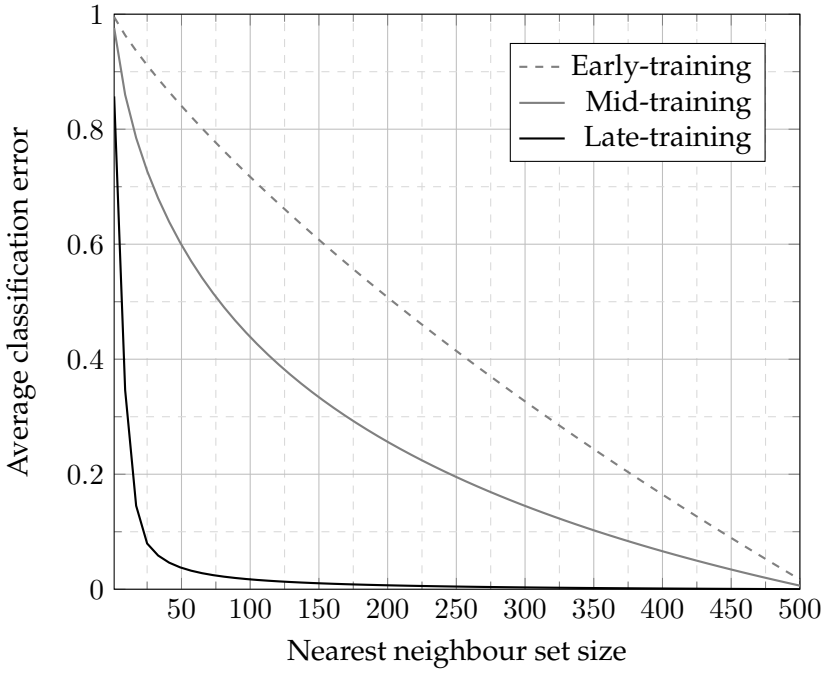


FIGURE 6.12: Increasing the number of samples returned by the ANN searches results in a reduced classification error when computing Gaussian kernel losses.

of finding nearby neighbours will decrease throughout the search. Additionally, as training progresses and strong class clusters begin to form, we find that the average classification error reduces with respect to a fixed size nearest neighbour set. Intuitively, this indicates that when the network becomes better at classifying training samples it also becomes easier to estimate the classification distributions. Ultimately, the size of these search neighbourhoods must be small enough to remain computationally tractable, but large enough to give a good approximation of the classification function throughout the network training.

6.3.2 Indexing and Updating the Embedding Space

In Section 6.2.1 we discussed the trade-off that needs to be made between online and offline construction of the ANN indexing graph. In the case of triplet networks, stochastic gradient descent requires the usage of training batches that will cause regular changes to the embedding space throughout each epoch. However, computing the triplet losses within each batch does not require any additional information about the structure of the embedding space. As such, it was acceptable to pre-compute triplets as an offline process at the start of each epoch. Unfortunately, this is not the case for the Gaussian kernel loss. The approximate classification function that we have proposed for the Gaussian kernel loss requires access to a local neighbourhood of samples as well as the up-to-date embeddings of these samples. These requirements push the Gaussian kernel loss towards using an online method for the ANN index construction.

In order to compute the training losses we must produce a nearest neighbour set $\mathcal{S}_i$ for each training sample $\mathbf{x}_i$ in each batch of an epoch. However, to keep the training tractable we cannot rebuild the nearest neighbour index as frequently as once per batch. The main issue with using less frequent graph construction is that changes in the distances between sample embeddings will invalidate some of the information that was used to construct an efficient index. Ultimately, this will either result in lower efficiency for the ANN searches or a less accurate approximation of the nearest neighbour sets. In fact, the nature of ANN search is already such that we are evaluating this same trade-off when choosing to approximate the summations in the

classification function. As such, we can simply continue to assign the number of search iterations such that the classifications remain accurate enough to be useful. In general, we would prefer to search a larger neighbourhood at a slightly higher computational cost. Using a smaller neighbourhood means that we could miss samples that are close enough to still impact the overall accuracy of a classification.

We find that the changes in the embedding space are gradual enough that a moderately sized neighbourhood allows for previous graphs to be used for several epochs before they require rebuilding. Specifically for the datasets considered in Section 6.4 we achieve competitive results when rebuilding the graph every 5-10 epochs and searching for neighbourhoods of between 100 and 500 indexed samples. These values allow for significant overlap between the neighbourhood sets that would be produced across any batch prior to rebuilding the graph. Each time that the graph is reconstructed, the previous edge lists can be used as a starting point for the traverse-add edge selection algorithm. Again, this utilises the gradual rate at which the embedding space changes and allows for the graph to retain any edges that will still be useful in the updated index.

Having access to a nearest neighbour set $\mathcal{S}_i$ is only useful when we also have up-to-date locations for the associated Gaussian kernels that are centred at each $f(\mathbf{x}_j, \theta) \in \mathcal{S}_i$. Again we find that the training becomes intractable if we forward propagate hundreds or thousands of additional samples along with each training batch. We instead rely on the robust nature of our training method and accept the usage of outdated sample embeddings for at least some of the loss calculations. While these values will not be exact, gradual changes to the embedding space allow for previous embeddings to be used for some period before they impede the classification. We update an array of the Gaussian kernel centres after the full forward pass that is required for the index construction. These entries can also be updated throughout the training for each sample that is contained within the forward pass of a particular batch. In general, we find that it is not necessary for the Gaussian kernel centres to remain up-to-date at all times in order for the training to converge.

6.4 Experimental Results

Here we explore the performance of both the smart mining method applied to triplet loss, and the Gaussian kernel loss. Through experimental results and discussion, we have evaluated the applicability of these methods for the task of transfer learning. Quantitative comparisons are provided in order to rank existing approaches that have demonstrated success in this area. The learned feature spaces are compared with a number of transfer learning metrics that score the embedding quality using previously unseen test classes. Qualitative results are also presented as a means of visually inspecting the functionality of our proposed methods.

The datasets used throughout this section are described in Table 1.4. Each sample image is scaled to a size of 256×256 pixels for input in the CNNs. We train and evaluate using the full images rather than cropping with the tight bounding boxes that are provided as this presents a more challenging task of focusing on the foreground class with a larger number of background pixels. Half of the classes are used with their ground-truth labels for training the networks, while the other half of the classes are withheld for the testing phase. The basic network architecture used throughout our experiments is the GoogLeNet model [99] from Szegedy et al. We initialize the network using pre-trained weights from Szegedy et al. that were produced for softmax classification by training with the ImageNet [113] dataset. The classifier is replaced with one of our training losses and the weights within the fully connected embedding space layer are then randomly initialized. Overall, this experimental set-up is consistent with a number of earlier publications in this area, which allows for simple and fair comparisons with these existing results.

For our smart mining method, we allow the training phase to run for a maximum of 20 epochs or until an earlier convergence is indicated by a repeated decrease in the validation score. Triplet mining is completely disabled during the first two training epochs, so each batch is formed entirely from random triplets. Following from previous triplet mining methods, we assign the intraclass distance margin m to a value of 0.2. The learning rate for the network is initially set to 0.1 and then halved every three epochs, while a constant weight decay of 0.0005 is also used. For the

randomly initialized fully connected layer, the learning rate is multiplied by 10 in order to accelerate convergence. Lastly, we use a 64 dimensional embedding space, since higher dimensional features produced by triplet networks have not shown an improvement over more compact features.

Both the smart mining and the Gaussian kernel loss computations use a FANNG to approximate a set of 100 nearest neighbours around each training sample. The indexing graphs are constructed until at least 98% of a traverse-add iteration is not finding additional edges. For tractability of training the Gaussian kernel network, the graph construction is only performed once every 10 epochs. Additionally, the network is trained for up to 50 epochs, with a learning rate of 0.00001 and weight decay of 0.0002. For embedding learning tasks, we initialise all of the Gaussian kernel weights to a value of 1 and select a standard deviation σ that is appropriate for the embedding dimensionality. The standard deviation is set to a value of 10 for a 64 dimensional embedding space, and then increased towards a value of 30 when the dimensionality is increased towards 1024 dimensions.

6.4.1 Quantitative Results

To compare the quality of the learned feature spaces we use two different measures that both evaluate performance with a transfer learning task. Firstly, each sample from the previously unseen test classes is passed through a trained model in order to produce a full set of feature vectors $f(X, \theta)$. The first evaluation method we use is a single valued measure called Normalised Mutual Information (NMI). NMI is calculated as the reduction in class label entropy when information is provided about local clusters. We compute these clusters on the embeddings of the test samples using a standard k-means clustering algorithm and with k equal to the number of test classes. NMI can then be computed with

$$NMI\big(f(X,\theta), Y\big) = \frac{2\Big(H(Y) - H\big(Y|f(X,\theta)\big)\Big)}{H(Y) + H\big(f(X,\theta)\big)}. \tag{6.13}$$

Here $H(Y)$ is the entropy of the class labels and $H\big(Y|f(X,\theta)\big)$ is the entropy once the clustering is taken into account. In the context of transfer learning, NMI provides a measure of how well the test samples have formed class clusters within a learned embedding space. The second metric we use is recall@r, which is often used to evaluate ANN search performance when approximate distances are being used. Here we compute recall@r as the percentage of embedded test samples that have another sample of the same class within their r nearest neighbours. This is similar to a nearest neighbour classifier, however correct classification only requires that the true class label is given a non-zero probability.

Tables 6.1 and 6.2 present the NMI and recall values for our smart mining method and for a number of other published results that have proven to be successful approaches to feature learning. Each of these other methods utilises a triplet network in addition to using either semi-hard mining [108], lifted structured feature embeddings [109], N-pair loss [114] or global loss [110]. The smart mining results are shown for both the linear and the automated parameter selection. The adaptive controller

TABLE 6.1: Triplet network performance on the CUB birds dataset.

Method	NMI	R@1	R@2	R@4	R@8
Semi-hard [108]	55.38	42.59	55.03	66.44	77.23
Lifted structure [109]	56.50	43.57	56.55	68.59	79.63
N-pair loss [114]	57.24	45.37	58.41	69.51	79.49
Smart mining + Linear control	58.10	45.90	57.65	69.63	79.83
Global loss [110]	58.61	**49.04**	**60.97**	**72.33**	**81.85**
Smart mining + Adaptive control	**59.57**	47.32	59.30	71.47	81.60

TABLE 6.2: Triplet network performance on the Stanford cars dataset.

Method	NMI	R@1	R@2	R@4	R@8
Semi-hard [108]	53.35	51.54	63.78	73.52	82.41
Lifted structure [109]	56.88	52.98	65.70	76.01	84.27
N-pair loss [114]	57.79	53.90	66.76	77.75	86.35
Smart mining + Linear control	58.24	56.11	68.34	77.99	85.92
Global loss [110]	58.20	61.41	72.51	81.75	88.39
Smart mining + Adaptive control	**62.23**	**67.20**	**77.99**	**85.92**	**91.40**

provides a significant boost to both the NMI and the recall scores, which indicates that a robust embedding space has been created. This is also achieved in considerably fewer epochs than are needed for the linear controller. Compared to the existing triplet methods, smart mining provides the best performance in terms of NMI score. However, for recall within a small neighbour set the addition of global loss does improve the results on one of the dataset. Since the smart mining method remains competitive in terms on NMI and for larger recall neighbourhoods, this could indicate that the exclusion boundaries did not shrink enough before convergence. Tighter exclusion boundaries would allow for mining with negative samples that are very close to the anchor sample, and likely to be within a small set of nearest neighbours. While this mining would occur the training classes, rather than the test classes, the learning of fine grained structures in the embedding space could still be transferable. Alternatively, it is also possible to combine the global loss with a triplet that is utilising the smart mining method.

Tables 6.3 and 6.4 contain the experimental results for learning methods that fully utilise local neighbourhoods, as opposed to the three samples used in the triplet loss. We compare our Gaussian kernel loss with the existing facility location clustering [112] method. These approaches are able to successful convert the additional information they have access to into a more robust embedding space than those produced by the triplet networks. We find that despite existing triplet methods and

TABLE 6.3: Clustering performance on the CUB birds dataset.

Method	Embedding Dimensions	NMI	R@1	R@2	R@4	R@8
Facility location clustering [112]	64	59.23	48.18	61.44	71.83	81.92
Gaussian kernel loss	64	61.26	51.15	64.64	75.57	84.72
	1024	63.95	57.22	68.75	79.12	87.14

TABLE 6.4: Clustering performance on the Stanford cars dataset.

Method	Embedding Dimensions	NMI	R@1	R@2	R@4	R@8
Facility location clustering [112]	64	59.04	58.11	70.64	80.27	87.81
Gaussian kernel loss	64	62.15	71.05	80.74	88.06	92.79
	1024	65.30	79.65	87.33	92.36	95.65

facility location clustering both receiving no additional benefits from using a higher dimensional embedding space, our Gaussian kernel method can make use of this additional model capacity. Even without the increased dimensionality, the Gaussian kernel loss consistently produces higher scores on our two evaluation metrics. This highlights that even for these datasets with a relatively small number of classes, and the small number of samples per class, a good loss function and training regime can still utilise additional model capacity without over-fitting to the data.

6.4.2 Qualitative Results

Visual inspection is a useful tool for quickly absorbing large amounts of information and is well suited for datasets of images. Here we present two collections of samples in order to assist in understanding the functionality of our proposed methods. Firstly, we observe the adaptability of our smart mining methodology using sets of mined triplets, which have each been collected from across a number of training epochs. Secondly, we inspect the feature space clusterings that are produced by our Gaussian kernel method. We present two dimensional projections of the feature space that have been produced using the t-Distributed Stochastic Neighbour Embedding (t-SNE) [115] method.

Figure 6.13 illustrates the behaviour of the smart mining algorithm with a collection of mined triplets. At the top of each set of seven images is an anchor sample that has been randomly sampled from across the entire training set. Each associated row then contains the positive and negative samples that complete each triplet. Both of these mined samples have been deemed to be of a suitable difficulty for the specified epoch. As such, the smart mining algorithm guarantees that the embedding of each negative sample is closer to the anchor than the associated positive sample. Additionally, each of the positive samples is as close to the anchor as possible, while still maintaining this distance relationship. Lastly, because there is no direct relationship between the selection of particular positive and negative samples, there are no guarantees over their degree of visual similarity. It is just as likely that these samples are embedded in opposing directions from the anchor, as it is that they are found in the same direction.

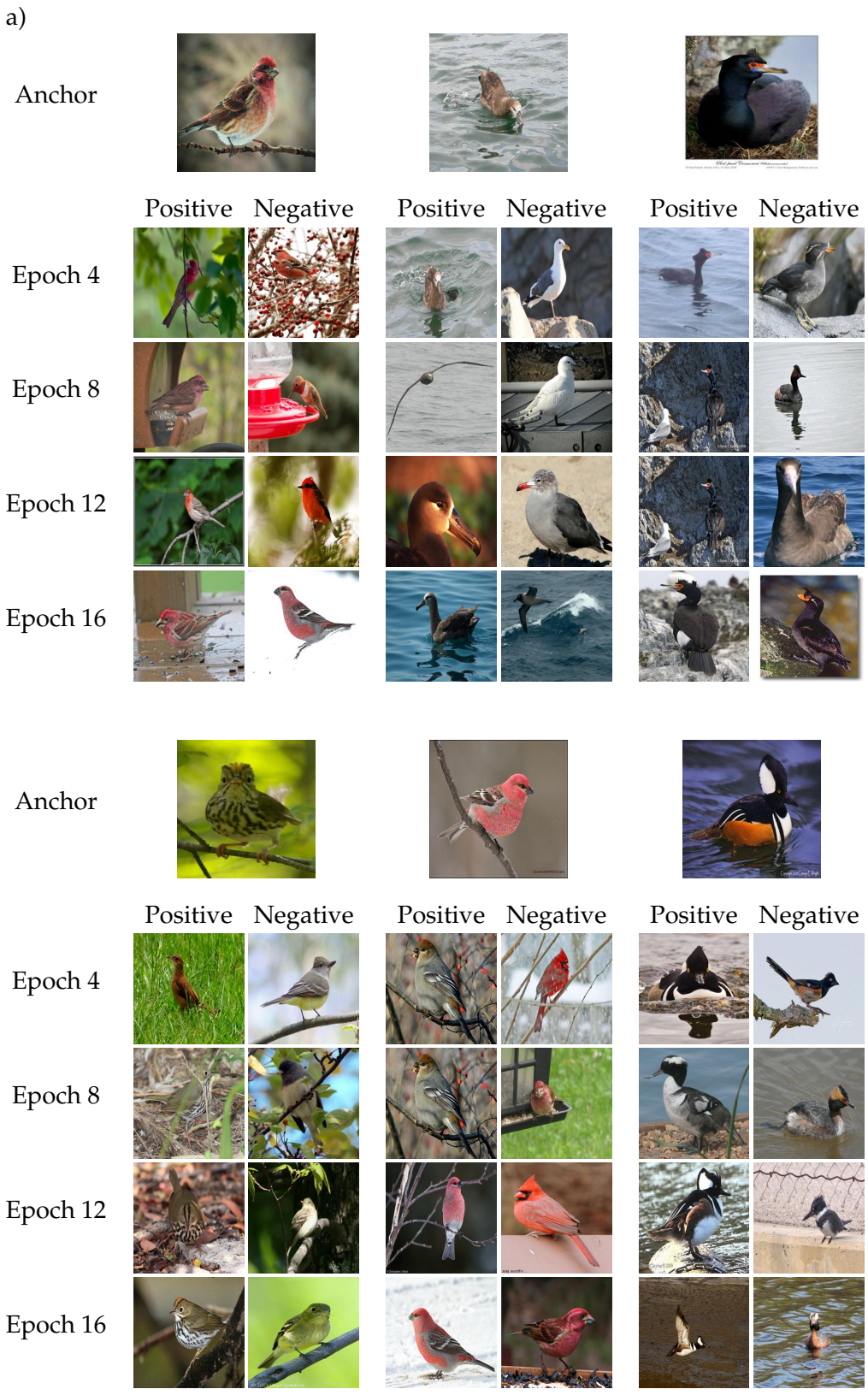
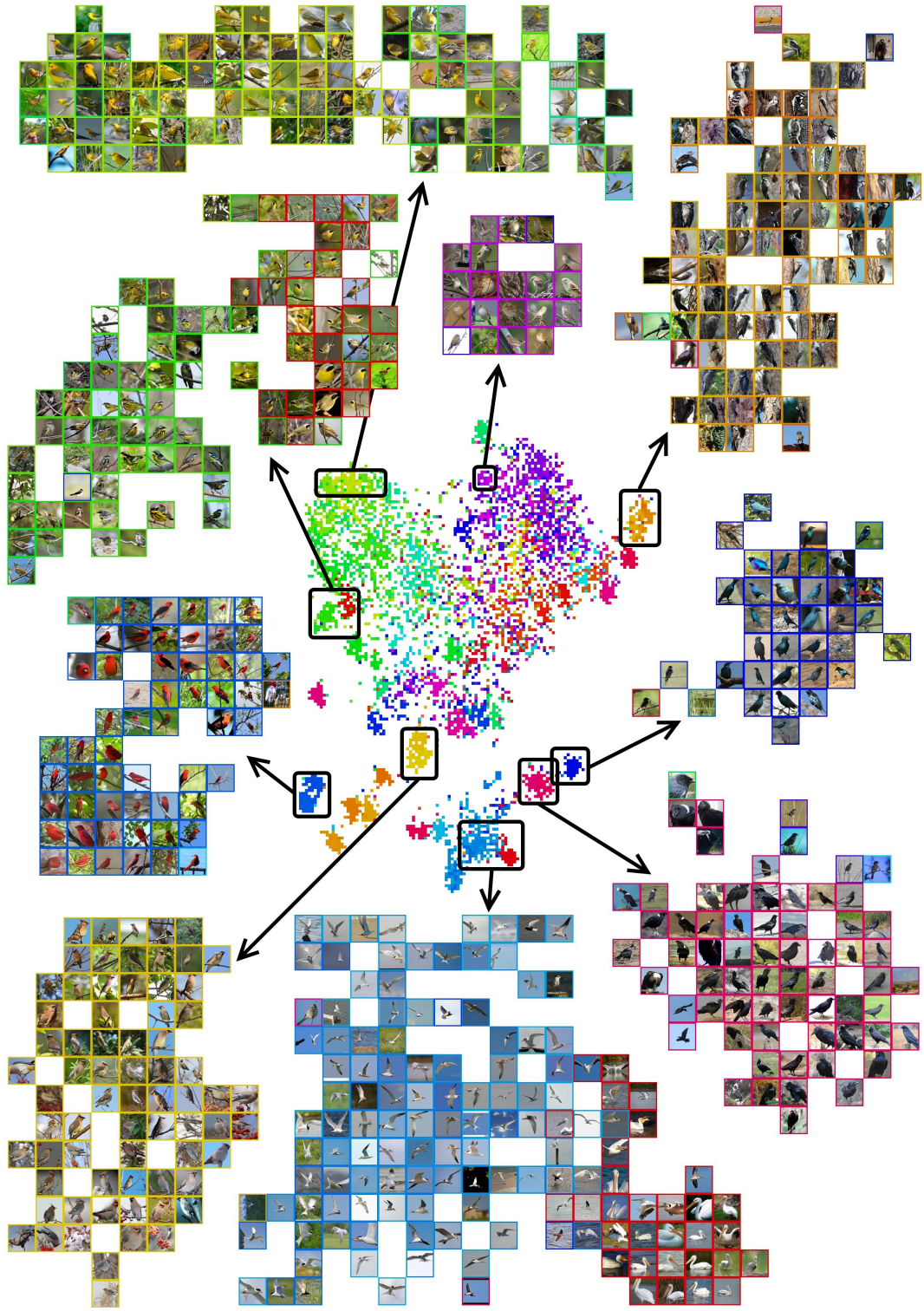

a)
Anchor
Positive
Negative
Positive
Negative
Positive
Negative
Epoch 4
Epoch 8
Epoch 12
Epoch 16
Anchor
Positive
Negative
Positive
Negative
Positive
Negative
Epoch 4
Epoch 8
Epoch 12
Epoch 16

FIGURE 6.13: Triplets mined by our smart mining methodology after increasing amounts of training. a, b) Triplets of samples from the CUB birds and Stanford cars datasets respectively.

In Figure 6.13a we see that the negative samples tend to become more similar in appearance to the anchors. Additionally, this also allows the anchors to share more fine-grained characteristics with their positive samples. Likewise, on earlier epochs the selection of negative samples from outside a larger exclusion boundary requires that the positive samples are more dissimilar. This can be seen with differences in very coarse characteristics such an large variations in colour or pose. Contrasting these results with the triplets seen in Figure 6.13b highlights some key differences between the two datasets. We see far more repetition of the positive samples in Figure 6.13b, which indicates that very few positive samples are embedded at a greater distance than the mined negative samples. The consistent differences in coarse characteristics between the anchor and positive samples also indicates either that most of the harder negatives have already been successfully mined, or that the network has learned that colour and pose are poor indicators for these classes. Overall, the smart mining method is able to adapt to changing feature spaces and present triplets at a suitable difficulty for the current embedding structure.

Figure 6.14 demonstrates the clusters that are formed when training with our Gaussian kernel loss. Each sample from the unseen test sets has been mapped to a learned feature space and then projected down into two dimensions with t-SNE. A unique colour has been assigned to each test class at random and then used to depict the relative embedding locations and as a border for corresponding sample images. These regions of sample images have been chosen to highlight visually interesting regions of the embedding space. While the t-SNE projection does attempt to retain all feature space structures, it is impossible to fully represent the complexity of these high dimensional spaces in this way. Even so, most regions of the projected space do still contain homogeneous sets of images. Within these regions it is also possible to see gradual changes in coarse features such as colour and pose.

a)



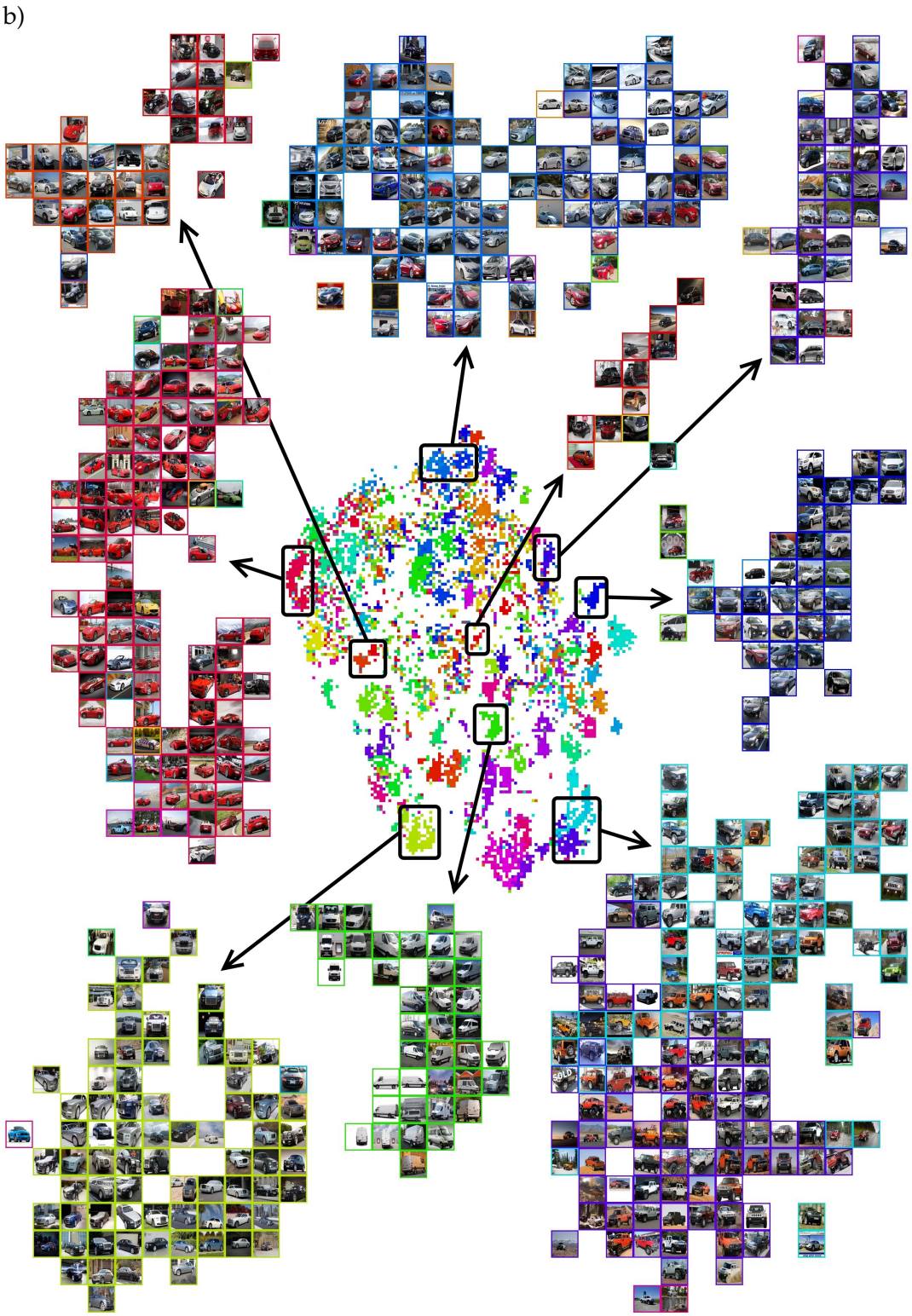


FIGURE 6.14: Visualisation using t-SNE dimension reduction. a, b) Feature space clusters from applying our Gaussian kernel method to CUB birds and Stanford cars images respectively.

Chapter 7

Conclusions

Consistent and sustained growth in the size and complexity of raw datasets has kept pace with the concurrent advancements in computational hardware. As such, the processing of large and high dimensional datasets still remains a significant bottleneck in many computational pipelines. Throughout this thesis we have analysed, explored and exploited the intrinsic structures of high dimensional image data. This has informed the development of efficient algorithms that achieved new state-of-the-art results in processing and summarising visual data.

In Chapter 2, we introduced several fundamental properties of high dimensional data. The development of an efficient Hausdorff dimensionality formulation then allowed for further analysis of the structures found in real world computer vision datasets. Analysis of large datasets was made feasible by our GPU accelerated implementation of this measure. In Chapter 5 these analysis techniques were then reapplied in order to contrast the feature spaces that are created by binary vectors.

Due to the prevalent use of feature correspondences in many vision related tasks, nearest neighbour search represents a significant bottleneck for a wide variety of applications. We have introduced this problem in Chapter 3 and discussed the trade-offs associated with various indexing techniques. Then in Chapter 4 we evaluated a number of simple occlusion rules that are capable of producing novel indexing graphs for the approximate nearest neighbour search problem. The edges of our graphs reflect the desirable structures of low dimensional manifolds while they index higher dimensional spaces. The simplicity of these occlusion rules has allowed us to formulate computationally efficient algorithms for constructing and searching

these graphs. Thorough exploration of many feasible indexing schemes has then culminated in the development of our Fast Approximate Nearest Neighbour Graphs (FANNGs). Chapter 5 further demonstrated the versatility of these graphs by demonstrating favourable performance when indexing binary feature spaces.

Chapter 6 applied FANNGs to the task of learning robust feature mappings. Here, the additional information provided by FANNG queries has enabled new state-of-the-art results in the area of transfer learning. Our first learning framework utilised local feature correspondences in order to accelerate and refine the training of triplet networks. Our smart mining procedure is able to efficiently produce challenging training samples, while our adaptive controller can automatically regulate their difficulty. Our second approach to feature learning is designed to utilise all of the correspondences found in local neighbourhoods of a learned feature space. Our approximation of a Gaussian kernel loss is able to efficiently construct a robust feature space that is suitable for transfer learning and classification tasks. Additionally, we have demonstrated the robustness of our indexing graphs with their ability to operate effectively after changes have been made to the indexed data.

7.1 Future Research Opportunities

My research has focused on alleviating common computational bottlenecks as well as understand the utility of information found in high dimensional visual data. Following these goals has led to the development of algorithms that improve upon existing computer vision systems. Continuing to re-evaluating the building blocks of these systems can lead to improvements that provide ongoing benefits in many areas. In relation to this, I believe that compelling future research opportunities can be found in the following areas:

- Further acceleration of our Hausdorff dimensionality formulation by utilising an approximate nearest neighbour method for finding relevant neighbours at each radius.

- Improving the construction of FANNGs by also allowing for the insertion and removal of individual vertices.
- Enabling the efficient indexing of larger datasets by combining the compact and memory efficient feature representations produced by quantisation methods with the reduced backtracking costs of indexing graphs.
- Applying the vast analytical power of machine learning to the selection of robust and efficient edges in approximate nearest neighbour indexing graphs.
- Leveraging the information provided by the indexing of neural network embedding spaces in order to refine the training of other machine learning frameworks.
- Transferring our methodology from applying FANNGs to feature learning into other fruitful areas that currently use correspondences or distance matrices and could benefit from the efficiency of an approximate method.
- Lastly, visual data is just one of the many high dimensional data sources that could potentially benefit for the analysis techniques and algorithms we have discussed.

Bibliography

[1] B. Harwood and T. Drummond, "FANNG: Fast approximate nearest neighbour graphs", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 5713–5722 (cit. on p. ii).

[2] B. Harwood, V. Kumar B G, G. Carneiro, I. Reid, and T. Drummond, "Smart mining for deep metric learning", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2017 (cit. on p. ii).

[3] B. J. Meyer, B. Harwood, and T. Drummond, "Deep metric learning and image classification with nearest neighbour gaussian kernels", in *25th IEEE international conference on image processing (ICIP)*, IEEE, 2018, pp. 151–155 (cit. on p. ii).

[4] D. G. Lowe, "Object recognition from local scale-invariant features", in *Proceedings of the 7th IEEE international conference on computer vision (ICCV)*, IEEE, vol. 2, 1999, pp. 1150–1157 (cit. on p. 4).

[5] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 33, no. 1, pp. 117–128, 2011 (cit. on pp. 4, 46).

[6] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding", in *IEEE international conference on acoustics, speech and signal processing (ICASSP)*, IEEE, 2011, pp. 861–864 (cit. on pp. 4, 46).

[7] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope", *International journal of computer vision (IJCV)*, vol. 42.3, pp. 145–175, 2001 (cit. on p. 4).

[8] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, IEEE, 2011, pp. 2564–2571 (cit. on pp. 4, 114).

[9] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz, "Multi-view stereo for community photo collections", in *Proceedings of the 11th IEEE international conference on computer vision (ICCV)*, IEEE, 2007, pp. 1–8 (cit. on p. 4).

[10] N. Snavely, S. M. Seitz, and R. Szeliski, "Modeling the world from internet photo collections", *International journal of computer vision (IJCV)*, vol. 80, no. 2, pp. 189–210, 2008 (cit. on p. 4).

[11] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary robust invariant scalable keypoints", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, IEEE, 2011, pp. 2548–2555 (cit. on pp. 4, 114).

[12] A. Alahi, R. Ortiz, and P. Vandergheynst, "FREAK: Fast retina keypoint", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2012, pp. 510–517 (cit. on pp. 4, 114).

[13] T. Trzcinski, M. Christoudias, and V. Lepetit, "Learning image descriptors with boosting", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 37, no. 3, pp. 597–610, 2015 (cit. on pp. 4, 114).

[14] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 30, no. 11, pp. 1958–1970, 2008 (cit. on p. 4).

[15] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The caltech-ucsd birds-200-2011 dataset", 2011 (cit. on p. 5).

[16] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization", in *Proceedings of the IEEE international conference on computer vision workshops*, 2013, pp. 554–561 (cit. on p. 5).

[17] V. Guruswami. (2012). Computer science theory for the information age, [Online]. Available: `http://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/` (visited on 11/30/2018) (cit. on p. 11).

[18] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space", in *International conference on database theory*, Springer, 2001, pp. 420–434 (cit. on p. 11).

[19] H. D. Mittelmann and F. Vallentin, "High-accuracy semidefinite programming bounds for kissing numbers", *Experimental mathematics*, vol. 19, no. 2, pp. 175–179, 2010 (cit. on p. 25).

[20] R. Basri and D. W. Jacobs, "Lambertian reflectance and linear subspaces", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 25, no. 2, pp. 218–233, 2003 (cit. on p. 28).

[21] B. B. Mandelbrot, *The fractal geometry of nature*. WH Freeman New York, 1983, vol. 173 (cit. on p. 30).

[22] B. C. Russell, A. Torralba, C. Liu, R. Fergus, and W. T. Freeman, "Object recognition by scene alignment", *Advances in neural information processing systems (NIPS)*, pp. 1241–1248, 2007 (cit. on p. 38).

[23] G. Shakhnarovich, P. Viola, and T. Darrell, "Fast pose estimation with parameter-sensitive hashing", in *Proceedings of the 9th IEEE international conference on computer vision (ICCV)*, IEEE, 2003, p. 750 (cit. on pp. 38, 41).

[24] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, vol. 2, 2006, pp. 2161–2168 (cit. on pp. 38, 44).

[25] P. Newman and K. Ho, "SLAM-loop closing with visually salient features", in *Proceedings of the IEEE international conference on robotics and automation (ICRA)*, IEEE, 2005, pp. 635–642 (cit. on p. 38).

[26] J. Cheng, C. Leng, J. Wu, H. Cui, H. Lu, *et al.*, "Fast and accurate image matching with cascade hashing for 3d reconstruction", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2014, pp. 1–8 (cit. on p. 38).

[27] O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2008, pp. 1–8 (cit. on p. 38).

[28] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing", in *Proceedings of the 25th international conference on very large data bases (VLDB)*, vol. 99, 1999, pp. 518–529 (cit. on p. 40).

[29] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions", in *The 47th annual IEEE symposium on foundations of computer science (FOCS)*, IEEE, 2006, pp. 459–468 (cit. on pp. 41, 115).

[30] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: Efficient indexing for high-dimensional similarity search", in *Proceedings of the 33rd international conference on very large data bases*, VLDB Endowment, 2007, pp. 950–961 (cit. on p. 42).

[31] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing", in *Advances in neural information processing systems (NIPS)*, 2009, pp. 1753–1760 (cit. on p. 43).

[32] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search", in *Proceedings of the 12th IEEE international conference on computer vision (ICCV)*, IEEE, 2009, pp. 2130–2137 (cit. on p. 43).

[33] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for scalable image retrieval", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2010, pp. 3424–3431 (cit. on p. 43).

[34] J. He, W. Liu, and S.-F. Chang, "Scalable similarity search with optimized kernel hashing", in *Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining*, ACM, 2010, pp. 1129–1138 (cit. on p. 43).

[35] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu, "Complementary hashing for approximate nearest neighbor search", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, IEEE, 2011, pp. 1631–1638 (cit. on p. 43).

[36] B. Kang and K. Jung, "Robust and efficient locality sensitive hashing for nearest neighbor search in large data sets", in *Advances in neural information processing systems workshops*, 2012, pp. 1–8 (cit. on p. 43).

[37] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon, "Spherical hashing", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2012, pp. 2957–2964 (cit. on p. 43).

[38] E. Silva, T. Teixeira, G. Teodoro, and E. Valle, "Large-scale distributed locality-sensitive hashing for general metric data", in *Proceedings of the international conference on similarity search and applications (SISAP)*, Springer, 2014, pp. 82–93 (cit. on p. 43).

[39] A. Andoni and I. Razenshteyn, "Optimal data-dependent hashing for approximate near neighbors", in *Proceedings of the 47th annual ACM symposium on theory of computing*, ACM, 2015, pp. 793–801 (cit. on p. 43).

[40] J. H. Friedman, F. Baskett, and L. J. Shustek, "An algorithm for finding nearest neighbors", *IEEE transactions on computers*, vol. 100, no. 10, pp. 1000–1006, 1975 (cit. on p. 43).

[41] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975 (cit. on p. 43).

[42] J. L. Bentley, "Multidimensional divide-and-conquer", *Communications of the ACM*, vol. 23, no. 4, pp. 214–229, 1980 (cit. on p. 43).

[43] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 1997, pp. 1000–1006 (cit. on p. 44).

[44] C. Silpa-Anan and R. Hartley, "Optimised KD-trees for fast image descriptor matching", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2008, pp. 1–8 (cit. on p. 44).

[45] J. Sivic and A. Zisserman, "Video Google: A text retrieval approach to object matching in videos", in *Proceedings of the 9th IEEE international conference on computer vision (ICCV)*, IEEE, 2003, pp. 1470–1477 (cit. on p. 44).

[46] G. Schindler, M. Brown, and R. Szeliski, "City-scale location recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2007, pp. 1–7 (cit. on p. 45).

[47] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces", in *ACM-SIAM symposium on discrete algorithms (SODA)*, vol. 93, 1993, pp. 311–321 (cit. on p. 45).

[48] N. Katayama and S. Satoh, "The SR-tree: An index structure for high-dimensional nearest neighbor queries", in *ACM special interest group on management of data (SIGMOD) record*, ACM, vol. 26, 1997, pp. 369–380 (cit. on p. 45).

[49] T. Liu, A. W. Moore, K. Yang, and A. G. Gray, "An investigation of practical approximate nearest neighbor algorithms", in *Advances in neural information processing systems (NIPS)*, 2005, pp. 825–832 (cit. on p. 45).

[50] H. Lejsek, F. H. Ásmundsson, B. Þ. Jónsson, and L. Amsaleg, "NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 31, no. 5, pp. 869–883, 2009 (cit. on p. 45).

[51] H. Lejsek, B. Þ. Jónsson, and L. Amsaleg, "NV-Tree: Nearest neighbors at the billion scale", in *Proceedings of the 1st ACM international conference on multimedia retrieval*, ACM, 2011, p. 54 (cit. on p. 45).

[52] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 36, no. 11, pp. 2227–2240, 2014 (cit. on pp. 45, 108, 115).

[53] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration", in *Proceedings of the international conference on computer vision theory and applications (VISAPP)*, 2009, pp. 331–340 (cit. on p. 45).

[54] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search", *European conference on computer vision (ECCV)*, pp. 304–317, 2008 (cit. on p. 46).

[55] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2013, pp. 2946–2953 (cit. on p. 47).

[56] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 36, no. 4, pp. 744–755, 2014 (cit. on p. 47).

[57] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2014, pp. 2321–2328 (cit. on p. 47).

[58] M. Norouzi and D. J. Fleet, "Cartesian k-means", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2013, pp. 3017–3024 (cit. on p. 47).

[59] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval", *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, vol. 35, no. 12, pp. 2916–2929, 2013 (cit. on p. 47).

[60] T. Zhang, C. Du, and J. Wang, "Composite quantization for approximate nearest neighbor search", in *Proceedings of the 31st international conference on machine learning (ICML)*, vol. 32, 2014, pp. 838–846 (cit. on p. 47).

[61] T. Zhang, G.-J. Qi, J. Tang, and J. Wang, "Sparse composite quantization", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 4548–4556 (cit. on p. 47).

[62] J. Martinez, J. Clement, H. H. Hoos, and J. J. Little, "Revisiting additive quantization", in *European conference on computer vision (ECCV)*, Springer, 2016, pp. 137–153 (cit. on p. 47).

[63] X. Wang, T. Zhang, G.-J. Qi, J. Tang, and J. Wang, "Supervised quantization for similarity search", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 2018–2026 (cit. on p. 47).

[64] H. Jégou, M. Douze, and C. Schmid, "Improving bag-of-features for large scale image search", *International journal of computer vision (IJCV)*, vol. 87, no. 3, pp. 316–336, 2010 (cit. on p. 48).

[65] J. Wang and S. Li, "Query-driven iterated neighborhood graph search for large scale indexing", in *Proceedings of the 20th ACM international conference on multimedia*, ACM, 2012, pp. 179–188 (cit. on p. 48).

[66] J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo, "Fast neighborhood graph search using cartesian concatenation", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2013, pp. 2128–2135 (cit. on p. 48).

[67] G. Navarro, "Searching in metric spaces by spatial approximation", *International journal on Very Large Data Bases (VLDB)*, vol. 11, no. 1, pp. 28–46, 2002 (cit. on pp. 48, 62).

[68] K. L. Clarkson, "Nearest-neighbor searching and metric space dimensions", *Nearest-neighbor methods for learning and vision: theory and practice*, pp. 15–59, 2006 (cit. on p. 49).

[69] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph", in *Proceedings of the international joint conference on artificial intelligence (IJCAI)*, vol. 22, 2011, p. 1312 (cit. on p. 49).

[70] J. C. Gower and G. J. Ross, "Minimum spanning trees and single linkage cluster analysis", *Applied statistics*, pp. 54–64, 1969 (cit. on p. 49).

[71] M. A. Wong and T. Lane, "A kth nearest neighbour clustering procedure", in *Computer science and statistics: Proceedings of the 13th symposium on the interface*, Springer, 1981, pp. 308–311 (cit. on p. 49).

[72] J. Chua and P. Tischer, "Minimal cost spanning trees for nearest-neighbour matching", *Computational intelligence for modelling, control and automation: Intelligent image processing, data analysis and information retrieval*, pp. 7–12, 1999 (cit. on p. 49).

[73] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures", in *Proceedings of the 20th international conference on world wide web*, ACM, 2011, pp. 577–586 (cit. on p. 50).

[74] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-nn graph construction for visual descriptors", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2012, pp. 1106–1113 (cit. on p. 50).

[75] Y.-m. Zhang, K. Huang, G. Geng, and C.-l. Liu, "Fast kNN graph construction with locality sensitive hashing", in *Joint european conference on machine learning and knowledge discovery in databases*, Springer, 2013, pp. 660–674 (cit. on p. 50).

[76] Y. Lifshits and S. Zhang, "Combinatorial algorithms for nearest neighbors, near-duplicates and small-world design", in *Proceedings of the 20th annual*

ACM-SIAM symposium on discrete algorithms, Society for Industrial and Applied Mathematics, 2009, pp. 318–326 (cit. on p. 51).

[77] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces", *Similarity search and applications*, pp. 132–147, 2012 (cit. on p. 51).

[78] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs", *Information systems*, vol. 45, pp. 61–68, 2014 (cit. on p. 51).

[79] G. T. Toussaint, "The relative neighbourhood graph of a finite planar set", *Pattern recognition*, vol. 12, no. 4, pp. 261–268, 1980 (cit. on p. 60).

[80] R. Urquhart, "Algorithms for computation of relative neighbourhood graph", *Electronics Letters*, vol. 16, no. 14, pp. 556–557, 1980 (cit. on p. 60).

[81] J. W. Jaromczyk and M. Kowaluk, "A note on relative neighborhood graphs", in *Proceedings of the 3rd annual symposium on computational geometry*, ACM, 1987, pp. 233–241 (cit. on p. 62).

[82] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions.", in *ACM-SIAM symposium on discrete algorithms (SODA)*, vol. 93, 1993, pp. 271–280 (cit. on p. 62).

[83] B. Naidan, L. Boytsov, and E. Nyberg, "Permutation search methods are efficient, yet faster search is possible", *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 12, pp. 1618–1629, 2015 (cit. on p. 108).

[84] T. Trzcinski, M. Christoudias, V. Lepetit, and P. Fua, "Learning image descriptors with the boosting-trick", in *Advances in neural information processing systems (NIPS)*, 2012, pp. 269–277 (cit. on p. 114).

[85] T. Trzcinski, M. Christoudias, P. Fua, and V. Lepetit, "Boosting binary keypoint descriptors", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, IEEE, 2013, pp. 2874–2881 (cit. on p. 114).

[86] Y. Feng, Y. Wu, and L. Fan, "Online learning of binary feature indexing for real-time SLAM relocalization", in *Asian conference on computer vision workshops*, Springer, 2014, pp. 206–217 (cit. on p. 114).

[87] R. Timofte, T. Tuytelaars, and L. Van Gool, "Naive bayes image classification: Beyond nearest neighbors", in *Asian conference on computer vision (ACCV)*, Springer, 2012, pp. 689–703 (cit. on p. 114).

[88] F. André, A.-M. Kermarrec, and N. Le Scouarnec, "Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan", *Proceedings of the VLDB Endowment (PVLDB)*, vol. 9, no. 4, pp. 288–299, 2015 (cit. on p. 115).

[89] T. Trzcinski, V. Lepetit, and P. Fua, "Thick boundaries in binary space and their influence on nearest-neighbor search", *Pattern recognition letters*, vol. 33, no. 16, pp. 2173–2180, 2012 (cit. on p. 115).

[90] L. Zhang, Y. Zhang, J. Tang, K. Lu, and Q. Tian, "Binary code ranking with weighted hamming distance", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2013, pp. 1586–1593 (cit. on p. 115).

[91] Y. Ma, H. Xie, Z. Chen, Q. Dai, Y. Huang, and G. Ji, "Fast search of binary codes with distinctive bits", in *Advances in multimedia information processing (PCM)*, Springer, 2014, pp. 274–283 (cit. on p. 115).

[92] Y. Ma, H. Zou, H. Xie, and Q. Su, "Fast search with data-oriented multi-index hashing for multimedia data", *KSII transactions on internet and information systems (TIIS)*, vol. 9, no. 7, pp. 2599–2613, 2015 (cit. on p. 115).

[93] I. Torres-Xirau, J. Salvador, and E. Pérez-Pellitero, "Fast approximate nearest-neighbor field by cascaded spherical hashing", in *European conference on computer vision (ECCV)*, Springer, 2014, pp. 461–475 (cit. on p. 115).

[94] M. Muja and D. G. Lowe, "Fast matching of binary features", in *Proceedings of the 9th IEEE conference on computer and robot vision (CRV)*, IEEE, 2012, pp. 404–410 (cit. on pp. 116, 124).

[95] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998 (cit. on p. 129).

[96] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems (NIPS)*, 2012, pp. 1097–1105 (cit. on p. 129).

[97] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines", in *Proceedings of the 27st international conference on machine learning (ICML)*, 2010, pp. 807–814 (cit. on p. 130).

[98] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *3rd International conference on learning representations (ICLR)*, 2015 (cit. on p. 130).

[99] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 1–9 (cit. on pp. 130, 160).

[100] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778 (cit. on p. 130).

[101] E. Hoffer and N. Ailon, "Deep metric learning using triplet network", in *International workshop on similarity-based pattern recognition (SIMBAD)*, Springer, 2015, pp. 84–92 (cit. on p. 137).

[102] J. Wang, Y. Song, T. Leung, C. Rosenberg, J. Wang, J. Philbin, B. Chen, and Y. Wu, "Learning fine-grained image similarity with deep ranking", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2014, pp. 1386–1393 (cit. on pp. 137, 138).

[103] J. Bromley, J. W. Bentz, L. Bottou, I. Guyon, Y. LeCun, C. Moore, E. Säckinger, and R. Shah, "Signature verification using a "Siamese" time delay neural network", *International journal of pattern recognition and artificial intelligence (IJPRAI)*, vol. 7, no. 04, pp. 669–688, 1993 (cit. on p. 137).

[104] A. Shrivastava, A. Gupta, and R. Girshick, "Training region-based object detectors with online hard example mining", *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016 (cit. on p. 138).

[105] K. SUNG, "Learning and example selection for object and pattern detection", *Ph. D thesis, MIT AI laboratory*, 1996 (cit. on p. 138).

[106] E. Simo-Serra, E. Trulls, L. Ferraz, I. Kokkinos, P. Fua, and F. Moreno-Noguer, "Discriminative learning of deep convolutional feature point descriptors", in *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2015, pp. 118–126 (cit. on p. 138).

[107] X. Han, T. Leung, Y. Jia, R. Sukthankar, and A. C. Berg, "MatchNet: Unifying feature and metric learning for patch-based matching", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 3279–3286 (cit. on p. 138).

[108] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 815–823 (cit. on pp. 138, 162).

[109] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese, "Deep metric learning via lifted structured feature embedding", *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016 (cit. on pp. 139, 162).

[110] V. Kumar B G, G. Carneiro, and I. Reid, "Learning local image descriptors with deep siamese and triplet convolutional networks by minimising global loss functions", *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016 (cit. on pp. 139, 162).

[111] E. Ustinova and V. Lempitsky, "Learning deep embeddings with histogram loss", in *Advances in neural information processing systems (NIPS)*, 2016, pp. 4170–4178 (cit. on p. 139).

[112] H. Oh Song, S. Jegelka, V. Rathod, and K. Murphy, "Deep metric learning via facility location", in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2017 (cit. on pp. 139, 163).

[113] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge", *International journal of computer vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015 (cit. on p. 160).

[114] K. Sohn, "Improved deep metric learning with multi-class n-pair loss objective", in *Advances in neural information processing systems (NIPS)*, 2016, pp. 1849–1857 (cit. on p. 162).

[115] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE", *Journal of machine learning research (JMLR)*, vol. 9, no. Nov, pp. 2579–2605, 2008 (cit. on p. 164).